Project 1 Writeup

Question 1: Remus

Main Idea

This code is vulnerable because `gets` doesn't check the length of the input from user, which provides us a way to write past the buffer and overwrite RIP of `orbit`.

Magic Numbers

I first determined the the address of `buf`, whose value is `0xffffdd48`. Then according to command `info frame`, RIP is at `0xffffdd5c`. By doing so, I learned that the location of return address is 20 bytes away from the start of buffer

Exploit Structure

The exploit includes three parts:

1. Write 20 dummy characters to overwrite `buf`, the compiler padding, and the SFP.
2. Overwrite the return address with the start of `SHELLCODE`, which is directly after the location of RIP (0xffffdd60).
3. Write the `SHELLCODE`.

This causes the orbit function to execute `SHELLCODE` at `0xffffdd60` when it returns.

Exploit GDB Output

I have to leave out this part, since I can't copy anything in QEMU interface...

Question 2: Spica

Main Idea

This code is vulnerable because the type of `size` in `display` is `int8_t`, a signed type, while the parameter in `fread` is unsigned. Thus the program has to do type cast when calling `fread`. Suppose `size` is negative, then `size < 128`, but after type cast, `size` is greater than 128, which gives us an idea of reading more bytes and write past `msg` in `display`.

Magic Numbers

I first determined the address of `msg`, whose value is `0xffffdc78`. Then I used `info frame` command and got the RIP of display was at `0xffffdd0c`. By doing so, I learned that the location of return address is `0x94` bytes away from the start of `msg`.

Exploit Structure

The exploit includes four parts:

1. Write the first byte represents the length of following message, which should be `b'\x98'`(0x94 + 4(the length of return address)).
2. Write the `SHELLCODE` from the start of `msg`.
3. Write `0x94 - len(SHELLCODE)` dummy characters to overwrite the rest `msg`, paddings, and the SFP.
4. Overwrite RIP with the start of `SHELLCODE`.

The exploit causes the function to start executing `SHELLCODE` at `0xffffdc78` when the it returns.

Exploit GDB Output

Question 3: Polaris

Main Idea

This code is vulnerable because in `dehexify`, when `c.buffer[i] == '\\'  && c.buffer[i + 1] == 'x'`, the function needs to peek `buffer[i + 2] and buffer[i + 3]`, which potentially leaves out the terminator of input and print out the canary value.

Magic Numbers

I first determined the address of `c.buffer`, whose value is `0xffffdd1c`. Then I determined the canary value is at `0xffffdd2c`, which is right after `c.buffer`. Last, I determined the RIP is at `0xffffdd3c`. By doing so, I knew that there is 12 bytes padding between RIP and canary value.

Exploit Structure

I sent two massages to the program:

The first message aims to trick program into printing canary value. Since every input will terminates with a `\x00` byte, I need to restrict my input's length to be 15 bytes. Besides, to leave out the terminator, `c.buffer[12]` should be`'\\'` and `c.buffer[13]` should be `'x'`.
Thus this massage includes two parts:

1. Write 12 dummy bytes to `c.buffer`.
2. Write `b'\\x\x00\x00'`.

Then the output should be 12 dummy characters + '\x00' + canary value + something else. I store the canary value, whose index is from 12 to 15.

The second message aims to overwrite the canary value with the original value so that the program won't trigger a segmentation fault and redirect the execution to our `SHELLCODE`. This message includes 6 parts:

1. Write `'\x00'` as the first byte to terminate the loop in function `dehexify`.
2. Write 15 dummy bytes to fill the rest of `c.buffer`.
3. Overwrite the canary value with itself.
4. Write 12 dummy bytes to overwrite the padding and SFP.
5. Overwrite the RIP with the start of `SHELLCODE`.
6. Write the `SHELLCODE`.

Exploit GDB Output

Question 4: Vega

Main Idea

The code is vulnerable because there is an off-by-one problem in function `flip`. If the length of input is 65 bytes, then the last byte will overwrite the least significant byte of SFP.

