Assignment 2: Software Security

Rukhshan Haroon (22100195@lums.edu.pk) is the lead TA for this assignment.

This project is split into **6** tasks. The whole assignment is worth **100** points and is due on **March 14** at **11:55pm**. We strongly recommend that you get started early.

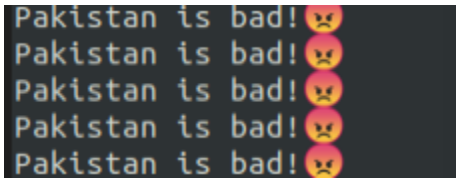
This is a group project; you will work in pairs and your partner will be the same as in Assignment 1.

The code and other answers your group submits MUST be entirely your own work, and you are bound by the Student Code. You MAY consult with other students about the conceptualization of the project and the meaning of the questions, but you MUST NOT look at any part of someone else's solution or collaborate with anyone outside your group. You may consult published references, provided that you appropriately cite them (e.g., with program comments), as you would in an academic paper.

Solutions MUST be submitted electronically on LMS within the due date.

Task 1: Dynamic Analysis [5 points]

In this part, you have been contacted by the National Cyber Security Division of Pakistan, they need your expertise for a matter of utmost attention. Recently, a twitter account has surfaced which is propagating hate against Pakistan by statements like this:



Fortunately, you have gotten access to the program running this account, which is an executable binary file “bot”. You were first suggested to just stop this twitter account, but you have realized that it would be a better idea to use it for putting out good words for Pakistan. However, you can't change the code for the bot since you don’t have access to it. Here we introduce you to a new tool called **Frida**.

Frida is a state of the art scripting utility being used by hackers and academics alike in the world, and is quite loved in the CTF community, due to its power across Android, Iphone and native code.

Frida affects the code in real time, therefore, make sure that the “bot” is running, when you run the Frida scripts.

You can install frida by going to your console and typing:

```
$  pip install frida-tools
```

Then run the following in your command line:

```
$  sudo sysctl kernel.yama.ptrace_scope=0
```

It allows tracing non-child processes on linux.

You know that this bot is calling a function named “tweet”, whose argument decides what the bot would tweet. You can run a python script “recon.py”(which is provided to you) to understand what kind of argument the “tweet” is receiving. Then you will use a frida script “attack.py” to change these arguments and make the bot tweet its immense admiration for Pakistan (Try 3,4 different arguments). If you use the right argument, you can see the output of the program change in real time.

In return, the National Cyber Security Division will appoint its chief security analyst (if you want, of course).

Submission Guidelines:

1. **Submit the attack.py file along with a screenshot of the desired output.**

Task 2: Defenses [20 points]

Let's start easy and have a look at defenses. You have been provided with a file called 'defenses.py'. This file implements a very (very) simple version of a program running simulator. You are supposed to implement the defenses in this file. Be careful to read instructions in the file carefully and only modify portions you are allowed to modify.

For this task you need to add the following defenses into defenses.py file:

1. ASLR
2. StackGuards (Stack Canaries)
3. Non-executable Stack

Further instructions are given in the **defenses.py** and **task2_guide.pdf** files.

Tip: To have a better understanding, I recommend reading through all of the code in the file. Printing program memory after each instruction may assist you in understanding what is going on.

Submission Guidelines:

1. **Submit the defenses.py file for this task.**

Task 3: Return-to-libc Attack [50 points]

Turning Off Counter-measures

Ubuntu and other Linux distributions have implemented several security mechanisms to make the buffer-overflow attack difficult. To simplify our attacks, we need to disable them first. Later on, we will enable them one by one, and see whether our attack can still be successful.

Address Space Randomization

Ubuntu and several other Linux-based systems use address space randomization to randomize the starting address of the heap and stack. This makes guessing the exact addresses difficult; guessing addresses is one of the critical steps of buffer-overflow attacks.

In this task, we disable this feature using the following command:

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

The StackGuard Protection Scheme

The GCC compiler implements a security mechanism called StackGuard to prevent buffer overflows. In the presence of this protection, buffer overflow attacks will not work. We can disable this protection during the compilation using the -fno-stack-protector option. For example, to compile a program example.c with StackGuard disabled, we can do the following:

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

Non-Executable Stack

Ubuntu used to allow executable stacks, but this has now changed: the binary images of programs (and shared libraries) must declare whether they require executable stacks or not, i.e., they need to mark a field in the program header. Kernel or dynamic linker uses 3 this marking to decide whether to make the stack of this running program executable or non-executable. This marking is done automatically by the recent versions of gcc, and by default, stacks are set to be non-executable. To change that, use the following option when compiling programs:

1. For executable stack

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

2. For non-executable stack:

   ```
   $ gcc -z noexecstack -o test test.c
   ```

Because the objective of this task is to show that the non-executable stack protection does not work, you should always compile your program using the "-z noexecstack" option in this lab.

