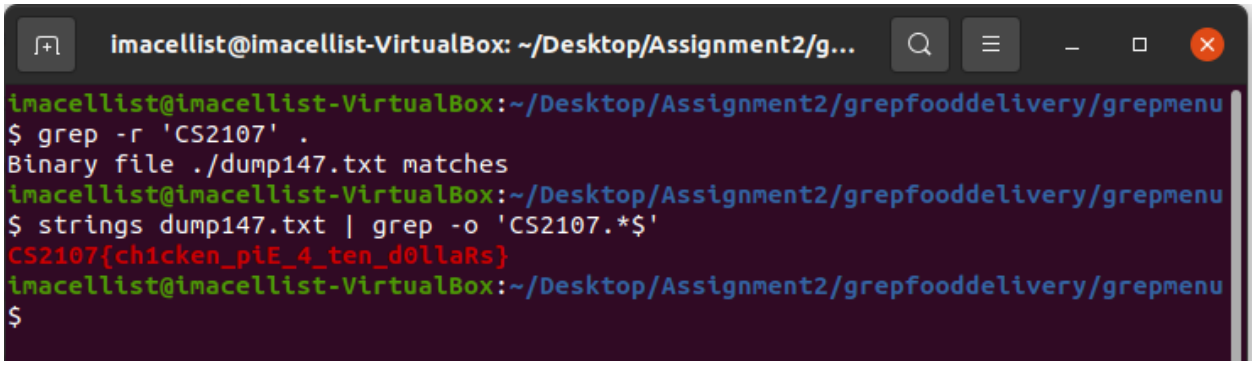
Section A: Warmup

A.1 Grep Food Delivery (2 Points)

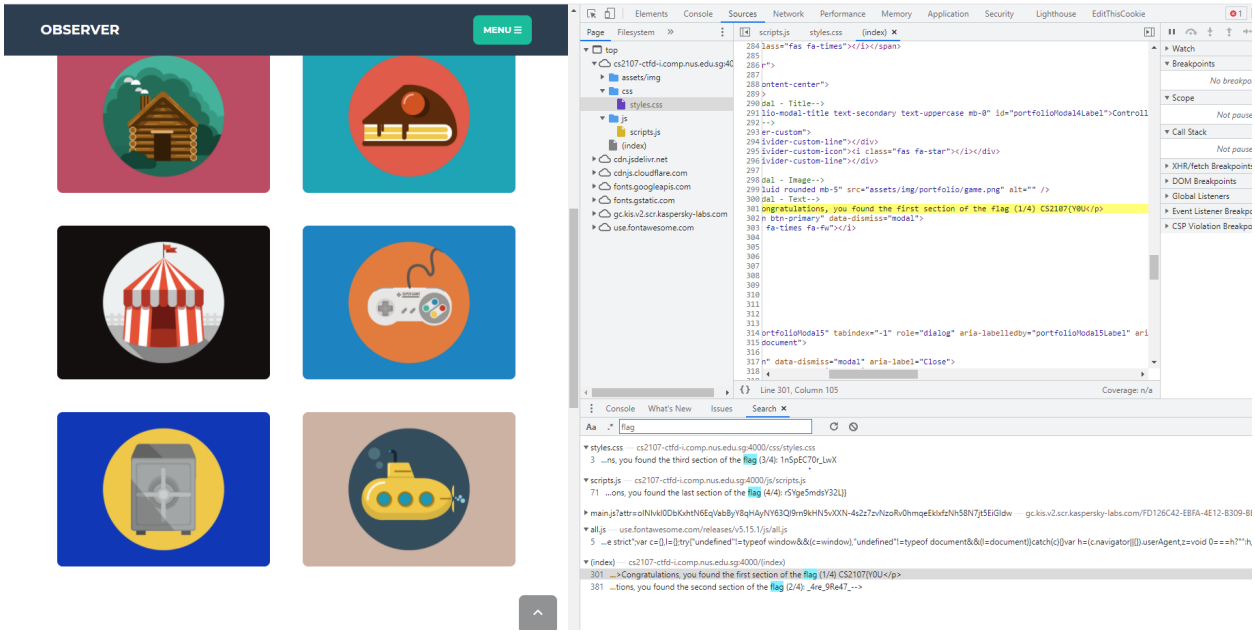
To search all files in the grepmenu directory recursively, we can use the -r option.

Seeing that there is a match in dump147.txt, we can pipe the strings of dump147.txt into grep 'CS2107.*$' to get the flag CS2107{ch1cken_piE_4_ten_d0llaRs}



A.2 Observer (6 Points)

Using the Search tab in DevTools, we can search all the source files for the keyword 'flag'.

