

Assignment 1

Due: Feb. 5th, 2015, 11:59PM EST

1 Working in a Group

You should form groups of two people (or, with emailed permission from a TA, a group of three) for this assignment. Group registration should be done by Jan. 20th. Directions for completing group registration are on Piazza, and you can use Piazza to find a partner. All members of a group will share the same score.

2 Assignment overview

There are three problems in this assignment. In each of them, you will be required to exploit the server to add a file containing information specified. In Problem #1 and Problem #2, the servers are running with stacks stored at fixed addresses. In Problem #3, a modified version of the server from Problem #1 is running with stacks located at random addresses.

2.1 Problem #1 (25 pts)

2.1.1 Introduction

The server in `server1` is part of a “geotagging” application. Its source code is provided in `server1.c`. Using this server, you can upload latitude and longitude information, along with a caption, about a given picture file (the image itself is uploaded separately).

Requests are formatted as follows:

```
IMG:<filename>;LAT:<latitude>;LON:<longitude>;CAP:<caption>;
```

Here are two examples:

```
IMG:walking.jpg;LAT:40.44417;LON:-79.94461;CAP:Due to apparent structural instability, the Walking to the Sky sculpture was replaced in 2009 by a sturdier version.;
```

```
IMG:burger.png;LAT:40.70330;LON:-74.011029;CAP:The $175 Richard Nouveau burger from Wall Street Burger Shoppe in New York is topped with foie gras, black truffles, and flakes of gold leaf.;
```

The server listens on port `18732` by default and keeps each connection alive for 5 seconds. An example client program `client1.c` is provided for your reference. You can modify `client1.c` to exploit a vulnerability in the server.

2.1.2 Requirement

Write a C client program that interacts with `server1` and exploits a buffer overflow to add a file called `foo.txt` to the remote system. The contents of the file should be **“You lose!”**. Suppose

`client1` is the binary compiled from your code. It should take *server* and *port* as parameters. More specifically, we will run your code as follows:

```
./client1 -port 18732 -server 127.0.0.1
```

After this command, the server should generate a file “foo.txt” with contents “**You lose!**” (quotation marks are not included).

2.2 Problem #2 (35 pts)

2.2.1 Introduction

The server in `server2` is a “hacker translation service.” Its source code is provided in `server2.c`. Using this server, you can transform text from English to l33tspeak automatically.

Requests are formatted as follows:

```
<type>|<text>
```

The type field selects the kind of text translation to do. Here are two examples:

```
1|Your mother was a hamster and your father smelt of elderberries!
```

```
2|My name is Inigo Montoya. You killed my father. Prepare to die!
```

The server listens on port *18732* by default and keeps each connection alive for 5 seconds. An example client program `client2.c` is provided for your reference. You can modify `client2.c` to exploit a vulnerability in the server.

Normal operation of the client:

```
unix45:54> ./client -port 18732 -server 127.0.0.1
```

The request is:

```
1|The cure for boredom is curiosity. There is no cure for curiosity. -Dorothy Parker
```

The response of the server is:

```
Th3 cur3 for bor3dom is curiosity. Th3r3 is no cur3 for curiosity. -Dorothy Park3r
```

2.2.2 Requirement

Write a C client program that interacts with `server2` and exploits an overflow to add a file called `bar.txt` to the remote system. The contents of the file should be “**I win!**”. Suppose `client2` is the binary compiled from your code. It should take *server* and *port* as parameters. More specifically, we will run your code as follows:

```
./client2 -port 18732 -server 127.0.0.1
```

After this command, the server should generate a file “bar.txt” with contents “**I win!**” (without quotation marks).

2.3 Problem #3 (40 pts)

2.3.1 Introduction

The server `server3` is a modification of `server1`. Differences between `server1` and `server3` include:

1. *Randomized stacks* While `server1`'s stack always starts at the same address, the address of `server3` varies. This is a good opportunity to learn about ASLR and how to exploit an ASLR-protected program.
2. *Input validation* The way the input is validated is slightly changed. You may want to spend time investigating the difference.
3. *No source code* This is a good chance for you to learn how to use `gdb` to analyze the binary without source code!

