COEN225 Lab 6: Format String Vulnerability

Overview

The learning objective of this lab is for students to gain experience with format string vulnerabilities. A format string vulnerability occurs when functions that handle format strings such as `printf()` is provided unsanitized user input.

In this lab, students will develop an attack that takes advantage of a format string vulnerability that allows the user input to control a format string to read and write data. Students will understand how format strings can be used for information leakage to carry out further attacks.

Lab Submission

Perform the tasks below and submit your responses to questions noted in the task submission sections. Include any source code you used to generate exploits or other tests.

The lab work and submission must be completed individually but working with other students is allowed. The Camino message board is the preferred method for asking questions so that the instructor and students can respond and allow all students to learn from it.

Lab Tasks

Initial Setup and Objectives

First, disable ASLR. Then review lab6.c and compile it using GCC (no extra flags are needed)

```
$ su
# sysctl -w kernel.randomize_va_space=0
# exit
$ gcc -o lab6 lab6.c
```

When the program starts, a random number is generated to be the secret value. This simulates a real-world scenario where data is not a part of the source code and is generated at runtime or is read in from a file such as private keys. This secret number is stored in `secrets[0]`. The goal of this lab is to leak the contents of the `secrets[0]` to obtain the secret number, and then overwrite `secrets[1]` to set it as the same value as `secrets[0]` to complete the challenge.

Note that unlike previous labs, this program run in an infinite loop until the challenge is complete, so the secret number will not change unless you terminate and re-run the program.

Important: You may NOT use GDB to obtain the secret number. You can accomplish all of the tasks in this task without GDB. You may use GDB to explore and developing techniques to attack the program, but you must complete all tasks by running the program without GDB.

The final goal is to overwrite `secrets[1]` to be the same as the secret value `secrets[0]` which is randomized at the start of the program. This objective can be broken up into the following objectives:

1. Find the absolute address of `secrets[0]` where the secret value is located
2. Leak the value of `secrets[0]` using the address
3. Modify the value of `secrets[1]` to match the secret value

All of these can be accomplished using format string vulnerabilities. Note that if you close and re-launch the program, it will generate a new random value for the secret, so all of these milestones must be completed with the same running process.

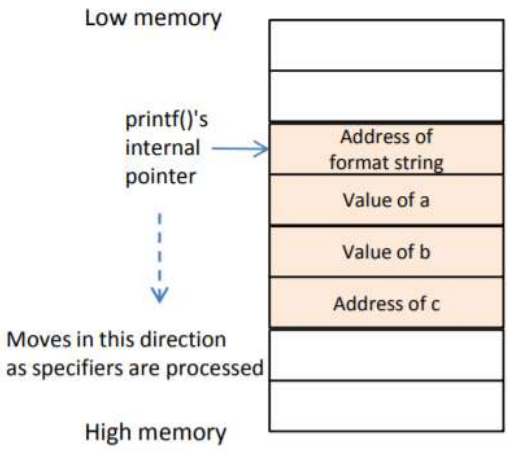
Task 1: Stack Information Leakage

Objective: Find the absolute address of `secrets[0]` where the secret value is located

Let's start by understanding the program. The value of `secret[0]` and `secret[1]` are located on the heap. However the address of `secret` (which is the same as `&secret[0]`) can be found on the stack, because the variable `secret` is allocated on the stack (it holds a pointer to the heap). You can take advantage of this information by using the format string vulnerability present in the program. Once it enters the loop, the program will ask for 2 inputs: first a hexadecimal number to write to the stack, and secondly a format string which is passed to a `printf()` function.

The user input is not sanitized, so you can pass a format string that has mismatched tokens with the number of arguments provided to `printf()`. As no additional arguments are passed to `printf()`, any specifier tokens in the format string such as %x will advance the printf function's variable argument pointer past its function frame into a different function's frame if given enough specifier tokens. For a `printf()` statement like the one below, the argument pointer will go down the stack 4 bytes at a time for each specifier.

```
printf ("a has value %d, b has value %d, c is at address: %08x\n",
          a, b, &c);
```



For this task, your goal is to exploit the vulnerability and print out the absolute address of `secrets[0]`. Use your knowledge of how local variables are constructed on the stack and other tips like the addresses/data printed out by the program to pinpoint where the secrets variable is relative to the others.

Task 1 Submission

What was the value of `secrets`? Remember that it is a *pointer* to the heap.

Task 2: Leaking the value of an arbitrary memory location

Objective: Leak the value of `secrets[0]` using the address

Now that we have the value of secrets and therefore the address of `secrets[0]`, we want to get the data stored at `secrets[0]` (where the random secret value is stored). This can be accomplished by using the %s specifier in a format string. The %s specifier interprets the data at the argument pointer as a string (a pointer to an array of characters). Therefore, you must align the specifier so that the corresponding argument pointer is located where the memory address of `secrets[0]` is in the stack. If done correctly, the value of `secrets[0]` will be printed as a string.

Tip #1: Check out how the random number is generated to see the full range of potential values. An ASCII character table may help as well.

Tip #2: If you are unsure it you have the correct answer, try replacing the random number generation code with a static value just for testing. This should not affect the location of the data to test your leakage attack.

Task 2 Submission

What is the secret value? Remember that you may NOT use GDB to obtain the secret value.

It may help you in the next task to write down the secret value represented as a character and as a decimal number.

Note that this is a random value so it will be different every time the program runs. Don't terminate the program now because you'll need it in the next task!

Task 3: Writing a value of an arbitrary memory location

Objective: Modify the value of `secrets[1]` to match the secret value

For the final step, we must write the secret value leaked in Task 2 into `secrets[1]`. This is similar to Task 2, except instead of using the %s specifier to read data at the given argument, you will now use the %n specifier; a specifier that writes data.

There are 2 additional things to keep in mind for this task.

First, in Task 2, the address to read was already available on the stack which made it easy to use. This time, we do not have the address of `secrets[1]` readily available on the stack so we have to insert it into memory. You can compute the address of `secrets[1]` given the address of `secrets[0]` because you know that `secrets` is an array of ints (4 bytes). It is possible to put this address into your format string, but addresses often include non-printable characters making it difficult to simply type the address in. In addition to the format string input, the program conveniently includes a `scanf()` call (thanks to a very nice professor), which will interpret your input as a hexadecimal value. Do not include the "`0x`" prefix. We can leverage this to put an address on the stack.

The other problem is that the value written to the address is dependent on the number of characters written so far. This can be adjusted pretty easily by using the width modifier on one or more specifiers so that you are writing the exact number of characters before the %n is processed.

You have successfully completed this challenge when you get the message: “Congratulations! secrets[1] matches SECRETVALUE”.

Task 3 Submission

What was the hexadecimal number and format string you used to get the success message?

Task 4: Address Space Layout Randomization

Enable ASLR in your session.

```
$ su
# sysctl -w kernel.randomize_va_space=2
```

Run lab6 again and use the technique developed in Tasks 1 to 3 to complete the challenge again.

Task 4 Submission

Were you successful with ASLR enabled? Does ASLR mitigate the attack?

END OF LAB