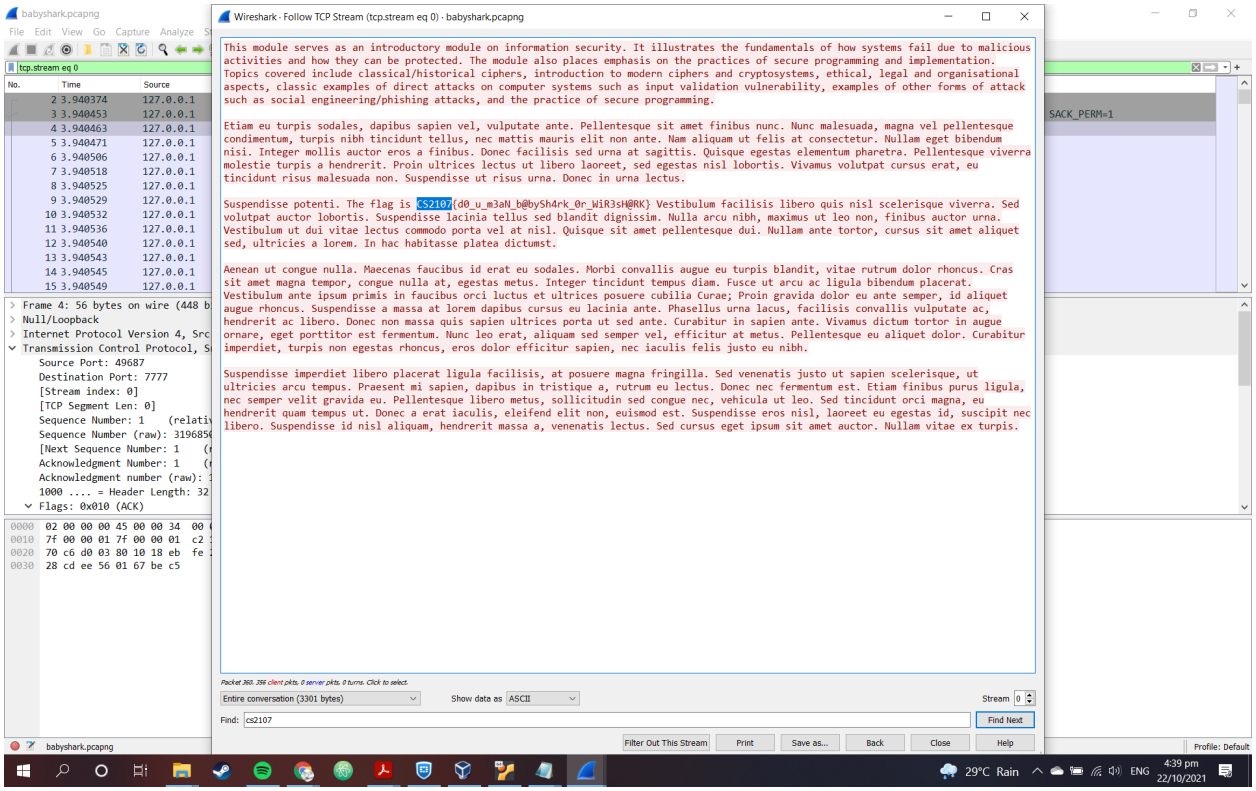


We find the flag split up into 4 parts inside index, scripts.js and styles.css forming CS2107{Y0U_4re_9Re47_1nSpEC70r_LwXrSYge5mdsY32L}.

Section B: Network

B.1 Babyshark (3 Points)

Using the Follow TCP Stream feature, we can follow a particular TCP conversation between 2 nodes. Right click on a TCP packet and select Follow -> TCP Stream to get the complete data.

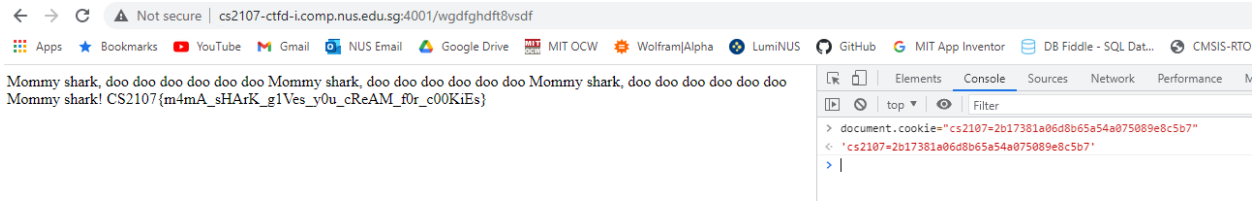


We find the flag to be CS2107{d0_u_m3aN_b@bySh4rk_0r_WiR3sH@RK}.

B.2 Mamashark (7 Points)

Using Wireshark, we can search for cookies by filtering packets with [frame contains “Cookie”] and Follow TCP Stream.

We see there was a HTTP GET request to /wgdfghdft8vsdf with Cookie: cs2107=2b17381a06d8b65a54a075089e8c5b7



By going to http://cs2107-ctfd-i.comp.nus.edu.sg:4001/wgdfghdft8vsdf and inserting the cookie with document.cookie=”cs2107=2b17381a06d8b65a54a075089e8c5b7”, we find the flag to be CS2107{m4mA_sHArK_g1Ves_y0u_cReAM_f0r_c00KiEs}.

Section C: Web

C.1 Secret Games (18 Points)

To get past Level 1: Red Light, Green Light, SQL injection of [‘ OR 1=1;--] is enough.such that the effective SQL statement becomes SELECT * FROM USERS WHERE username = ‘’ OR 1=1; -- (comment out everything else) which is True for all.

Level Up!
CS2107{9r33N_l19h7_R3D_L19H7_C644XCDx7yq9qP93}

We get the first flag to be CS2107{9r33N_l19h7_R3D_L19H7_C644XCDx7yq9qP93}.

To access Level 2, we need to enter the SHA256 hash of the first flag which we can do with an online SHA256 Encrypter to get
462661539c32b29a6972e7b0e636c0ab81a50c0ffb0aa140c4b145fd18a4a196

For Level 2: Ppopgi, simple SQL injection no longer works as an error message ‘SQL Injection detected’ pops up whenever the input contains [OR]. To bypass this, using common tricks like [admin’;--] worked.

Challenge Cleared!
CS2107{15_9lUc053_t45tY_54qkuxw2q5QJTP7W}

We get the second flag to be CS2107{15_9lUc053_t45tY_54qkuxw2q5QJTP7W} and the corresponding SHA256 hash **7e242f4a19819f873e8097b1f098be1a004d93f1eeed6d02a05c0ba1432df8e3**

For Level 3: Marbles Game, there seems to be SQL Injection detection for both OR and admin. To bypass this, we can simply use string concatenation like [a' || 'dmin';--] so admin is no longer detected.