Magic Numbers

I first determined the address of `buf`, whose value is `0xffffdc90`. Then I used `info frame` in function `invoke` to determine the SFP is at `0xffffdcd0` and the value is `0xffffdcdc`. By doing so, I learned that SFP is 64 bytes away from the start of `buf`. So we can overwrite the least significant byte of SFP of invoke. In this question, we overwrite it with `\x90`. When it returns, the code will execute

```
leave
  = mov %ebp, %esp
    pop %ebp

ret
  = pop %eip
```

After executing `leave`, we have `%ebp = 0xffffdc90` and `%esp` points to RIP. After returning from `invoke`, the code returns to `dispatch` and is going to execute `leave` and `ret` again. After executing `leave`, we have `%esp = 0xffffdc94`. If the program executes `ret`, then `%eip` will get the value stored in `0xffffdc94`. So this is the address we need to store our `SHELLCODE` address.

Finally, I used `p environ[4]` and got

```
$1 = 0xffffdfaa "EGG=xxx" (xxx represents our SHELLCODE)
```

I inferred that the start of `SHELLCODE` is `0xffffdfae`.

Exploit Structure

The exploit includes four parts:

1. Write four dummy bytes to overwrite buffer.

2. Write four bytes of number to be flipped.
3. Write 56 dummy bytes to overwrite the rest of buffer.
4. Write one byte to overwrite the least significant byte of SFP.

Exploit GDB Output

Question 5: Deneb

Main Idea

This code is vulnerable because it checks the file length before reading content. Thus the file could be extended and write past the buffer.

Magic Numbers

I first determined the address of `buf`, whose value is `0xffffdcc8`. Then I used `info frame` and got the RIP is at `0xffffdd5c`. By doing so, I learned that the location of RIP is 0x94 bytes away from the start of `buf`.

Exploit Structure

The exploit includes two messages:

The first message is the length of our final file length. We will discuss it later.

The second massage is to write the `hack` file, which includes three parts:
1. Write `0x94` bytes of dummy bytes to overwrite `buf`, paddings and SFP.
2. Overwrite the RIP with the start of `SHELLCODE`.
3. Write the `SHELLCODE`.

The length of file is `0x94 + 4 + len(SHELLCODE) = 224`, so the first message should be 224.

Exploit GDB Output

Question 6: Antares

Main Idea

This code is vulnerable because it suffers Format String Vulnerability. We can overwrite the return address in two times with option `%hn`, and redirect the code to the start of `SHELLCODE`.

Magic Numbers

I first determined the start of `SHELLCODE` is at `0xffffdeb2`. Then I found that `buf` starts at `0xffffdc80`. Besides, when the program calls `printf(buf)`, the argument `buf` is located at `0xffffdc40`. By doing so, we know that the start of `buf` is 15 words from argument `buf`. Finally, I determined the RIP of `calibrate` is at `0xffffdc6c`. Thus second half is `0xdc6c`, which locates at `0xffffdc6c` and first half is `0xffff`, which locates at `0xffffdc6e`.

Exploit Structure

The exploit includes many parts...

1. This part includes 4 dummy bytes + `'\x6c\xdc\xff\xff'` + 4 dummy bytes + `'\x6e\xdc\xff\xff'`
2. This part includes `"%c" * 15` to print the 15 words between start of `buf` and argument `buf`.
3. This part includes many options, since we use them to overwrite the RIP. The first option should be `"%" + str(0xdc6c - (16 + 15))" + "u"`, the second option should be `"%hn"`, the third option should be `"%" + str(0xffff - 0xdc6c)" + "u"` and the last should be `"%hn"`.

Exploit GDB Output

Question 7: Rigel

Main Idea

This code is vulnerable because `err_ptr` can do us a favor in redirecting the execution to `SHELLCODE` even if there is ALSR.

Magic Numbers

I first determined the location of `err_ptr` is `0xffb1d080`. Then I determined the start of `buf` is at `0xffb1cfec`. Besides, I determined that the RIP of `secure_gets` is at `0xffb1d07c`. By doing so, I learned that RIP is 0x90 bytes away from the start of `buf` and location of `err_ptr` is right after RIP. I used `ret2ret` to exploit this program, so I found an address of `ret` instruction, whose value is `0x08049472`

Exploit Structure

The exploit includes 3 parts:

1. Write as many `nop` as possible to increase the exploiting possibility.
2. Write the `SHELLCODE`
3. Overwrite the RIP with the address of `ret` instruction.

Exploit GDB Output