access_to_the_peanut_cave: CS557 Project 3

Thomas Le Baron
Team 1008
Computer Science
Worcester Polytechnic Institute
Worcester MA, USA
tlebaron@wpi.edu

Brittany Lewis
Team 1008
Computer Science
Worcester Polytechnic Institute
Worcester MA, USA
bfgradel@wpi.edu

***Abstract*—**

I. Finding access_to_the_peanut_cave

Our docker image for access_to_the_peanut_cave is located at:

The challenge binary is:

The solution is at:

Hashes obtained using linux md5sum:

The hash for the binary is:

The hash for the source file is:

II. Introduction

access_to_the_peanut_cave is a challenge binary that leverages three different types of challenges as locks that an attacker has to break through in order to launch a shell. In order to solve these locks an attacker must have knowledge of randomization in C, linux file read conventions, and buffer overflows. In this work we explain our challenge binary access_to_the_peanut_cave as well as providing details on how to exploit it.

III. Design

access_to_the_peanut_cave is a binary which involves a series of three challenges that need to be correctly completed in order to enter the secret peanut cave (and launch /bin/shh using execve to gain privilege escalation).

A. Lock 1: Random password

Our first lock was inspired by a CTF binary at THOMAS PUT THE LOCATION WHERE YOU FOUND THAT THINGY. In access_to_the_peanut_cave we ask for a key and then perform an xor with a key that we ́randomly ́ generated using the rand function and compare it to a value 0x9000ddo9 (goood dog).

Rand(): The core of the vulnerability comes from our incorrect usage of the rand function. We forgot to actually seed our random function! As a result, it will be given the default seed of 0. This means that we can easily create a separate script that calls rand the same way, and gets the same "random" number. This will allow us to get the correct value to open this lock.

Scanf(): With this lock we ran into an issue later in our code. In the original challenge binary, the way that scanf was called, it left behind a newline character. This meant that our gets function did not work since it stopped immediately at the new line character left behind by scanf. We changed the call to scanf to expect a new line character so that it would not leave behind issues that would affect our program's execution later on.

B. Lock 2: File IO Lock

Our second lock was inspired by a CTF binary at THOMAS PUT THE LOCATION WHERE YOU FOUND THAT THINGY. This lock uses an argument passed to the main function, and subtracts a particular key from it before utilizing it as the file descriptor for a read operation. The trick to this lock is that if the read function is passed a file descriptor of 0, it will read from stdin by default. This will then allow the attacker to input the correct password.

C. Lock 3: Time of Check, Time of Use Lock

Our third lock was our own design to utilize a classic buffer overflow to overwrite a value before it is used. In this case, we call execve with the values of buffer 2 (which is set to be the same as argv[1]). However, argv[1] was utilized early in the program and needed to be set to a particular value in order to pass lock 2. Not a problem! We have a gets call on a buffer allowing for a buffer overflow. This allows us to overflow into buffer2 and set the value to be "/bin/shh" in between when it is used to open lock 2, and when it is used in our third lock to call execve.

IV. Implementation

The following steps must be followed to get privilege escalation:

1) Compile the code with: `gcc -o peanut peanut.c`. Note that we do not have to disable any default stack protections such as stack canaries or ASLR.
2) The exploit can be run with this command: THOMAS PUT COMMAND HERE this includes the first argument as 0x1234, the value it needs to be to clear the second lock
3) On the first prompt type X which will correctly XOR with the random number to provide `0x9000dd09` and unlock the first lock

4) On the second prompt type ILOVEPEANUT unlocking the second lock
5) On the third prompt type a padding of size X followed by `/bin/shh`, this will overflow `buffer` into `buffer2` and in doing so allow us to pass `/bin/shh` as an argument to execve to launch the shell.

V. Exploitation

Our exploit involves a set of 4 different inputs in order to break each successive lock. The combination of these inputs allows us to break through the three layers of locks in order to launch a shell with privilege escalation.

A. Before program execution

Before the start of program execution there are several steps that need to be taken. First, we must find what the value of random is going to be. The easiest way of doing this is to create a program which simply creates a random number using rand and prints it out. In our case, we used a file called test.c to do this, the code for doing so is in lines 8-10 in our test.c file. Once we get that value (NUMBER FROM RANDOM) we can perform and XOR with `0x9000dd0g` from line 21 in our challenge binary (the first lock we need to break) in order to get a value we will need later in execution. That number turns out to be NUMBER.

We will also need to use GDB to disassemble the code in order to grab the locations of buffer and buffer2 so that we can know how much padding we need in order to overflow the correct value into buffer2 if we don't want to run the program multiple times. Alternatively, we found our padding simply by running the program once with strace and seeing what was passed to execve.

In addition, when we are calling the program we will need to give it an argument equal to 0x1234 (from line 23 of our challenge program) so that eventually when the program grabs a file descriptor, we will have an fd of 0 allowing us to read from stdin and break the second lock.

B. During program execution

When the program is running, there will be three prompts. On the first prompt, we have to input the value we got from XORing rand earlier, NUMBER. This will result in the comparison being true, and will open the first lock.

For the second lock, our program will already have pulled argv[1] and, if it was set to the correct number, the read function is now reading from stdin. Thus we just now need to provide it with the string it is expecting to read from the file. In this case as we see on line 27 of our challenge program that the value it is looking for is `ILOVEPEANUT`. When we input that into the prompt, we will pass the second lock.

Finally, we will need to make sure that buffer2, which will automatically copy the value in argv[1] (0x1234) is set to be /bin/shh instead. In order to do that, we have to do a buffer overflow. We could do that by coming through the disassembly before we begin (as previously stated), however in our case we just supplied an input of "AAAABBBBCCCC..." and then using strace to see what was passed to execve. Using this method we found that we needed a buffer of size X before inputting /bin/shh. We then redid our execution and put in out correct buffer and /bin/shh on the third prompt. This successfully launches a shell.

VI. Conclusion

In this work we have introduced a new challenge binary access_to_the_peanut_cave, a binary with a series of three locks that can be exploited to launch a shell. We also showed how to exploit this binary by explaining the vulnerabilities in the rand() function in C, the affect of using the read function on a given file descriptor, and how our buffer overflow allows the user to modify sensitive data to launch a shell.