3. 64 bit stack overflow

Exercise 3: Mitigating Stack Smashing Attacks on 64-bit Linux

In this exercise, you will learn and implement various techniques to mitigate stack smashing attacks on a 64-bit Linux system. By the end of this exercise, you should be able to secure a vulnerable program against stack-based buffer overflow attacks using different mitigation strategies.

Objectives

1. Understand the vulnerabilities in stack-based buffer overflow.
2. Implement stack canaries to detect buffer overflows.
3. Enable and test Address Space Layout Randomization (ASLR).
4. Mark the stack as non-executable using NX (No-eXecute) bit.
5. Use compiler options to enhance security.

Prerequisites

1. A 64-bit Linux environment (e.g., Ubuntu).
2. Basic understanding of C programming.
3. Familiarity with Linux command line.

Exercise Steps

Step 1: Analyzing the Vulnerable Program

Start with the following vulnerable C program:

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
```

```
{
    char buf[512];
    strcpy(buf, argv[1]);
    return 0;
}
```

Save this program as `vulnerable.c`.

Step 2: Implementing Stack Canaries

Recompile the program with stack canaries enabled. Stack canaries are special values placed on the stack to detect buffer overflows.

```
gcc -fstack-protector-all -o vulnerable_canary vulnerable.c
```

Test the new binary:

```
./vulnerable_canary $(python2 -c 'print "A"*550')
```

The program should terminate with a stack smashing detected error, indicating that the stack canary was triggered.

Step 3: Enabling ASLR

Address Space Layout Randomization (ASLR) randomizes the memory addresses used by system and application processes, making it harder for an attacker to predict the location of specific parts of the program.

Check the current status of ASLR:

```
cat /proc/sys/kernel/randomize_va_space
```

Enable ASLR by setting the value to 2:

```
sudo sysctl -w kernel.randomize_va_space=2
```

To make this change permanent, add the following line to `/etc/sysctl.conf`:

```
kernel.randomize_va_space = 2
```

Reboot your system to apply the changes.

Step 4: Marking the Stack as Non-Executable (NX)

The NX (No-eXecute) bit marks certain areas of memory as non-executable, preventing code execution from the stack.

Recompile the program with NX enabled:

```
gcc -z noexecstack -o vulnerable_nx vulnerable.c
```

Verify that the stack is marked as non-executable:

```
gdb vulnerable_nx
```

```
gef> checksec
```

It should display `x` for non-executable stack.

Step 5: Using Compiler Options

Enhance the security of your program using additional compiler options:

```
gcc -fstack-protector-strong -D_FORTIFY_SOURCE=2 -Wl,-z,relro,-z,now -o secure_program vulnerable.c
```

These options enable stack protection, buffer overflow detection, and read-only relocation.

Testing Mitigations

1. **Stack Canaries**: Run the program with an overlong input and observe the stack smashing detected message.
2. **ASLR**: Run the program multiple times and note the changing addresses in `/proc/[pid]/maps`.

3. **NX Bit**: Attempt to execute shellcode from the stack and verify that it fails.
4. **Compiler Options**: Test the program to ensure that additional security checks are in place.

Thats it for 64-bit stack overflow. Next is Return-to-libc.