Configuring /bin/sh

In Ubuntu 16.04, the /bin/sh symbolic link points to the /bin/dash shell. The dash shell in Ubuntu 16.04 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. The dash program in Ubuntu 12.04 does not have this behavior. 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. We use the following commands to link /bin/sh to zsh:

```
$ sudo ln -sf /bin/zsh /bin/sh
```

Different Environment Variables while running GDB

This is the layout of your program on the memory. If the environment variables inside the gdb could be different from those in your shell, it will mess up the addresses that you will be using for the Buffer Overflow Attack. You can checkout the environment variables in your shell using:

```
$ env
```

And inside the GDB using gdb-peda$ show env Usually GDB adds two new environment variables LINES and COLUMNS. You can unset them using:

```
gdb-peda $ unset LINES

gdb-peda $ unset COLUMNS
```

Make sure that the environment variables are the same inside and outside the gdb, to make sure that your attacks work outside of gdb as well.

The Vulnerable Program

The program provided to you in the 'retlib.c' file has a buffer overflow vulnerability. It first reads an input of size 300 bytes from a file called badfile into a buffer of size BUF SIZE, which is less than 300. Since the function fread() does not check the buffer boundary, a buffer overflow will occur.

This program is a root-owned Set-UID program, so if a normal user can exploit this buffer overflow vulnerability, the user might be able to get a root shell. It should be noted that the program gets its input from a file called badfile, which is provided by users. Therefore, we can construct the file in a way such that when the vulnerable program copies the file contents into its buffer, a root shell can be spawned.

Compilation

Let us first compile the code and turn it into a root-owned Set-UID program. Do not forget to include the -fno-stack-protector option (for turning off the StackGuard protection) and the "-z noexecstack" option (for turning on the non-executable stack protection). It should also be noted that changing ownership must be done before turning on the Set-UID bit, because ownership changes cause the Set-UID bit to be turned off:

```
$ gcc -fno-stack-protector -z noexecstack -o retlib retlib.c

$ sudo chown root retlib

$ sudo chmod 4755 retlib
```

Step 1: *Finding out the addresses of libc functions*

In Linux, when a program runs, the libc library will be loaded into memory. When the memory address randomization is turned off, for the same program, the library is always loaded in the same memory address (for different programs, the memory addresses of the libc library may be different). Therefore, we can easily find out the address of system() using a debugging tool such as gdb. Namely, we can debug the target program retlib. Even though the program is a root-owned Set-UID program, we can still debug it, except that the privilege will be dropped (i.e., the effective user ID will be the same as the real user ID). Inside gdb, we need to type the run command to execute the target program

once, otherwise, the library code will not be loaded. We use the p command (or print) to print out the address of the system() and exit() functions (we will need exit() later on).

```
$ touch badfile

$ gdb -q retlib

gdb-peda$ run

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$ quit
```

It should be noted that even for the same program, if we change it from a Set-UID program to a non-Set-UID program, the libc library may not be loaded into the same location. Therefore, when we debug the program, we need to debug the target Set-UID program; otherwise, the address we get may be incorrect.

Submission Guidelines

1. **Submit the Addresses of both these functions in the PDF File**

Step 2: *Putting the shell string in the memory*

Our attack strategy is to jump to the system() function and get it to execute an arbitrary command. Since we would like to get a shell prompt, we want the system() function to execute the "/bin/sh" program. Therefore, the command string "/bin/sh" must be put in the memory first and we have to know its address (this address needs to be passed to the system() function). There are many ways to achieve these goals; we choose a method that uses environment variables. You are encouraged to use other approaches. When we execute a program from a shell prompt, the shell actually spawns a child process to execute the program, and all the exported shell variables become the environment

variables of the child process. This creates an easy way for us to put some arbitrary string in the child process's memory. Let us define a new shell variable MYSHELL, and let it contain the string "/bin/sh". From the following commands, we can verify that the string gets into the child process, and it is printed out by the env command running inside the child process.

```
$ export MYSHELL=/bin/sh

$ env | grep MYSHELL

MYSHELL=/bin/sh
```

