NUS CS-3235: Computer Security March 13, 2020

Assignment 2

Lecturer: Reza Shokri *Student: Name Number*

Instructions

Set up : This assignment consists of three coding exercises. First, download the assignment archive from Luminus. Unzip the archive as shown below:

```
$ unzip a2.zip
```

Report : Please write a short report A01xxxx_report.pdf (replace `A01xxxx` with your student id.) explaining how you solved the three exercises in three sections.

- We will not consider exploits that do not include a write-up so please make sure to include them in the submission.
- The report should **include screenshots** wherever you feel it is needed for better explanations similar to the tutorial notes.
- When you use addresses in your payloads **explain how you found them**. Any valid way to find addresses (e.g.: for stack buffers or global variables) is fine even if it was not shown in the tutorials.
- Be sure to mention in your report **how to run the exploit** (e.g., "I used the following command to run the exploit: `./vuln < payload`").
- We recommend you use the Ubuntu VM to solve the tasks.
- **DO NOT CHANGE THE SOURCE CODE**!

- Use font size 12 for the report. You are strongly encouraged to use LaTeX (use the provided template).

Submission : In order to submit the assignment, please make sure to first clean each directory from unnecessary files (e.g., .gdb_history or peda-session-*). Leave inside all the payloads and scripts (if any) that you used in the corresponding folder. After that, please archive the whole directory structure in a ZIP file like this:

```
$ zip -r A01xxxx_a2.zip a2
```

Replace `A01xxxx` with your student id. Please submit this archive to LumiNUS.

We will ignore all the submissions that violate any of the above. You have **2 weeks** to submit your solutions.

Compiling Before starting, make sure to disable address randomization (ASLR). (use the command in the tutorial notes part 2). Some exercises are not possible to solve with address randomization. Each task is placed in a different directory in the assignment archive. Each task has its own Makefile. Please make sure to **build the binary executable with the make command**. Do not compile the files without "make" since they have special compile options to enable some attacks.

1 Buffer Overflow

The code is in the buffer_overflow folder. In this task there is input read from 2 files ./exploit1 and ./exploit2. The data is read into 2 `BUFSIZE` buffers but in the end all the data is copied into one stack buffer of the same size. This causes a buffer overflow.

Goal: Create the 2 files ./exploit1 and ./exploit2 such that when you run the program it spawns a shell.

Helpful hints:

- If you cannot understand immediately how the bytes are placed in buf then try to create simple ./exploit1 and ./exploit2 files (e.g. one has As and the other Bs) and inspect the program in gdb.
- We recommend to generate ./exploit1 and ./exploit2 using a script. In this way it is easier to write non printable characters and make small adjustments.
- The data memory mappings are executable (including the stack). This means a shellcode attack can be used. You can use the shellcode from here.
- Be careful when you overflow the buffer. Some local variables might be overwritten. Take care when overwriting the `idx`, `byte_read1`, `byte_read2` variables as they may cause the for loop to run indefinitely. To get around this, you may want to place the same values they had before being overwritten.

This exercise is only considered solved if you are able to get a shell by running the `./buffer_overflow` file **outside gdb**. Write the report explaining your exploit in the A01xxxx_report.pdf Buffer Overflow section.

2 Format String Attack

The code is in the format_string folder. The task calls `printf(buf)` where buf is controlled by the user. There is a format string bug which has to be exploited in order to change the value of the global `jackpot` variable.

Goal Use the format string bug to write the value 0x1337 (4919 decimal) into the `jackpot` variable. The check has to pass and the string "You Won!" must be printed at the end.

Helpful hints:

- Follow the tutorial notes (part2) to find out how to overwrite global variables using a format string bug.
- Use the %c specifier with padding to force printf to write the amount of bytes that you want. (ex: %100c will write 100 bytes).
- It is recommended to write your payload in a file since you cannot type unprintable characters directly.

It is not important what is printed to the terminal as long as you force the program to take the if branch that prints "You Won!". Write the report explaining your exploit in the A01xxxx_report.pdf Format String Attack section. Mention in the report how the exploit is run (e.g.: `./format_string < payload`).

3 Return-oriented Programming

This task reads input both from the standard input and from a file ./exploit. At first glance there is no buffer overflow occurring here.

Goal: Find a way to cause a buffer overflow. Use the "Return-oriented Programming" (ROP) technique to read any file that the user can access (call `open`, `read` - from file, `write` - standard output). For this task, you may need to perform independent research.

Helpful hints:

- To find a way to cause the buffer overflow check the types of the variables being compared. Is there something wrong with having signed long variables representing lengths?
- In order to read a file and print you will have to call this sequence of functions:
  - `open(filename, 0)`
  - `read(fd, buffer, bytes)`
  - `write(1, buffer, bytes)`
- The filename has to be the address of the string that is the filename. First you have to write the file that you want to open somewhere (maybe on the stack?).
- The `fd` is the file descriptor returned by open (what is that value? can you predict it?)
- The `buffer` argument has to be a location in memory which is writable. There are many such locations including the stack. You can pick any location as long as you do not overwrite important variables.

- The number for bytes can be any amount even if it is larger than the actual file. Make sure to pick a big enough number to be able to read the whole file.
- We write to file descriptor 1 because that is the file descriptor for the standard output.
- The `open`, `read` and `write` functions can be found in the standard C library which is in the process' address space when it is run. You can find their addresses in gdb.
- To be able to control the arguments of the functions you will need gadgets that will write values into the `rdi`, `rsi`, `rdx` registers. Check tutorial notes (part2) to remember how to find gadgets. If the binary does not contain the gadgets that you need, there are plenty of gadgets in the C standard library (hint: help `asmsearch`, `vmmap`).

It is possible to read any file by spawning a shell. Do not solve the task in this way. Solve it by calling `open`, `read` and `write`. You can pick any file to read and print, for example you can use the ./rop.c file. Write the report explaining your exploit in the A01xxxx_report.pdf Return-oriented Programming section. If you need to use input through the keyboard then specify what input in the report.