**Challenge 5.**

We found a suspicious binary running on one of our servers. [Here](https://r0.nzcsc.org.nz/challenge5/buff) 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:



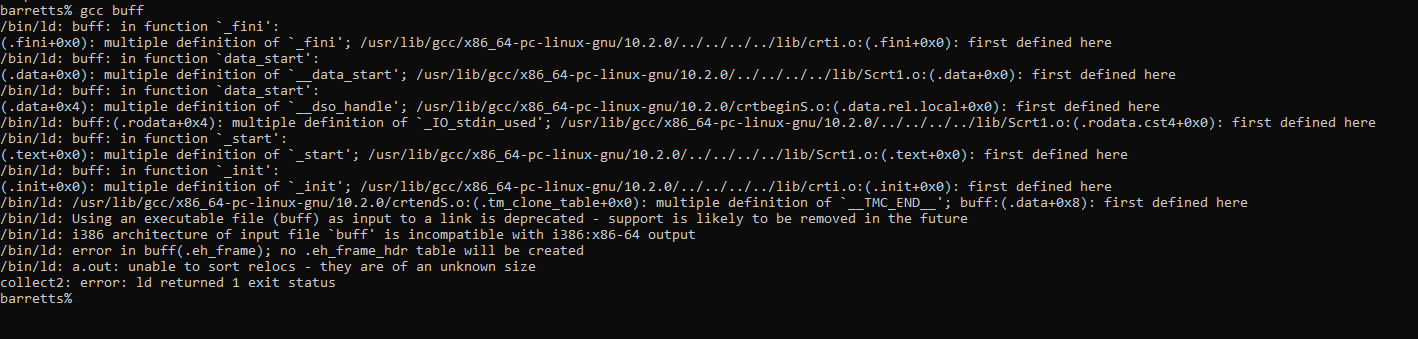
This looked like something being with a c library.

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

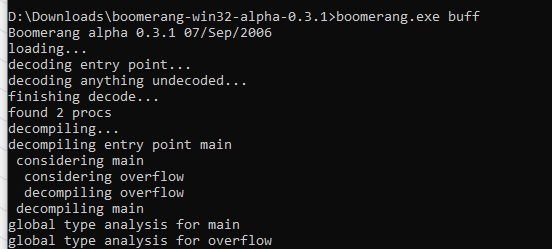


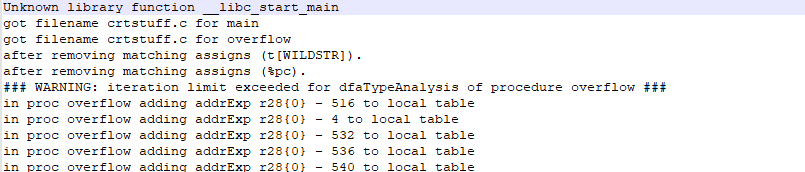
The extra characters I was getting at the end made me think they weren’t clearing they’re buffer after use.

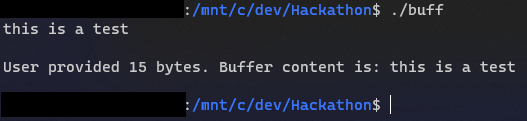
1. I tried supplying 129 bytes and then only 1 to see what extra memory I could access. Only garbage results were returned. I also noticed that there was a max length of input around 400 bytes.
2. Inspecting the page source didn’t show anything interesting.
3. It was time to run the binary file. An attempt to compile didn’t result in anything.



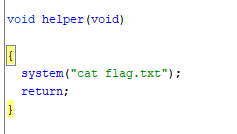
1. Attempts to run buff *./buff* resulted in permission denied.
2. At this point I used boomerang to attempt to gain the source code from the binary.