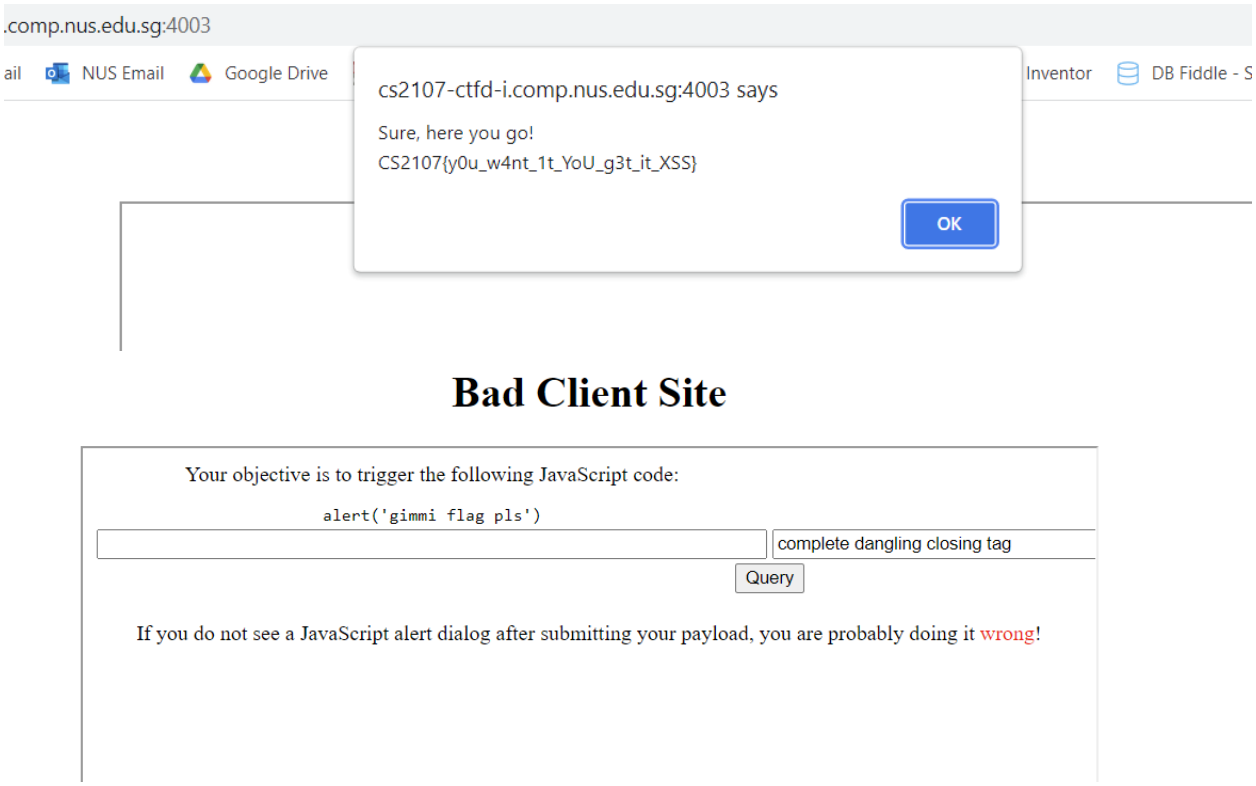
We finally get the final flag CS2107{4R3_w3_47_7h3_3nd_qgkZFF5rTAG99mkq}.

C.2 Bad Client Site (12 Points)

Inspecting the checker.html code, we see there is an input with id = ”payload” with value set to our input.

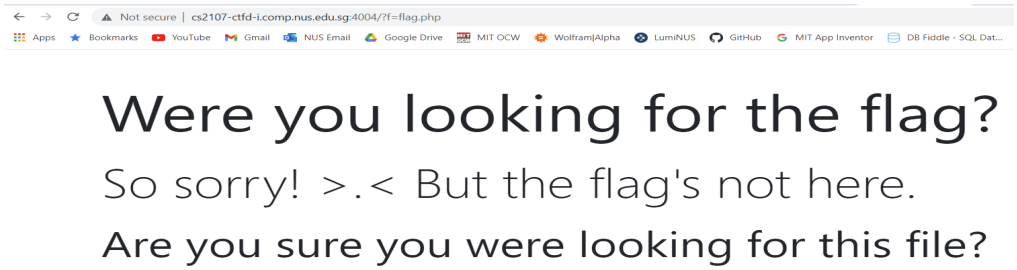
```
td align="center">Your objective is to trigger the following JavaScript code: <pre>alert('gimmi flag

td>
   <input type="text" size="70" id="payload" name="payload" value="">
/td>
```

In order to trigger the JavaScript code: alert('gimmi flag pls'), we can perform a reflected XSS attack by setting value to [”><script>alert(‘gimmi flag pls’)</script><input type="text" size="70" value="complete dandling closing tag]. This injects the alert script as well as completes the dandling “> closing tag from the initial input.

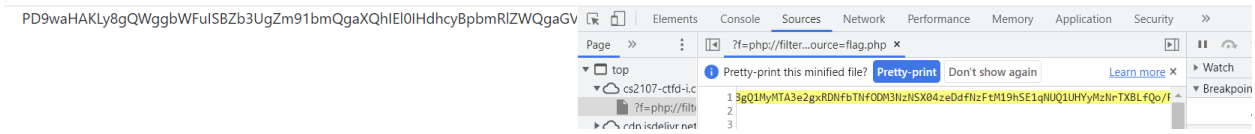


C.3 Secure Home (12 Points)

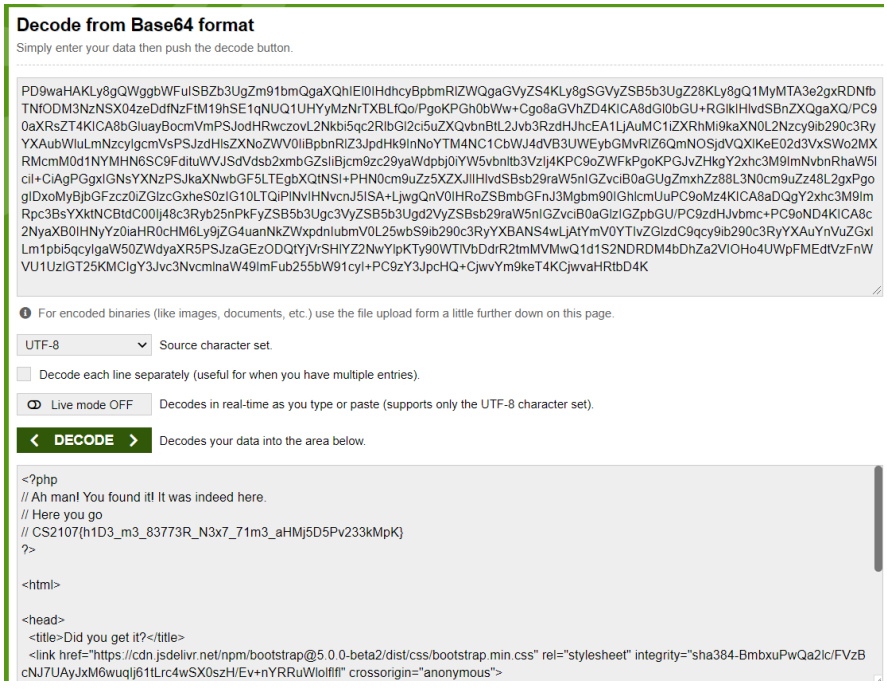
The task preamble tells us there is a file flag.php and that we can check it out using the URL parameter f. Trying ?f=flag.php, we get this message



However, we might be able to get the source code by exploiting the local file inclusion vulnerability. php://filter/convert.base64-encode/resource= allows us to include the local file and encode the output as base64.
By going to http://cs2107-ctfd-i.comp.nus.edu.sg:4004/?f=php://filter/convert.base64-encode/resource=flag.php, we get



And after decoding it,



We get the flag to be CS2107{h1D3_m3_83773R_N3x7_71m3_aHMj5D5Pv233kMpK}.

