1. 64 bit stack overflow

Exercise 1: Understanding Stack-based Buffer Overflow

In this exercise, you will learn the fundamental concepts of stack-based buffer overflow vulnerabilities. By the end of this exercise, you should understand how buffer overflows occur, how to identify them, and how they can be exploited.

Objectives

1. Understand the basic concepts of stack-based buffer overflow.
2. Analyze a vulnerable C program.
3. Observe the behavior of the program during a buffer overflow.
4. Identify the consequences of a buffer overflow.

Prerequisites

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

Definitions

Register: A register is a small, fast storage location within a CPU that holds data temporarily for quick access during computation. Registers are used to store instructions, addresses, or any kind of data that the CPU needs to access quickly.

Buffer: A buffer is a temporary storage area in memory used to hold data while it is being transferred from one place to another. Buffers help manage data flow between processes or between different components of a computer system, ensuring smooth and efficient data handling.

Function call: A function call is an instruction in a program that transfers control to a function, allowing the function to execute its code. After the function completes its task, control is returned to the point in the program where the function was called.

Exercise Steps

Step 1: Writing a Vulnerable Program

Start with the following simple C program that contains a buffer overflow vulnerability:

```
#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 `vul.c`. (This has been done for you in the folder on the desktop)

Step 2: Compiling the Program

Compile the program with no stack protection mechanisms to observe the effects of the buffer overflow:

```
gcc -no-pie -fno-stack-protector -z execstack -o vul vul.c
```

Step 3: Analysing the Program

Run the program with a normal input to see its expected behaviour:

```
./vul $(python2 -c 'print "A"*519')
```

The program should execute successfully without any errors or outputs.

Step 4: Observing Buffer Overflow

Next, run the program with an excessively long input to cause a buffer overflow:

```
./vul $(python2 -c 'print "A"*521')
```

Observe the program's behaviour. It may crash or behave unexpectedly due to the overflow.

Step 5: Debugging with GDB

Use GDB to analyse the program's behaviour during the buffer overflow. Start GDB and load the program:

```
gdb vul
```

Set a breakpoint at the `vulnerable_function`:

```
(gdb) run $(python -c 'print("A"*600)')
```

When the breakpoint is hit, examine the stack and registers to understand the overflow:

```
(gdb) info frame
(gdb) x/64wx $rsp
```

Look for patterns in the memory (e.g., a series of 'A's (414141...) and note the program's counter.

Step 6: Understanding the Consequences

Discuss the potential consequences of a buffer overflow:

1. **Program Crashes**: The most common immediate effect.
2. **Data Corruption**: Overwriting adjacent memory can corrupt data.

3. **Security Vulnerabilities**: Buffer overflows can be exploited to execute arbitrary code, leading to unauthorized access or control over the system.

Step 7: Mitigation Techniques

Learn about basic mitigation techniques to prevent buffer overflows:

1. **Bounded Functions**: Use functions like `strncpy` instead of `strcpy` to prevent overflows.
2. **Stack Canaries**: Insert special values to detect and prevent buffer overflows.
3. **Compiler Protections**: Use compiler options like `fstack-protector` to enable stack protection mechanisms.

Recompile the program with stack protection:

```
gcc -fstack-protector -o secure_program vul.c
```

Run the secure program with the same long input and observe the difference:

```
./secure_program $(python2 -c 'print "A"*600')
```

The program should detect the buffer overflow and terminate safely.

2. 64 bit stack overflow

3. 64 bit stack overflow

Solutions 64bit Stack Overflow