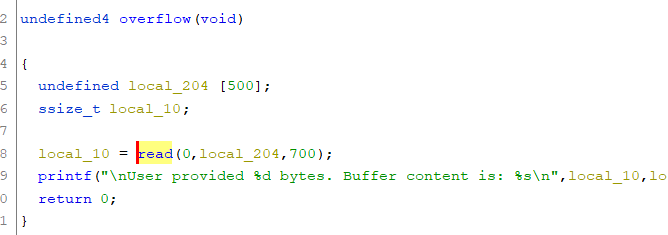
1. The resulting source code was a strange conglomeration of references that didn’t resemble code. 
2. I assumed from this that the program was just a reference call to other programs/tables/procedures. This would also explain why I couldn’t run it.
3. I changed it to be executable via *chmod +x buff*



1. 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.

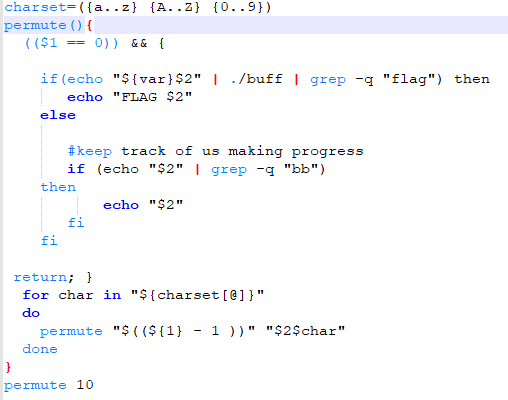


1. From close inspection it looks like this function isn’t used. This made me wonder if I could overflow the buffer and pass an address to the function that would result in the flag being printed out. I noticed in a function that is being called an array of length 500.



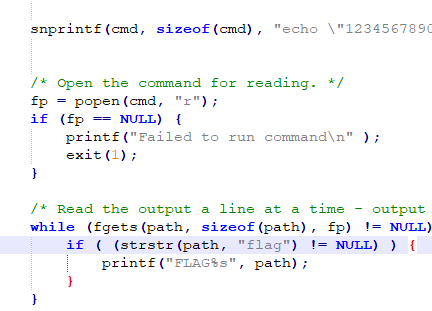
Providing a 500 character string. Followed by an address (I tried 0804847d) didn’t seem to do much. But I could see the address part wasn’t being returned as a string because it overflowed the buffer. I tried writing the address backwards as is often the case for loading it in hex.

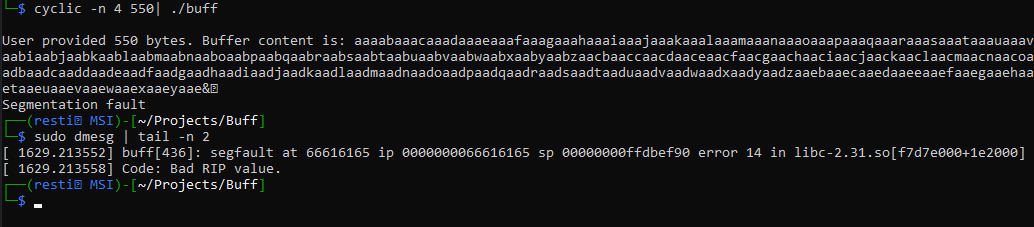
1. Analyzing the c code closer I decided the buffer overflow wouldn’t be that simple. The read function seemed to be a simple scanf into the array returning the size of the output.
2. At this point a bash script seemed like a worthwhile option. This tries every possible iteration of 10 characters at the end of 500 chars (too fill up the buffer).



I setup a fake flag.txt file for the buff program to cat and let it run. I didn’t really think it was going to work but it was definitely worth the shot.

1. 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.



1. Downloading cyclic and running against the binary allowed me to capture the memory address on which the program tried to jump to 0x66616165. From here I used cyclic to indicate the length of the buffer overflow needed which is 516 bytes. 



Using objdump we can also confirm the address of our function we would instead like to route the computer towards 0804847d (noting we need to write this backwards).

Unfortunately the server returns nothing. Which implies a segmentation fault.