ECE 568S
Lab #1: Software Vulnerabilities

Background

In this lab you are asked to take on the role of a penetration tester. You will be evaluating whether various small applications can be exploited by an attacker. In the course of this lab you will learn about software vulnerabilities that might reveal confidential data, leak memory addresses, and/or allow code to be injected.

This lab consists of five sections, totalling 20 marks. This lab may be done individually, or in groups of two. Final submissions are due by **11:59pm on Friday, January 31st**.

A Word of Caution

Software vulnerabilities and the methods used to exploit them can be impacted by many factors, including the CPU, operating systems, compiler and libraries installed on the system you are evaluating. As a result, please make sure you do your work on the ECF servers: solutions that you craft on your personal laptop (or some other UofT computer lab) may fail to work properly when we try to mark your work on ECF. Please do your work on the ECF workstations, or by logging in remotely to **remote.ecf.utoronto.ca**.

Setting Up

Download the source file associated with this lab, unpack it, and compile the "target" applications:

```
$ mkdir ece568
$ chmod og-rwx ece568
$ cd ece568
```

Place your copy of the "ece568_lab1.tar.gz" file in this directory, and then:

```
$ tar zxvfp ece568_lab1.tar.gz
$ cd lab1/targets
$ make
```

target1: Information Leakage [2 marks]

We will be starting off with a very simple exploit that involves no code injection. When you run "target1" you will see that it simply prints its name and then exits:

```
$ ./target1
Running ./target1
$
```

Next, move into the "exploits" directory, compile the code there, and run the "exploit1" program:

```
$ cd ../exploits
$ make
$ ./exploit1
Running ./target1
$
```

If you look at the source code for exploit1.c, you will see that it is calling execve(); this causes **exploit1** to start running **target1**:

```
execve(TARGET, args, env);
```

Your assignment is modify **exploit1.c** so that it provides malicious input, via the *args* and/or *env* parameters, in order to exploit **target1**. Your goal is to have **exploit1** cause **target1** to "leak" its secret string:

```
$ ./exploit1
Running ./target1
***ECE568 SECRET STRING!
$
```

Looking at the source code in **target1.c**, it is using the strncpy() function to copy in whatever is provided in argv[0]:

```
strncpy(programName, argv[0], PROGRAMNAME_LEN);
printf("Running %s\n", programName);
```

Have a careful look at the documentation for *strncpy()*. In particular, look at how it handles the case where the source string (including the terminating null character) is longer than the destination buffer.

Please ensure that your program output looks exactly like what's shown in the example, above. The "secret string" should print on the second line, and nothing other than what's shown above should be printed. For example, this would **not** be counted as correct output:

```
$ ./exploit1
ECE568 exploit1
Calling execve…
Running ./target1 ***ECE568 SECRET STRING!
$
```

Also, please note that when we are marking your assignment we will be using a "target1" with a different "secret string"; you need to *actually* exploit the vulnerability in **target1**. Modifying **exploit1.c** and replacing the execve() call with:

```
printf("Running ./target1\n***ECE568 SECRET STRING!");
```

will **not** be counted as an acceptable solution. ;-)

target2: Leaking Addresses [3 marks]

It is sometimes valuable to be able to determine where various variables are located in memory. In this section you will be extending what you learned in the last part, and exploiting a software vulnerability in order to leak a memory address from the target application.

In **target2.c** the *printProgramName()* function contains a local array called *programName*:

```
void
printProgramName(const char * buffer)
{
     // Print the program name

     char programName[PROGRAMNAME_LEN];
```

Your assignment is to modify **exploit2.py,** so that it exploits a vulnerability in **target2** and determines the memory address of the *programName* array.

If you run target2 in gdb, you can find the address of *programName*:

```
$ cd ../targets
$ gdb target2
(gdb)  break printProgramName
(gdb)  run
(gdb)  print &programName
$1 = (char (*)[10]) 0x56820e96
(gdb)  quit
```

As you're working on this part, you will also need to examine the stack frame in order to get other addresses and memory offsets. (The gdb "*info frame*" command would be a good place to start.) The lecture slides and our discussion of stack layout will be good additional references.

