Secure Software Systems *Lujo Bauer*

Assignment 1

Due: Feb. 10th, 2014, 11:59PM EST

1 Working in Group

You should form in groups of two or three people for this assignment. Group registration should be done before Jan. 29th. Directions for completing group registration will be placed on Piazza. You could use Piazza to find a partner.

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 required information. In Prob #1 and Prob #2, the servers are running with stacks at fixed addresses. In Prob #3, a modified version of server from Prob #1 is running with stacks at random addresses.

2.1 Prob #1 (30 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 content 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, you code is expected to run as follows:

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

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

2.2 Prob #2 (40 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.

```
unix45:52> cat foo.txt
Do you feel lucky, punk?
unix45:53>
```

Then your client should produce the following output:

```
unix45:54> ./client -port 18732 -server 127.0.0.1
Do you feel lucky, punk?
unix45:55>
```

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, your code is expected to run as follows:

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

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

2.3 Prob #3 (30 pts)

2.3.1 Introduction

The server `server3` is a modified version of `server1`. The differences between `server1` and `server3` include:

1. *Randomized stacks* While `server1`'s stack always starts at the same address, the address of `server2` actually 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 still 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, you 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 content **"I win again!"** (quotation marks are not included).

3 Submission

You should turn in a tarball "handin.tar" to Autolab containing exactly three files (client1.c, client2.c, client3.c). As the `Autolab` image we are using has a 64-bit linux kernel, your *.c files should be built with gcc in 64-bit linux kernel. Before you submit your files to Autolab, we suggest that you should 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 recommand 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 show up.

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

1. The C files you submit should have the exactly the **same name** as required.
2. The C files you submit should compile.
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. Your should **NOT** try to bring down the environment of `Autolab`.

5 Attack Plan

For those who are not very familiar with buffer overflow, here is a brief attack plan you could follow:

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!