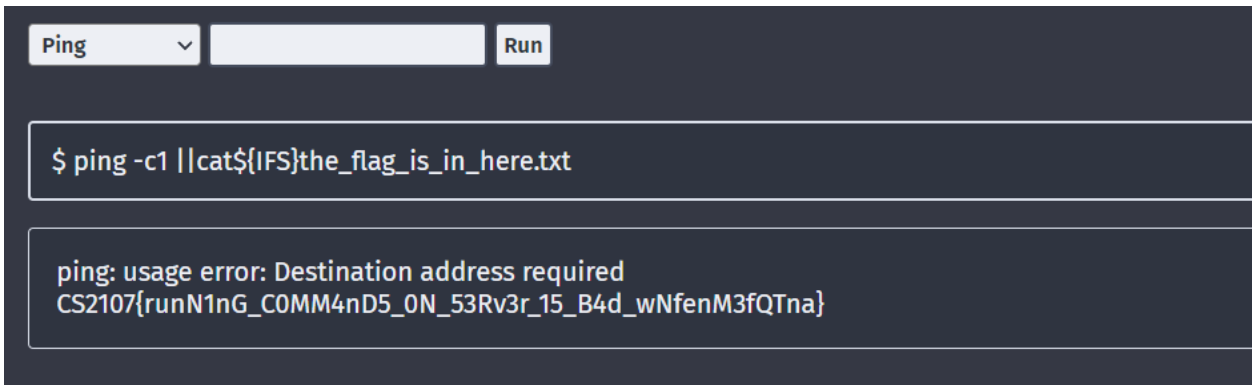
C.4 Favourite Tools (18 Points)

This seems to be a command line injection challenge.

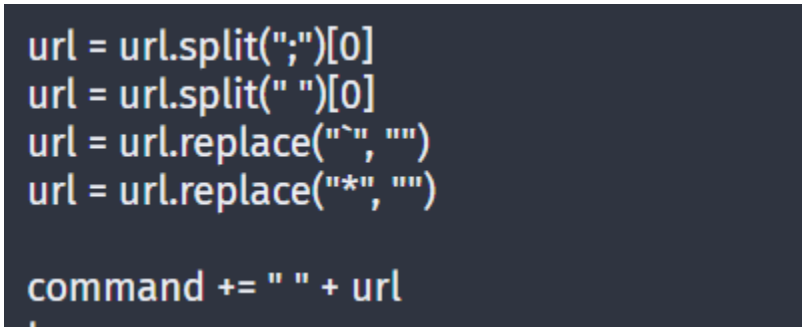
However, characters after a space character seem to be cut off. We also have to find a way to send a valid command after the ping.

Looking around the internet, using the || operator allows us to send a second command when the first ping fails. Using ||ls, we can list out the files that are accessible.

To bypass input filtering of the space character, we can use the alternative ${IFS} which is a special shell variable with default value <space><tab><newline>. With this, we can ||cat${IFS}the_flag_is_in_here.txt to get the flag CS2107{runN1nG_C0MM4nD5_0N_53Rv3r_15_B4d_wNfenM3fQTna}.



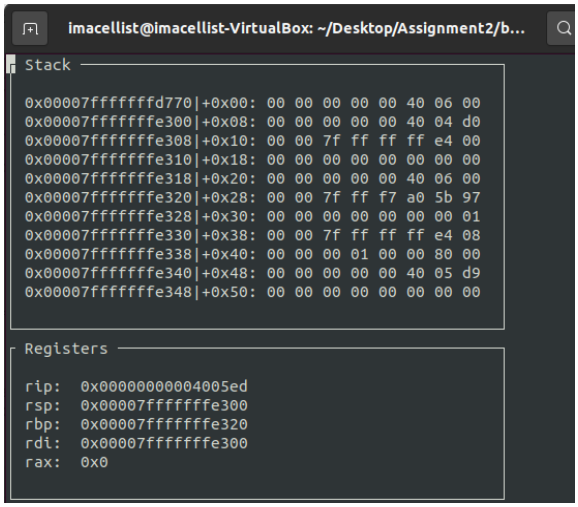
We can also see the actual blacklist with ||cat${IFS}service.py