Note: We are not directly using ASLR support from the OS. Instead, we implement a similar stack randomization using `mmap`. You are welcome to analyze how we do it. However, the techniques that could be used to exploit program with ASLR will work here.

2.3.2 Requirement

Write a C client program `client3.c` that interacts with `server3` and exploits an overflow to add a file called `foobar.txt` to the remote system. The contents of the file should be "I win again!". You should **NOT** try to bruteforce the server. You should come up with an exploit that will **always** generate the file `foobar.txt`. Suppose `client3` is the binary compiled from your code, it should take `server` and `port` as parameters. More specifically, your code is expected to run as follows:

```
./client3 -port 18732 -server 127.0.0.1
```

After this command, the server should generate a file "foobar.txt" with contents "**I win again!**" (quotation marks are not included).

2.3.3 Partial Credit

For Problem #3 only, we will offer partial credit if your solution does not completely work. If your exploit works every time, you just need to turn in `client3.c`. If, however, your solution only works part of the time, or if you manage to exploit some vulnerabilities, yet not enough to complete the specified problem, you should turn in both `client3.c` and a text file called `explanation.txt` that briefly summarizes the progress you made, why what you did works, and where you are stuck. This explanation may be up to 2 paragraphs long.

3 Submission

You should turn in a tarball "handin.tar" to Autolab containing either exactly three files (`client1.c`, `client2.c`, `client3.c`) or exactly four files (`client1.c`, `client2.c`, `client3.c`, `explanation.txt`). As the Autolab image we are using has a 64-bit linux kernel, your *.c files should be built with gcc in a 64-bit Linux kernel. Before you submit your files to Autolab, we suggest you check your C files compile on Andrew Linux machines (`linux.andrew.cmu.edu`). For example, Autolab will try to compile `client1` from the file "client1.c" with the following command:

```
gcc -g -o client1 client1.c
```

4 Grading

We will use **Autolab** to automatically grade your submissions. You can submit multiple times to **Autolab** for grading. However, we strongly recommend that you first *debug your code using the binary of the servers we provide and **gdb***. As **Autolab** is not designed for your testing and debugging, the results are not immediately available after you submit your tarball. According to our tests, it may take time from several seconds to several minutes before the grading results appear.

In order to enjoy the benefit from **Autolab**, here are some rules you should obey:

1. The C files you submit should have the **exact name** as specified.
2. The C files you submit should compile on Andrew, as specified above.
3. The files generated by the servers because of your exploit should have exactly the **same name** as required.
4. The files generated by the servers because of your exploit should have exactly the **same content** as required.
5. You should **NOT** try to bring down the environment of **Autolab**.

5 Attack Plan

For those who are not very familiar with buffer overflows, here is a suggested attack plan:

1. Get familiar with x86 assembly and calling conventions.
2. Get familiar with buffer overflow attacks.
3. Experiment with the server binary and the provided clientX.c, try to use debug tools (gdb) to analyze the binary.
4. Identify the places of the vulnerability.
5. Search for a usable shellcode.
6. Rewrite clientX.c to exploit the server.
7. Debug using gdb to understand why your exploit work or why it does not.
8. Double check that your clients strictly follow the requirement and grading rules.
9. Submit to Autolab.
10. Congratulations!

6 Resources

Some resources that may be useful:

1. A gdb tutorial: <http://www.cs.cmu.edu/~gilpin/tutorial/>
2. A tutorial about how to generate useful shellcode: <http://ohhara.sarang.net/security/adv.txt>
3. Shellcode arsenal: <http://www.exploit-db.com/shellcode/>
4. ASLR Smack & Laugh Reference(Just google it! A lot of useful techniques to exploit ASLR program)
5. Readings on the course website!