Project Buffer Overflow

Due: October 6, 2015

1 Introduction

The Pawnee Library Department is filled with mean, conniving, rude and extremely well read people who try their best to take advantage of the good people of Pawnee. It is time for them to have a taste of their own medicine so the Parks Department has come up with a plan to teach them a lesson after learning a bit of x86-64 assembly and architecture.

2 Assignment

This assignment will help you develop a detailed understanding of x86-64 calling conventions and stack organization. It involves applying a series of *buffer overflow attacks* on an executable file called `buffer`.

In this project, you will gain firsthand experience with one of the methods commonly used to exploit security weaknesses in operating systems and network servers. Our purpose is to help you learn about the runtime operation of programs and to understand the nature of this form of security weakness so that you can avoid it when you write system code. We do not condone the use of this or any other form of attack to gain unauthorized access to any system resources.

This assignment contains the following files:

- *buffer*: The buffer program you will attack.
- *makecookie*: Generates a "cookie" based on your userid.
- *hex2raw*: A utility to help convert between string formats.
- *buffer.pdf*: This document.

Your task during this assignment is to use buffer overflow attacks to cause the `buffer` program to behave in unexpected ways.

3 Userids and Cookies

Phases of this project will require a slightly different solution from each student. The correct solution will be based on your userid.

A *cookie* or *hash* is a string of eight hexadecimal digits generated from your userid in such a way that distinct userids will (with high probability) produce distinct cookies. You can generate your cookie with the `makecookie` program giving your userid as the argument. For example:

```
./makecookie jcarberr
0x44e22c05
```

In two of your three buffer attacks, your objective will be to change values to detriment the library department. In three of those four attacks, you will accomplish this by supplying machine code instructions to the `buffer` program.

A problem with doing so is that Linux does not allow data on the program stack to be executed as machine instructions. However, your spy inside the Library Department anticipated this safeguard, and tricked the programmers that wrote libraries security into moving the stack to a different, executable memory location. This means that the instructions that you place on the stack can indeed be executed.

4 The `buffer` Program

The `buffer` program reads a string from standard input. It does so with the function `getbuf` defined below:

```
/* Buffer size for getbuf */
#define NORMAL_BUFFER_SIZE 32

int getbuf() {
     char buf[NORMAL_BUFFER_SIZE];
     Gets(buf);
     return 1;
}
```

The function `Gets()` is similar to the standard library function `gets()`—it reads a string from standard input (terminated by '`\n`' or end-of-file) and stores it (along with a null terminator) at the specified destination. In this code, you can see that the destination is an array `buf` having sufficient space for 32 characters.

`Gets()` (and `gets()`) grabs a string off the input stream and stores it into its destination address (in this case `buf`). However, `Gets()` has no way of determining whether `buf` is large enough to store the whole input. It simply copies the entire input string, possibly overrunning the bounds of the storage allocated at the destination.

If the string typed by the user to `getbuf()` is no more than 31 characters long, `getbuf()` will correctly return 1, as shown by the following execution example:

```
./buffer -u jcarberr
Type string: I love CS 33.
Oops: getbuf returned 0x1
```

Typically an error occurs if a longer string is entered:

```
./buffer -u jcarberr
Type string: It is easier to love this class when you are a TA.
Ouch!: You caused a segmentation fault!
```