We will use the address of this variable as an argument to system() call. The location of this variable in the memory can be found out easily using the following program:

```
void main(){

    char* shell = getenv("MYSHELL");

    if (shell)

    printf("%x\n", (unsigned int)shell);
}
```

If the address randomization is turned off, you will find out that the same address is printed out. However, when you run the vulnerable program retlib, the address of the environment variable might not be exactly the same as the one that you get by running the above program; such an address can even change when you change the name of your program (the number of characters in the file name makes a difference). The good news is, the address of the shell will be quite close to what you print out using the above program. Therefore, you might need to try a few times to succeed.

Step 3: *Exploiting the buffer-overflow vulnerability*

We are ready to create the contents of badfile. Since the content involves some binary data (e.g., the address of the libc functions), we can use C or Python to do the construction.

Using Python

We provide you with a skeleton of the code, with the essential parts left for you to fill out.

You need to figure out the three addresses and the values of X, Y, and Z. If your values are incorrect, your attack might not work. In your report, you need to describe how you decide the values for X, Y and Z. Either show us your reasoning or, if you use a trial-and-error approach, show your trials.

Using C

We provide you with a skeleton of the code, with the essential parts left for you to fill out.

You need to figure out the addresses in lines marked by ✰, as well as to find out where to store those addresses (i.e., the values for X, Y, and Z). If your values are incorrect, your attack might not work. In your report, you need to describe how you decide the values for X, Y and Z. Either show us your reasoning or, if you use a trial-and-error approach, show your trials

After you finish the above program, compile and run it; this will generate the contents for badfile. Run the vulnerable program retlib. If your exploit is implemented correctly, when the function bof() returns, it will return to the system() function, and execute system("/bin/sh"). If the vulnerable program is running with the root privilege, you can get the root shell at this point.

```
$ gcc -o exploit exploit.c

$./exploit // create the badfile

$./retlib // launch the attack by running the vulnerable program

# <---- You've got a root shell!
```

Submission Guidelines:

1. **Submit the ‘retlib.c’ file along with the ‘exploit.py’ or ‘exploit.c’ file (whichever one you used). Name the exploit file ‘task2_exploit.c’ or ‘task2_exploit.py’.**
2. **Submit a screenshot of the shell in the PDF file you will be creating.**

Task 4: No Exit [5 points]

Is the exit() function really necessary? Please try your attack without including the address of this function in ‘badfile’. Run your attack again, report and explain your observations.

Submission Guidelines:

1. **Submit the ‘retlib.c’ file along with the ‘exploit.py’ or ‘exploit.c’ file (whichever one you used). Name the exploit file ‘task3_exploit.c’ or ‘task3_exploit.py’**

Task 5: A Different Name [5 points]

After your attack is successful, change the file name of retlib to a different name, making sure that the length of the new file name is different. For example, you can change it to newretlib. Repeat the attack (without changing the content of 'badfile’). Will your attack succeed or not? If it does not succeed, explain why.

Submission Guidelines:

1. **Add the answer to the question with the new name you gave to this file in the PDF File you will be creating.**

Task 6 Modern Defenses [15 points]

1. Today most of the computers are 64-bit architecture based and explain how that helps further secure the system against Buffer Overflow Vulnerability. Focus on ASLR in your answer.

2. Modern Software is mostly developed using a framework. Such as Vue.js (for Web Applications).Some Frameworks have built in security mechanisms against buffer overflow vulnerabilities. Explain how these security mechanisms work.

3. You have read about the 3 most common defenses against Buffer Overflow vulnerabilities. Namely ASLR, Non-Executable Stack, Stack Guards. Think of a new defense on your own against Buffer Overflow vulnerability and explain the defense. Also include in your answer if the attack is on the system side (OS) or program side.

Submission Details

1. Document your answers in a PDF file. Name the file as report.pdf
2. Submit the following files:
   2.1. defenses.py
   2.2. retlib.c
   2.3. task2_exploit.c or task2_exploit.py
   2.4. task3_exploit.c or task3_exploit.py
   2.5. Report.pdf
3. Submit all files on LMS in a single ‘ZIP’ file format named {rollnumber}_A2.zip e.g 20100177_A2.zip