

Challenge 5.


We found a suspicious binary running on one of our servers. [Here](#) is the binary file for you to see. The interface below allows you to send an input to the binary. Can you see if you can make it output the vaccine component?

1. I downloaded the binary file to inspect and see if there was any obvious answer to what could be submitted. Inside the binary file the following stood out:

```
.....f.....~.....  
...libc.so.6._IO_stdin_used.printf.read.system.__libc_sta  
rt_main.__gmon_start__.GLIBC_2.0.....  
      ii      N      ii
```

This looked like something being with a c library.

2. Putting some input into the message field returned some interesting results.

User provided 5 bytes. Buffer content is: sdfd 

.....

The extra characters I was getting at the end made me think there was a buffer overflow happening.

3. I tried supplying 129 bytes and then only 1 to see what extra memory I could access. Only garbage results were returned. I also noticed that there was a max length of input around 400 bytes.
4. I changed it to be executable via `chmod +x buff` to run some tests on my virtual machine.
5. At this point I used boomerang to attempt to gain the source code from the binary. The resulting source code wasn't intelligible.



```
Unknown library function __libc_start_main  
got filename crtstuff.c for main  
got filename crtstuff.c for overflow  
after removing matching assigns (t[WILDSTR]).  
after removing matching assigns (%pc).  
### WARNING: iteration limit exceeded for dfaTypeAnalysis of procedure overflow ###  
in proc overflow adding addrExp r28{0} - 516 to local table  
in proc overflow adding addrExp r28{0} - 4 to local table  
in proc overflow adding addrExp r28{0} - 532 to local table  
in proc overflow adding addrExp r28{0} - 536 to local table  
in proc overflow adding addrExp r28{0} - 540 to local table
```

6. Time to bring out the big guns and see if we can reverse engineer the file with Ghidra. A little internal inspection reveals something quite interesting.

```
void helper(void)
{
    system("cat flag.txt");
    return;
}
```

7. From close inspection it looks like this function isn't used. This made me wonder if I could overflow the buffer and pass an address to the function that would result in the flag being printed out. I noticed in a function that is being called an array of length 500.

```
2 undefined4 overflow(void)
3
4 {
5     undefined local_204 [500];
6     ssize_t local_10;
7
8     local_10 = read(0, local_204, 768);
9     printf("\nUser provided %d bytes. Buffer content is: %s\n", local_10, local_204);
10    return 0;
11 }
```

Providing a 500 character string. Followed by an address (I tried 0804847d) didn't seem to do much. But I could see the address part wasn't being returned as a string because it overflowed the buffer. I tried writing the address backwards as is often the case for loading it in hex.

8. At this point a bash script seemed like a worthwhile option. I setup a fake flag.txt file for the buff program to cat and let the automation run.

```
#!/bin/bash
# Create a fake flag.txt file
echo "flag.txt" > flag.txt

# Run the buff program
./buff 0804847d

# Check the output
cat flag.txt
```

9. Considering that would take all of eternity I also wrote up a c program to run the same functionality. That was considerably more efficient but nevertheless ineffective.

