CS2107 Self-Exploration Activity 11: Buffer Overflow Vulnerability & Exploitation

Notes:

In this Activity 11 about **buffer overflow vulnerability and exploitation**, you will perform the following:

1. To overflow a target buffer in a given sample vulnerable program, and observe a **stack smashing** in action;
2. (*Optional*) To overflow the target buffer with a specially crafted string containing a shellcode, and observe how a **root shell** can be obtained following the shellcode execution.

You can also view the ***two accompanying demo videos*** about buffer overflow and buffer overflow exploitation, which have been uploaded to our module's Multimedia on LumiNUS.

Task 1: Overflowing a Buffer in a Vulnerable Program

Download the C program named `BO_target.c` which has been uploaded to LumiNUS. Inspect this vulnerable program and notice its `vuln_function()`, which insecurely performs a `strcpy()` operation from a **user-inputted string** into a fixed-sized buffer declared within the function.

First, disable the *Address Space Layout Randomization (ASLR)* on your Linux by setting the kernel parameter to 0. This step is needed so that, for learning purposes, you can exploit the vulnerable program more easily in Task 2 later:

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

You can additionally run the following command to make sure that the kernel parameter indeed has been se to 0 (disabled ASLR):

```
$ sudo sysctl kernel.randomize_va_space
```

Now, compile the program to generate two **executable variants** as follows:

```
$ gcc BO_target.c -o BO_target_1
```

```
$ gcc BO_target.c -o BO_target_2 -z execstack
  -fno-stack-protector
```

Notice that the **`BO_target_1`** executable is built by using the **default setting** of the gcc compiler, which embeds the ***canary-based stack protector*** into the generated executable, and also ***disables any code execution*** in the process' call stack. The **`BO_target_2`** executable, meanwhile, has no canary-based stack protection nor stack-execution prevention.

Before running the two generated executables, make them both set-UID-root programs as follows:

```
$ sudo chown root:root BO_target_?
```

```
$ sudo chmod u+s BO_target_?
```

Finally, run the two executables, and do supply **different input strings** with several different lengths. Do ensure that you enter a string **longer than** 30 characters to see how the two corresponding processes handle a buffer overflow incident. You should see how the stack protector in **`BO_target_1`** detects the stack-smashing incident and abort its program execution. **`BO_target_2`**, meanwhile, gets aborted due to the *segmentation fault* exception raised by the OS.

Task 2: (*Optional*) Exploiting the Vulnerable Program to Obtain a Root Shell

From Task 1, you know that the program has a buffer overflow vulnerability. Your supplied long input strings, however, usually will only **crash** the program without performing any exploitations. Now, let us go further to exploit **`BO_target_2`** so that we can obtain a **root shell**.

To exploit the executable, you need to specially craft your **input string**. As mentioned in the lecture, such a string crafting must be done in a precise manner. To facilitate our string crafting, let us use a file named `badinput` to hold your input string. You can manually edit the file, for instance, using `gedit` as follows:

```
$ gedit badinput
```

Subsequently, you can run the target executable using the input file as follows:

```
$ ./BO_target_2 < badinput
```

To help you craft your exploit string, you can download a Python script named `craft_input.py` from LumiNUS. The script writes a prepared **shellcode** into `badinput`. The shellcode first runs `setuid(0)` in order to bypass the privilege-downgrading mechanism of bash as discussed earlier, and then executes a few operations explained in http://shell-storm.org/shellcode/files/shellcode-219.php.

To craft an input string that is workable on your machine, which is assumed to be a 32-bit machine, the script needs you to supply the two following parameters:

1. The **offset** (distance) of the return address in the call stack from (i.e. relative to) the `buffer` array to be overflowed;
2. The **memory address** of the shellcode portion of the input string as stored in the call stack of the target process.

You thus need to perform the following series of steps in order to find out the two parameters on your machine:

- First, compile the source file again with debug information included:

```
$ gcc BO_target.c -o BO_target_gdb -g
  -z execstack -fno-stack-protector
```

- Then, run gdb as follows:

```
$ gdb -q BO_target_gdb
```

- Inside gdb, run the following gdb commands:
  - `b vuln_function`
  - `run`
  - `p &buffer` which gives you an address A_1
  - `p $ebp` which gives you an address A_2
  - `p/d` <A_2> - <A_1> which gives you a number $x = A_2 - A_1$ in decimal

In `craft_input.py`, you then need to replace `$ebp?` with A_2, and replace `x?` with x = <$ebp> - <&buffer>.

Once your `craft_input.py` is ready, make it as an executable file and run it as follows:

```
$ chmod u+x craft_input.py
$ ./craft_input.py
```

Finally, run the target executable using your crafted `badinput`:

```
$ ./BO_target_2 < badinput
```

If everything is correctly performed, you should be able to obtain a **root shell**!

(When and as necessary, you can check the uploaded accompanying demo video on buffer overflow exploitation to see how the steps above can be done.)