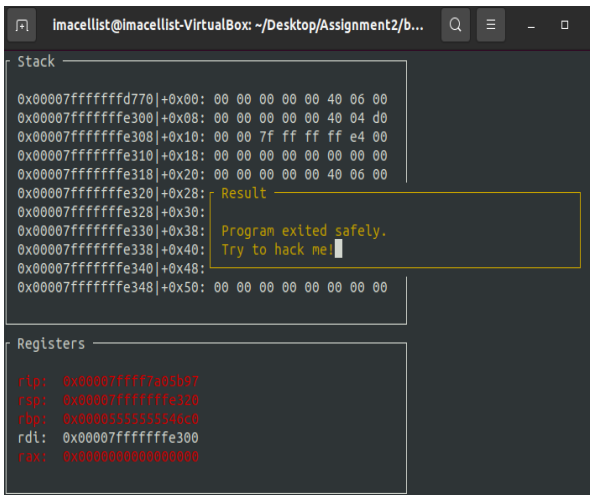


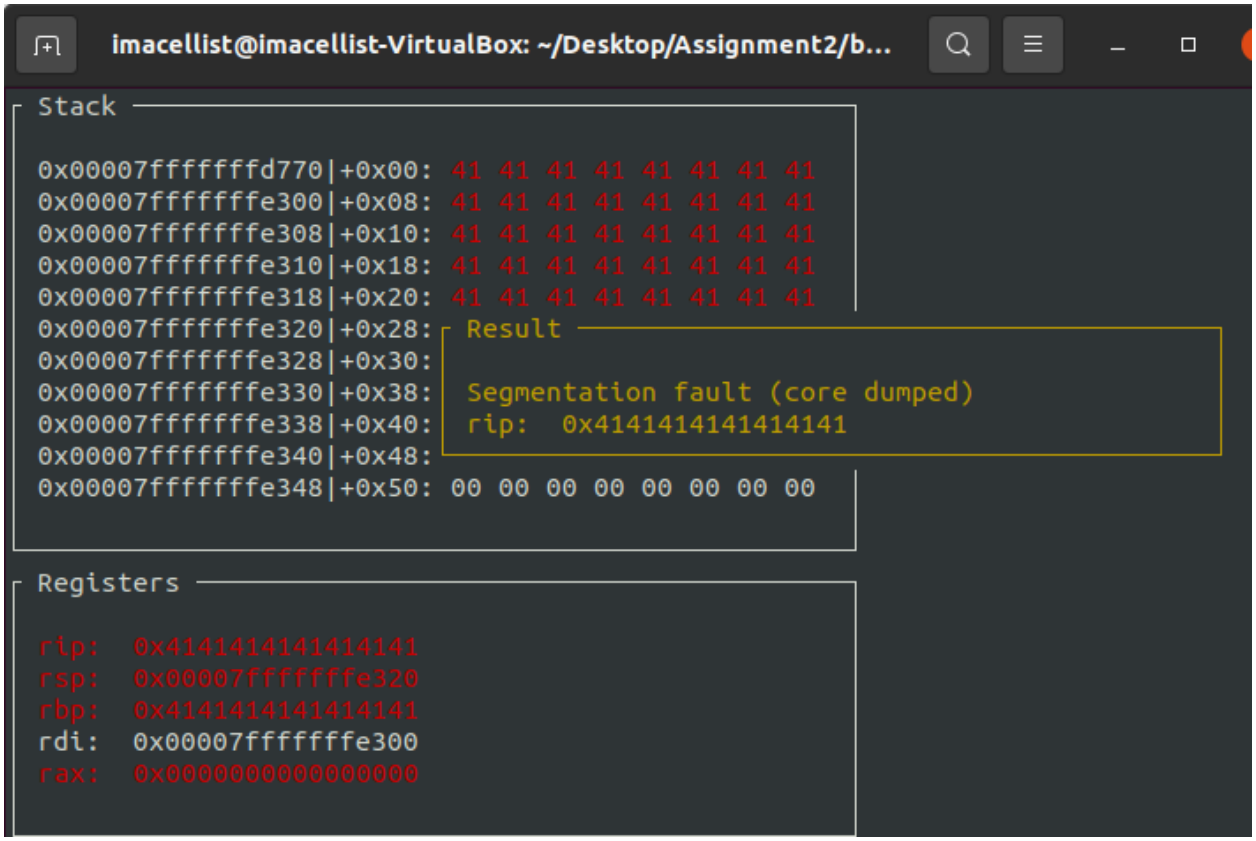
Section D: Binary

D.1 BofSchool (7 Points)

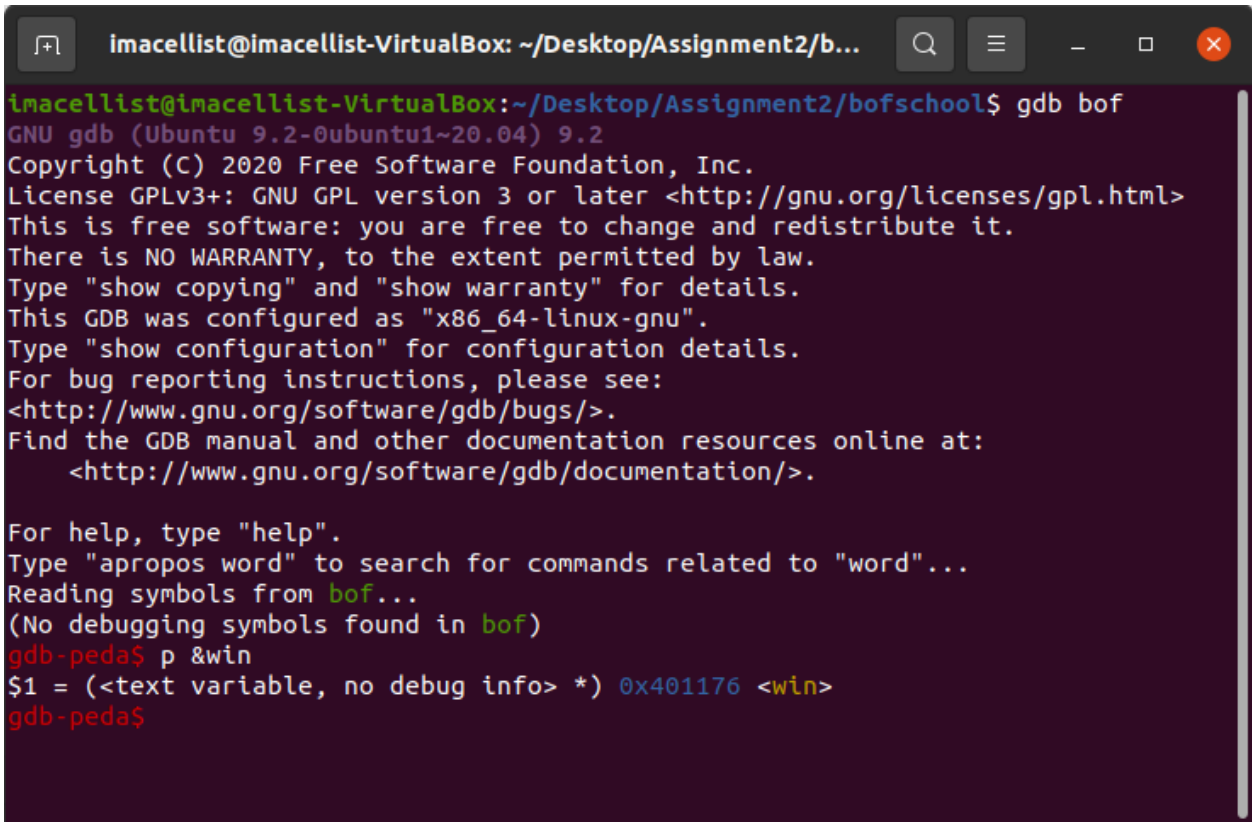
Inspecting the stack and registers using ./bofschool, we find that the return address is stored at 0x00007fffffffe320, hence we will need 40 bytes of padding before we can override the return address.