You may have noticed that **target2** is compiled with some extra code. This code in **stackSetup.c** bypasses a number of security features in the compiler and the operating system, such as page-protection and address-space layout randomization (ASLR). This means that the address of *programName* will remain constant; this will make it easier for you to verify that your solution is correct. (Some of these features will help you later in this assignment, as well.)

The exploit code for this section is a simple Python script. (We are using Python because it is a bit simpler to capture program output and manipulate strings.) You are provided with some sample code in **exploit2.py** that might be helpful when creating your solution to this part:

```
# Print the target2 output as a byte-array
print( result.stdout )

# Print the target2 output as a string of hex characters
print( result.stdout.hex() )
```

```
# Convert a byte-array to an int, and print the value
byteArray = b'\x78\x56\x34\x12'
address = int.from_bytes(byteArray, "little")
print("0x%016x" % address)
```

Your goal is to modify **exploit2.py** so that it runs **target2**, exploits the vulnerability, and prints (only!) the address of *programName*. Your code is likely working correctly if the output matches what you found in gdb, above:

```
$ cd ../exploits
$ ./exploit2.py
0x0000000056820e96
$
```

Again, please be careful that your program output is exactly in the format shown, above: just a single line, printing the address in the format shown above. (You will need to remove the extra print lines that are in the sample code.)

As before, we will be testing your code with a different **target2** and a different address.

target3: Buffer Overflow + Code Injection [5 marks]

In this section you will be extending what you learned in the last part and adding in the injection of shellcode. The lecture notes and the papers available on Quercus by Crispin Cowan ("*Buffer Overflows: Attacks and Defences for the Vulnerability of the Decade*") and Aleph One ("*Smashing the Stack for Fun and Profit*") will all be useful references.

When you run **target3** it simply prints a line on the screen and exits:

```
$ cd ../targets
$ ./target3
Running target3…
$
```

Your task in this section is to exploit **target3** by injecting it with shellcode. The shellcode you are to use in this section is provided in **./exploits/shellcode.h**:

```
static char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff./shell";
```

This code has been slightly modified from the code described in the Aleph One paper; it has been updated to run on a 64-bit architecture, and has been modified to call a local program called **shell**:

```
$ cd ../exploits
$ ./shell
ECE568: Successful exploit!
```

You are provided an example **exploit3** that uses *execve()* to call **target3:**

```
$ ./exploit3
Running target3…
$
```

Your assignment is modify **exploit3.**c so that it provides malicious input, via the *args* and/or *env* parameters, in order to exploit **target3** via a buffer-overflow exploit. Your goal is to cause **target3** to run the **shell** program:

```
$ ./exploit3
Running target3...
ECE568: Successful exploit!
$
```

Again, please be careful that your program output is exactly in the format shown, above. As before, we will be testing your code with a slightly different set of inputs. (For this section, however, you can assume that the memory addresses in the target application will remain the same when we test your exploit.)

target4: Double Free + Code Injection [5 marks]

In this section you will be doing another shellcode injection. However, instead of a buffer-overflow, you will be exploiting a "double free" vulnerability.

The target for this lab (**target4.c**) perform some dynamic memory allocation. However, instead of malloc() and free(), it uses tmalloc() and tfree(); you will find the source code for these functions in **tmalloc.c**.

If you look closely at **target4.c** you will see that it has a bug:

```
char *    p = tmalloc(72);
char *    q = tmalloc(120);

tfree(p);
tfree(q);

p = tmalloc(192);

strncpy(p, argv[0], 192);

tfree(q);
```

The last line (above) should free the pointer "p", but it accidentally frees the pointer "q". The results in a very common issue, referred to as a "double-free" vulnerability. Our lecture notes will be a good reference for this section.

