1. 32 bit Return to Lib-c

Exercise 1: Understanding Return to Lib-c

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.

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.

Definitions

- **Code Reuse Attack**: A Code Reuse Attack (CRA) is a type of cyber attack where the attacker exploits existing, legitimate code in a program to perform malicious actions, often bypassing security mechanisms. Common techniques include return-to-libc and Return-Oriented Programming (ROP).
- **NX/XD bit**: The NX (No eXecute) bit, also known as the XD (eXecute Disable) bit, is a hardware feature used in CPUs to mark certain areas of memory as non-executable. This prevents the execution of code from those regions, enhancing protection against buffer overflow attacks by stopping the execution of injected malicious code.

Exercise Steps

What we have to look out for:

1. EIP is overwritten with the address of the `system()` function residing in the libc library.
2. Immediately after the address of `system()`, there is the address of the function `exit()`, so that once `system()` returns, the vulnerable program jumps to `exit()`, allowing the program to gracefully exit.
3. After the address of `exit()`, there is the address of a memory location containing the string `/bin/sh`, which is the argument that needs to be passed to `system()`.

```
int system(const char *command);
```

Step 1: Writing a Vulnerable Program

We will be using the same program as the 64-bit stack overflow for simplicity:

```
#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;
}
```

Save this program as `prog1.c`. This program reads an input string from the command line and copies it into a buffer without bounds checking, making it vulnerable to a buffer overflow attack.

Compile the program using:

```
# First disable ASLR
cat /proc/sys/kernel/randomize_va_space # check to see if i
t's 0

# if not:
echo 0 > /proc/sys/kernel/randomize_va_space

cc prog1.c -mpreferred-stack-boundary=2 -o prog1
# The boundary has been set to 16 bytes.
```

Step 2: Running the Program in GDB

Let's inspect the program and see what's happening.

```
gdb prog1
# Launches GDB with the compiled program 'lib'.
b main
# Sets a breakpoint at the beginning of the 'main' function.
r
# Runs the program until it hits the breakpoint.
```

Now that the program has run, let's see the security mechanisms in place. You should see NX ticked:

```
checksec
# Displays the security mechanisms (like NX) that are enabled
for the program.
```

Let's see where `system()` is located:

```
p system
# Prints the memory address of the 'system' function in libc.
```

Find the `exit()` function next:

```
p exit
# Prints the memory address of the 'exit' function in libc.
```

Here are examples of expected outputs for these commands:

- `checksec` might show:

  ```
  Canary                        : ✘
  NX                            : ✓
  PIE                           : ✓
  Fortify                       : ✘
  RelRO                         : Partial
  ```

- `p system` might show:

  ```
  $1 = {<text variable, no debug info>} 0x7ffff7e21390 <syst
  em>
  ```

- `p exit` might show:

  ```
  $2 = {<text variable, no debug info>} 0x7ffff7e1d160 <exit
  >
  ```

These addresses will be important for the next exercises, where we find `/bin/sh` and exploit it.

Next Steps

In the upcoming exercises, you will:

1. Find the address of the string `/bin/sh`.
2. Craft the payload to overwrite the return address and redirect execution to `system()`.
3. Use the addresses of `system()`, `exit()`, and `/bin/sh` to achieve a successful return-to-libc attack.

2. 32 bit - Return to Libc

3. 32 bit - Return to Libc

Solutions 32 bit Return-To-Libc