Buffer Overflow: 64 Bit Application

1 Overview

The `bufoverflow` lab introduced you to buffer overflow vulnerabilities and potential exploits of those vulnerabilities. That lab included a vulnerable program that ran as a 32-bit x86 application. This lab includes the very same vulnerable program source code, 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 `bufoverflow` 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 buf64
```

A link to this lab manual will be displayed.

The home directory of the resulting computer contains the source code of the vulnerable program (`stack.c`) and a template for the program that constructs the malicious data file (`exploit.c`). It also includes files for generating shell code.

3 Tasks

You will modify the `exploit.c` program such that it generates a malicious data file that will cause the vulnerable program to enter a shell. This lab does not require that you get a root shell – an application shell is sufficient. And your exploit need only run with Address Space Layout Randomization disabled, with an executable stack, and with stack protection disabled.

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 vulnerabilities and exploits.

3.1 Explore

Review the differences between the files in this lab, and the files in the `bufoverflow` lab. Note the `stack.c` files are the same – but with potentially different buffer sizes. Look at the assembly in `shell.c` and compare that to the assembly comments to the object code found in the `bufoverflow exploit.c` file.

Disable ASLR:

```
sudo sysctl -w kernel.randomize_va_space=0
```

and use the `compile.sh` script to compile the C programs and assemble the shell.asm Run the `stack` program. Run it in the debugger. Explore.

3.2 Shell code

In this lab, you will need to update the `exploit.c` program to include the shell code. Observe that the 64-bit shell code has been assembled into the shell.bin file. You must figure out how to get that into your `exploit.c` program. Note you have been provided with a Python script called `hexit.py`, and that may be of use. Look at the file from the `bufoverflow` lab as an example.

3.3 Overwriting return address

By now you should have observed that the `stack` program crashes when it encounters a bad return address. As in the `bufoverflow` lab, you will want to control that return address value.

What might you want that return address value to be? Consider the properties of such a value and how that might affect the `strcpy` function behavior.

3.4 Get a shell

Alter the `exploit.c` so that it generates a badfile that causes the `stack` application to give you a shell. Once you get a shell, cat the exploit.c file from within the shell:

```
cat exploit.c
```

3.5 Follow on

Think about how you might approach an exploit if the `stack.c` program were compiled without disabling stack protection and stack execution.

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.