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

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

V. Exploitation

VI. Conclusion