You will want to focus on these lines inside tfree() in particular:

```
if (q != NULL && GET_FREEBIT(q)) /* try to consolidate leftward */
{
        CLR_FREEBIT(q);
        q->r = p->r;
        ((CHUNK *)(uint64_t)(p->r))->l = (uint32_t)(uint64_t)q;
        SET_FREEBIT(q);
        p = q;
}

q = RIGHT(p);
if (q != NULL && GET_FREEBIT(q)) /* try to consolidate rightward */
{
        CLR_FREEBIT(q);
        p->r = q->r;
        ((CHUNK *)(uint64_t)q->r)->l = (uint32_t)(uint64_t)p;
        SET_FREEBIT(q);
}
```

One of those "if" blocks will be useful for you – and you'll want to avoid the other one from executing. Have a close look at how the CHUNK data structure is defined and used, and how the "free" bits are stored.

As in the last section, your task is to exploit **target4** by injecting it with shellcode. You will be using the same shellcode as in the prior section.

You are provided an example **exploit4** that uses *execve()* to call **target4:**

```
$ ./exploit4
Running target4…
$
```

Your assignment is modify **exploit4.c** so that it provides malicious input, via the *args* and/or *env* parameters, in order to exploit **target4** via a double-free exploit. Your goal is to cause **target4** to run the **shell** program:

```
$ ./exploit4
Running target4...
ECE568: Successful exploit!
$
```

Again, please be careful that your program output is exactly in the format shown, above. As before, we will be testing your code with a slightly different set of inputs. (For this section, however, you can assume that the memory addresses in the target application will remain the same when we test your exploit.)

target5: Format String Vulnerability + Code Injection [5 marks]

In this final section, you will be doing one last shellcode injection, by exploiting a "format string" vulnerability in **target5**:

```
snprintf(buffer, BUFFER_LEN, args);
```

Our lecture notes will be a good reference for this section. Also note that, when calling execve(), your argv[] array can contain multiple strings; they will be adjacent to one another in memory. Each string is terminated with a '\0', and empty strings will add an additional '\0' byte. For example:

```
char * argv[]={"\x11\x22\x33", "", "\x55\x66", "", "", NULL};
```

will place the following bytes in memory:

```
0x11  0x22  0x33  0x00  0x00  0x55  0x66  0x00  0x00  0x00
```

As in the last section, your task is to exploit **target5** by injecting it with shellcode. You will be using the same shellcode as in the prior section.

As before, you are provided an example **exploit5** that uses *execve()* to call **target5:**

```
$ ./exploit5
Running target5…
$
```

Your assignment is modify **exploit5.c** so that it provides malicious input, via the *args* and/or *env* parameters, in order to exploit **target5** via a format-string exploit. Your goal is to cause **target5** to run the **shell** program:

```
$ ./exploit5
Running target5...
ECE568: Successful exploit!
$
```

Again, please be careful that your program output is exactly in the format shown, above. As before, we will be testing your code with a slightly different set of inputs. (For this section, however, you can assume that the memory addresses in the target application will remain the same when we test your exploit.)

Submitting

You will need to submit the source code for your exploits, along with a brief explanation of what you did. For each target please explain, in at most 100 words, what the vulnerability was and how you exploited it. Please place these explanations in a single file, called **explanations_lab1.txt**.

IMPORTANT: Your **explanations_lab1.txt** file must start with the names and student numbers of your group members, prefixed by #:

```
#name1, studentNumber1, email1
#name2, studentNumber2, email2
```

For example:

```
#Jane Skule, 998877665, jane.skule@utoronto.ca
#John Skule, 998877556, john.skule@utoronto.ca
```

It is very important that this information (and the filename) is correct: your mark will be uploaded to Quercus by student number and the email you give here will be how we will get the results of the lab back to you. (We frequently receive submissions with incorrect student numbers, and missing lab partners: please take a moment and double-check.)

Submit your lab assignment via the ECF *submit* command:

```
submitece568s 1 explanations_lab1.txt exploit1.c exploit2.c
exploit3.c exploit4.c exploit5.c
```

Please note the number one (not the letter “L”) as the first parameter of the submit command. A man page on ECF (man submit) contains more information on the ECF submit command. The submission command for the lab will cease to work after the lab is due.