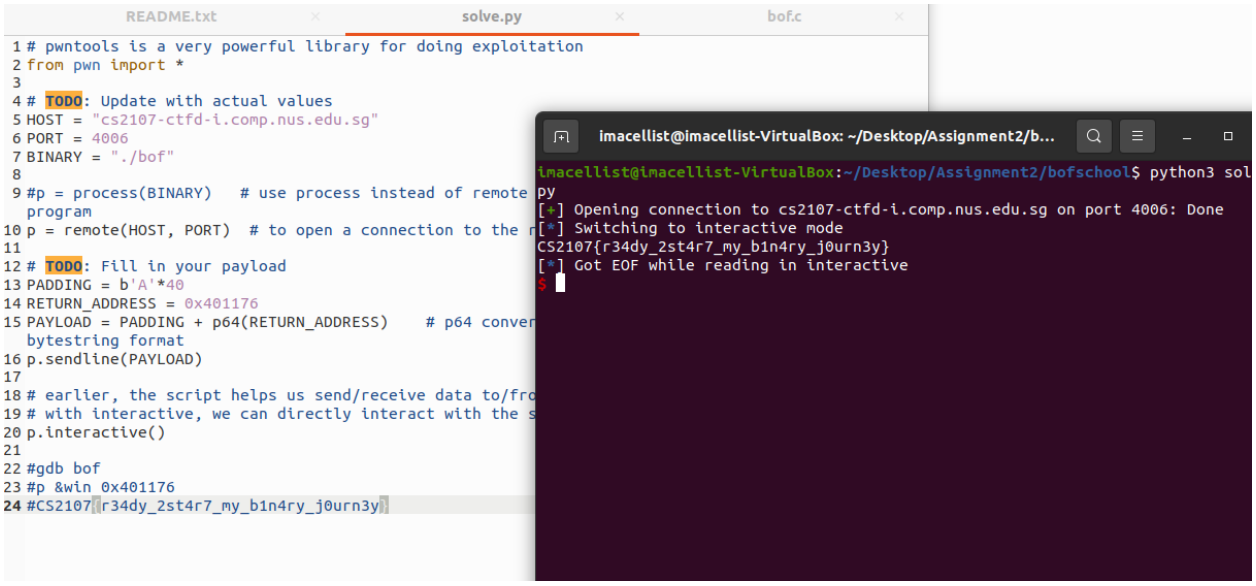




From bof.c, we know we need to overwrite the return address with the address of win(), which can be found using gdb to print the address of win (0x401176)



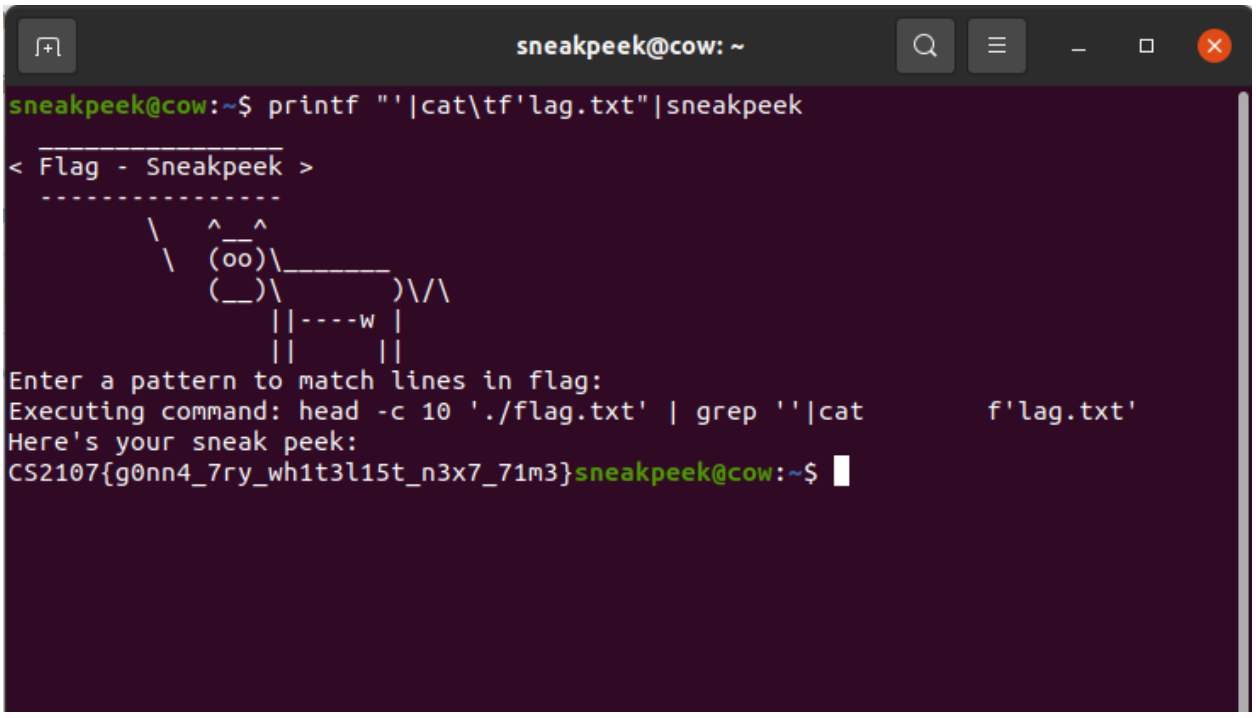
The last step would just be to complete the payload with 40 bytes of padding and the new return address to get the flag CS2107{r34dy_2st4r7_my_b1n4ry_j0urn3y}



D.2 Sneak peek (20 Points)

Similar to both Favourite Tools (C4) and Bad Client Site (C2), we can end the grep command with ‘ and append another cat flag.txt command at the back with | or ;. To break up the filtered keyword flag we can just insert a ‘ to get f’lag which also closes the ending ‘ from the original grep command.

To bypass the whitespace filter, we can insert a substitute for whitespace. However, alternatives like <, {,}, ${IFS} also get blocked by the filter. Luckily, we can use \t in the shell as a tab character and pipe the string into sneakpeek to get the flag CS2107{g0nn4_7ry_wh1t3l15t_n3x7_71m3}.



