Lab 1: Buffer Overflow Attacks

Lab Overview

The goal of this lab is to understand buffer overflow vulnerabilities. To this end, you need to complete two tasks and explore how to enable buffer overflow attacks using two different techniques. The first task is to inject shellcode into a vulnerable problem and trick the victim to launch the malicious code on your behalf. The second task is to use return-to-libc technique to further bypass the data execution prevention (DEP) mechanism and exploit the buffer overflow vulnerability without introducing additional code to stack.

Requirement

You are required to submit a written lab report. In this report, you need to use screenshots to explain step-by-step what you have done, what you have achieved and what you have observed.

Lab Environment

This lab needs to be conducted in a prebuilt virtual machine image, where the lab environment has been established. You can download this image from here: lab 1 VM. This is a 32-bit Ubuntu image. To run it, you need to create a new VM in VirtualBox using the existing virtual hard disk file, SEEDUbuntu-16.04-32bit.vmdk, uncompressed from the given .zip file. Once you can start this VM, you can log in the system with username *seed* and password *dees*.

Preparation

First, to enable our attacks, we need to disable the existing defense techniques including DEP, ASLR and stack canary. To this end, you need to compile the code with specific options and change OS configuration:

1) To disable DEP, you need to compile the vulnerable code using the -z execstack option. Notice that this is only needed in Task 1.

   ```
   gcc -o vulnerable -z execstack vulnerable.c
   ```

2) To disable stack canary, you need to compile the vulnerable code with the -fno-stack-protector option:

   ```
   gcc -o vulnerable -fno-stack-protector vulnerable.c
   ```

3) To disable ASLR, you need to use the following command:

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

Second, you need to set the vulnerable code to be a `Set-UID` program. A `Set-UID` program can enable normal users to conduct critical operations that require root privilege. It can escalate its privilege by setting its own effective user ID (euid) to be 0 (root) during execution and drop the privilege upon completion. As a result, if we can exploit a buffer overflow vulnerability in a `Set-UID` program in the middle of its execution, we can run arbitrary code with root privilege. To set an executable to be a `Set-UID` program, you can use the following command:

```
sudo chown root vulnerable
sudo chmod 4755 vulnerable
```

Third, you need to configure the `/bin/sh`. In Ubuntu 16.04 VMs, the `/bin/sh` symbolic link points to the `/bin/dash` shell, which has a countermeasure that prevents itself from being executed in a `Set-UID` process. Basically, if dash detects that it is executed in a `Set-UID` process, it immediately changes the effective user ID to the process's real user ID, essentially dropping the privilege. Since our victim program is a `Set-UID` program, and our attack relies on running `/bin/sh`, the countermeasure in `/bin/dash` makes our attack more difficult. Therefore, we will link `/bin/sh` to another shell that does not have such a countermeasure. A shell program called `zsh` is also installed in this Ubuntu 16.04 VM. We use the following commands to link `/bin/sh` to `zsh`

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

Task 1: Shellcode Injection

You can download a vulnerable program and a template of exploit code at vulnerable1.c and exploit1.c. Your task is to complete the exploit code so as to attack the vulnerable one via injecting shellcode. To launch an attack, you need to first run the exploit code that creates an attack file, `badfile`, and then run the vulnerable code to read from this file. If your `badfile` has been correctly generated, the attack will be successfully launched at this point. Since the vulnerable code is a `Set-UID` program which runs at root privilege, you should expect to see a root shell (starting with a "`#`" symbol) after a successful exploitation.

Hint: To craft the attack buffer in the exploit code, you need to figure out 1) the value of fake return address and 2) where to put this fake return address in the buffer.

1) **The value of return address**. You can use `gdb` to debug the vulnerable program and find the address of the vulnerable buffer. Your injected shellcode is actually located somewhere above the buffer. Particularly, you can first set the return address to be any arbitrary value so as to create an attack file to feed the vulnerable program. Then, once you find out a functional fake return address, you can use that to replace the old one. Notice that when you debug a copy of `Set-UID`

program, the discovered address during debugging is slightly deviated from the one in normal execution. However, this difference is small.

2) **The location to insert fake address**. you need to figure out the distance between the vulnerable buffer and the return address stored on the stack. The vulnerable buffer may not be placed at the top of stack frame (i.e., adjacent to saved base pointer) due to different compiler conventions. To find where the buffer is located, you can use `objdump` to dissemble the compiled binary to observe how memory space is allocated for the buffer. For instance, you can use the following command to output the disassembly to a text file vulnerable.dis and look into its content:

```
objdump -d vulnerable > vulnerable.dis
```

Task 2: Return-to-libc Attack

You can download a vulnerable program and a template of exploit code at vulnerable2.c and exploit2.c. This time, your task is to complete the exploit code to launch a return-to-libc attack. Again, to enable this attack, you need to run the exploit code and vulnerable program consecutively. However, instead of injecting shellcode, you need to craft an attack buffer with only addresses and parameters of libc functions. Particularly, we expect you to launch a new shell using `system()` call with the "`/bin/sh`" parameter and then use "`exit()`" to terminate the original victim program.

Hints:

1) **The addresses of libc calls**. Since we have already disabled ASLR, you can debug the vulnerable code to discover the addresses of `system()` and `exit()`. For instance:

```
$gdb vulnerable

gdb-peda$ b main
gdb-peda$ r
gdb-peda$ p system
$1 = {<text variable>, <no debug info>} 0xb7da4da0
<__libc_system>
gdb-peda$ p exit
$2 = {<text variable>, <no debug info>} 0xb7d989d0 <__GI_exit>
```

2) **The address of the parameter string "`/bin/sh`"**. We can put this string in memory and get its address through environment variables. Particularly, we introduce a new shell variable `BINSH` and make it point to `/bin/sh`:

```
export BINSH=/bin/sh
```

To help you retrieve the address of this variable, we have intentionally put a `printf()` statement, `printf("%x\n, getenv("BINSH"))`, at the beginning of the vulnerable program

to display this address in the victim program's memory. Notice that the addresses of the same environmental variable can be different in individual programs depending on the other string symbols being used.

3) **The memory location to place the parameter string**. When a function is called, the system will create a new stack frame for this function and then fetch its parameters from the stack. By convention, the parameters are stored immediately above the return address, as depicted in the following figure.