João Pedro MARÇAL STORINO

ICM 2A – Sécurité, Confiance, Confidentialité

Lab 01

GDB LESSON

1. Overview of GDBLesson Activity:

The **GDBLesson** activity introduces the basic usage of **GDB (GNU Debugger)** to debug C programs, which is essential for understanding the behavior of code at runtime and identifying potential errors. This skill was then applied to debug two C programs: **sampleMath.c** and **sampleMath2.c**, allowing a step-by-step analysis of variable values and correcting logical issues in the code.

2. Results from GDBLesson:

Debugging sampleMath.c

2.1. **Initial Setup**:

After inspecting the file, the program was compiled using the following command:

$gcc -g sampleMath.c -o sampleMath

$./sampleMath

which revealed a logical error causing incorrect division operations.

2.2. **GDB Usage**:

To debug the program, **GDB** was launched with:

$gdb sampleMath

Using commands such as list, b 9 (breakpoint), run, step, and print, the error was identified in the while loop condition. The program was designed to stop when num became negative, but this condition was not functioning correctly.

2.3. **Fix**:

The while loop was modified to correctly handle the condition, ensuring it would stop when num became zero:

```
while(num > 0) { /* Modify this line only */
   total = count / num;
}
```

After recompiling, the program executed successfully.

Debugging sampleMath2.c

2.4. Compilation and Initial Test:

The **sampleMath2.c** program was compiled using:

$gcc -g sampleMath2.c -o sampleMath2

When executed, it showed incorrect results for various test inputs, such as:

$./sampleMath2 1

$./sampleMath2 342

2.5 **GDB Debugging**:

The initial value of the variable total was found to be incorrect (32767). After setting a breakpoint and stepping through the code using print total, the issue was identified.

2.6. **Fix**:

To fix the issue, the total variable was initialized to 0 at the start of the function:

total = 0;

After recompiling, the program produced the expected results, such as for input 342, yielding a correct result of **117,477**.

GDB BUFOVERFLOW

1. Overview:

The **Labtainer buffer overflow lab** demonstrates a classic vulnerability in C programs where input data exceeds the bounds of a buffer, leading to overwriting of adjacent memory spaces. This lab aims to teach the exploitation of such vulnerabilities to gain unauthorized root access by manipulating a program's memory and stack.

2. Steps Followed:

Step 1: Disable Address Space Layout Randomization (ASLR)

ASLR randomizes memory addresses in each program run, making exploitation harder. The first step was to disable ASLR to make the memory layout predictable.

$sudo sysctl -w kernel.randomize_va_space=0

This allowed us to have consistent memory addresses, critical for reliable buffer overflow exploitation.

Step 2: Compiling call_shellcode.c

The **call_shellcode.c** file was compiled and tested to ensure that the shellcode would execute correctly, launching a shell as expected.

$gcc -m32 -z execstack -o call_shellcode call_shellcode.c

$./call_shellcode

This step verified that the code worked and would open a shell. The use of -m32 compiles the program for 32-bit systems, and -z execstack marks the stack as executable, a necessary condition for running shellcode.

Step 3: Compiling and Testing the Vulnerable Program

The vulnerable program (**stack.c**) was compiled with the necessary flags to make it exploitable:

$gcc -m32 -z execstack -fno-stack-protector -o stack stack.c

The -fno-stack-protector flag disables the stack protection, ensuring no canary values are placed between buffers to detect overflow attempts.

Step 4: Editing exploit.c

To exploit the buffer overflow, the **exploit.c** script was modified to inject shellcode into the buffer. By calculating the offset, we ensured that the return address was overwritten with the address of the shellcode.

long address = 0xffffd404;

long *ptr = (long *) (buffer + 182);

*ptr = address;

This snippet ensures that the return address points to the shellcode's location in memory.

Step 5: Running the Exploit

After editing and compiling the **exploit.c** script, I attempted to execute the exploit to gain root access by overwriting the return address with the location of the shellcode. The following steps were performed:

$./exploit

$./stack

However, instead of successfully executing the shellcode and gaining root access, the program crashed, resulting in a **"Segmentation fault (core dumped)"** error. This error indicates that the exploit was unable to successfully overwrite the return

address with the correct shellcode location, leading to an invalid memory access and causing the program to crash.

5. Conclusion:

Despite following the correct steps to execute a buffer overflow attack, the exploit was not successful, and the program consistently crashed with a segmentation fault. This outcome highlights the precision needed when performing buffer overflow attacks, as even slight variations in memory layout can lead to failure. It also underscores the importance of using **modern security measures** such as **stack protection (canaries)**, **ASLR (Address Space Layout Randomization)**, and **non-executable stacks**, which can make such exploits more difficult or prevent them entirely.