D.3 Address Book (20 Points)

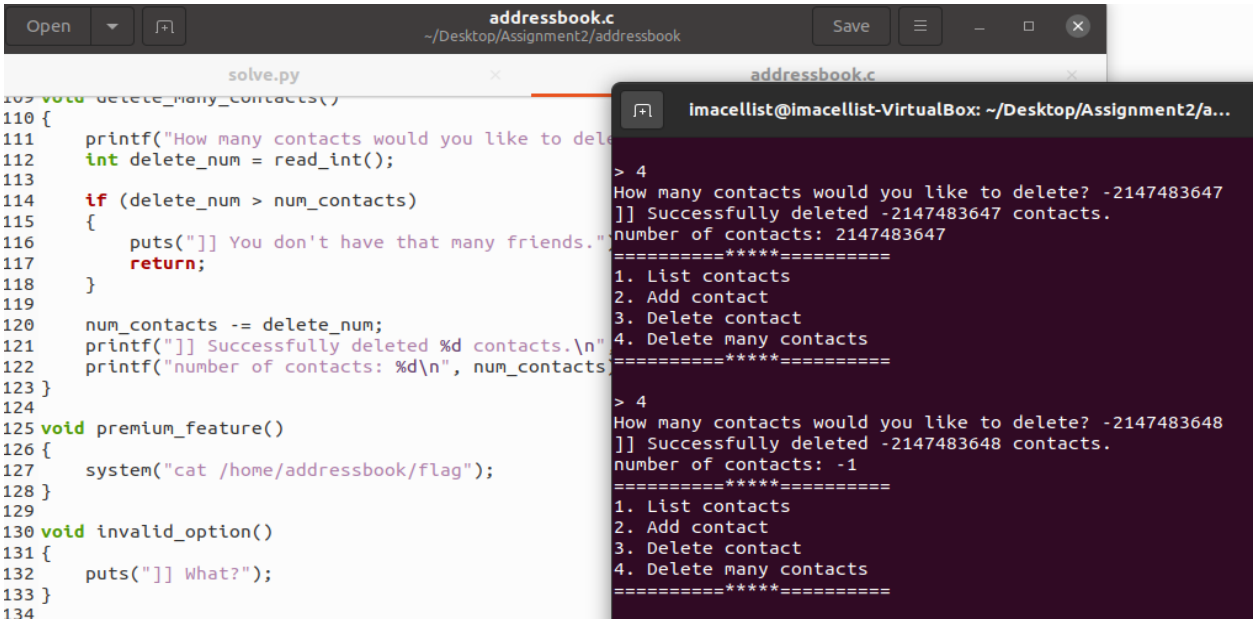
From addressbook.c, we see that we can only get the flag when we input 999 while the variable is_premium_user is not 0, so we need to find a way to overwrite is_premium_user.

```
Reading symbols from addressbook...
gdb-peda$ p &is_premium_user
$1 = (int *) 0x4040ac <is_premium_user>
gdb-peda$ p &contacts
$2 = (struct Contact (*)[20]) 0x4040c0 <contacts>
gdb-peda$ p &contacts[-1]
$3 = (struct Contact *) 0x404098
gdb-peda$
```

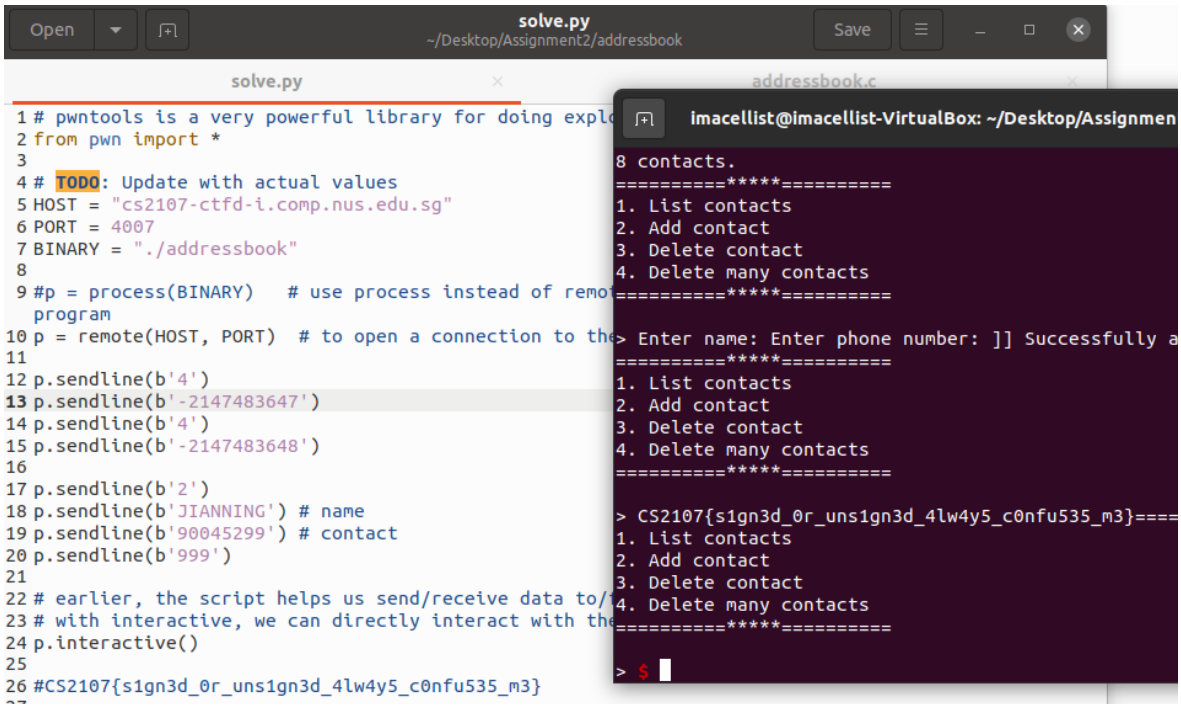
Using gdb again to find the address of the variables, we realise is_premium_user is at 0x4040ac and the contacts array is at 0x4040c0. Exploiting the fact that negative array indices are valid in c, we see that is_premium_user is within the range of contacts[-1].

We need to find a way to decrement num_contacts to -1, however, delete_contact() only allows us to decrement num_contacts if num_contacts > 0. Fortunately, delete_many_contacts() does not have such a check. We can even input negative numbers to bypass the delete_num > num_contacts check. Since the range of twos complement is from -2^31 to 2^31 - 1, we can first input -2147483647 to get num_contacts to be 2147483647, then input -2147483648 (-2^31) to overflow and get -1 in twos complement (FFFFFFFF).

