Format String: 64 Bit Application

1 Overview

The `formatstring` lab introduced you to `printf` vulnerabilities and potential exploits of those vulnerabilities. That lab included a vulnerable program that ran as a 32-bit x86 application. This lab includes that same source code with one change, however it compiles and runs as a 64-bit application.

1.1 Background

The student is expected to have an understanding of the Linux command line, and some amount of low level programming. It is expected that the student will have completed the `formatstring` lab.

2 Lab Environment

This lab runs in the Labtainer framework, available at http://nps.edu/web/c3o/labtainers. That site includes links to a pre-built virtual machine that has Labtainers installed, however Labtainers can be run on any Linux host that supports Docker containers.

From your labtainer-student directory start the lab using:

```
labtainer format64
```

A link to this lab manual will be displayed.

The home directory of the resulting computer contains the source code of the vulnerable program (`vul_prog.c`) and an executable instance of the program.

3 Tasks

A learning objective of this lab is to appreciate some of the differences between 32-bit and 64-bit x86 applications, and how those differences might affect printf vulnerabilities and exploits. Program descriptions and background material on `printf` behavior are not repeated here. Refer to the `formatstring` lab manual to refresh your memory.

As with the first task of the `formatstring` lab, address space layout randomization ASLR) will be enabled in this lab:

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

3.1 Explore

Review the `vul_prog.c` source code and note its single difference from the version found in the `formatstring` lab. Based on your experience with the `formatstring` lab, explain why this source code change was made.

Use the `file` command to display properties of the `vul_prog` executable. Run the `vul_prog` and observe how its interface looks the same as the version from the `formatstring` lab. Execute the program within gdb and explore the stack structures at different points in the program execution. Use the gdb disassemble directive to view the assembly language instructions.

3.2 Task 1: Exploit the vulnerability

The program has the two secret values stored in its memory as were found in the `formatstring` lab. You will perform a subset of the tasks from the `formatstring` lab, specifically:

- Print out the secret[1] value.
- Modify the secret[1] value to equal 0xa.

For this lab task, you are not to modify the code. Namely, you need to achieve the above objectives without modifying the vulnerable code. The order and sequence in which you achieve the objectives does not matter. Feel free to explore and experiment as long as you succeed in each at least once.

3.3 Task 2: Memory randomization

In the `formatstring` lab, you modified the source code to eliminate setting the `input_int` variable from user input. You also disabled ASLR to simplify the process of exploiting the program. Your exploit technique then embedded the secret's address within the input string. That technique will not work in this 64-bit environment. Why is that? What is the broader implication for 64-bit programs?

4 Submission

After finishing the lab, go to the terminal on your Linux system that was used to start the lab and type:

```
stoplab
```

When you stop the lab, the system will display a path to the zipped lab results on your Linux system. Provide that file to your instructor, e.g., via the Sakai site.

This lab was developed for the Labtainer framework by the Naval Postgraduate School, Center for Cybersecurity and Cyber Operations under sponsorship from the DoD CySP program.