We can check this by modifying addressbook.c to printf num_contacts everytime we delete. We can even print the value of is_premium_user to check what value was overwritten.



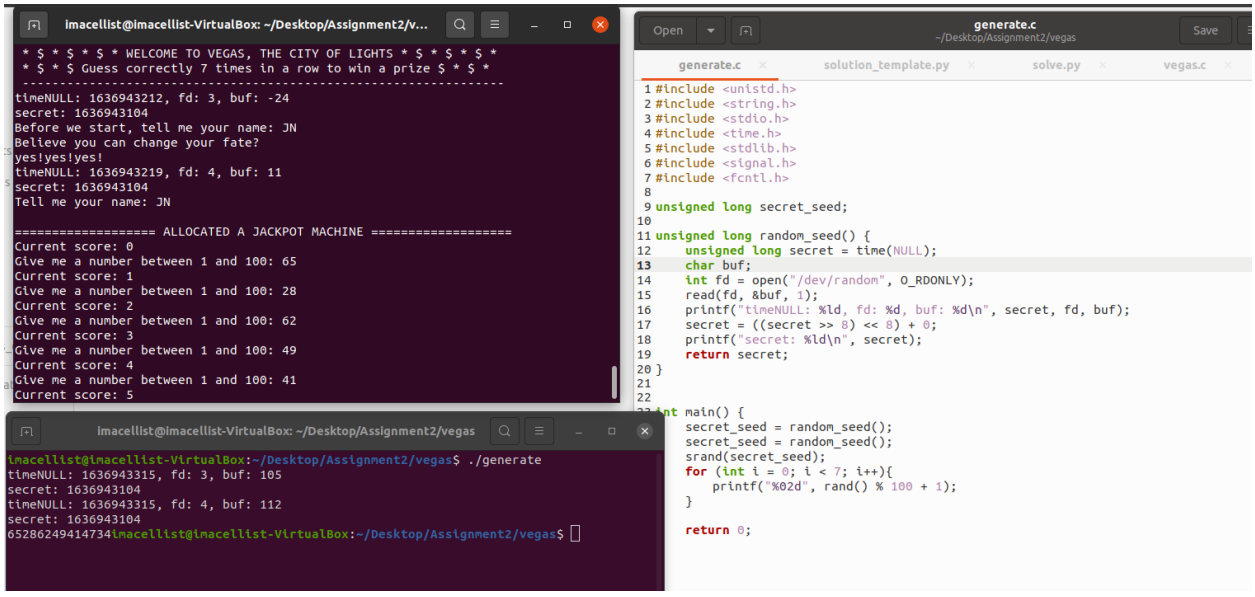
The current address of contacts[-1] will be at 0x404098, which is 20 bytes away from is_premium_user. Since the contact structure is 40 bytes with 20 bytes for name and 20 bytes for phone number, input to phone_number will determine is_premium_user. Any non-zero value will do and we get the flag CS2107{s1gn3d_0r_uns1gn3d_4lw4y5_c0nfu535_m3}.



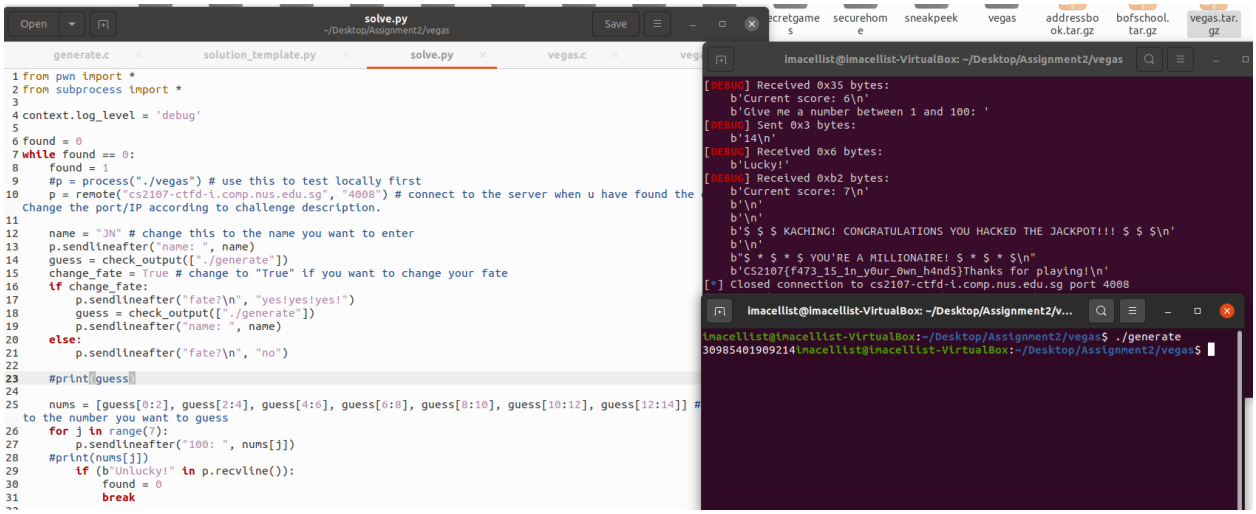
D.4 Vegas (25 Points)

Taking a closer look at how the random seed is generated in vegas.c, we realise that the secret depends on time(NULL) and reading a random character from /dev/random. time(NULL) can be easily exploited by calling another program right after we decide to change our fate. The random character can also be easily brute-forced since it is only 1 character.

Once we generate the same secret, the same seed will give the same number generator sequence when we use srand(secret_seed). The subsequent 7 jackpot numbers will then be deterministic. To test this out, we can create a generate.c program which uses the same random_seed() function as vegas.c, except now, we can print out all the random variables by adding printf statements. Setting buf = 0 for now, we see that vegas.c and generate.c indeed generate the same sequence when called at around the same time.



To incorporate this into our python function, we call ./generate right after we change our fate using yes!yes!yes! We then save the output in an array to check against the jackpot lucky numbers. If we get ‘Unlucky’, we break out of the current iteration and try again.



We get the flag to be CS2107{f473_15_1n_y0ur_0wn_h4nd5}.

Section F: Bonus

F.1 PWNing Address Book 5 Points)

F.2 Vegas 4fun (5 Points)