

Access_to_the_peanutcave_reboot: 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_PEAUTCAVE_REBOOT

Our docker image for `access_to_the_peanutcave_reboot` is located at:

- **Challenge binary:**
`access_to_the_peanutcave_reboot`
- **Solution:** described in detail in section IV
- **md5sum hash of the binary file:**
`dfe9ba8583638b2b3b57859dcd87ad3a`
- **md5sum hash of the source file:**
`b022244184a0dc658f94809925d6923`

II. INTRODUCTION

`access_to_the_peanutcave_reboot` is a challenge binary that leverages multiple 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_peanutcave_reboot` as well as providing details on how to exploit it.

III. DESIGN

`access_to_the_peanutcave_reboot` is a binary which involves a series of three challenges that need to be correctly completed in order to enter the secret peanut cave. The inner part of the cave gives great power to anyone who can reach it (launch `/bin/shh` using `execve` to gain privilege escalation for instance). The main idea of this challenge is to show that implementing a series of weak protection is not enough to secure a program. On top of the protections added to this binary, the stack canaries and ASLR are enabled.

The three locks are the following:

- **Random password:** To enter the inner room, the user have to enter a correct password. By design, the password should be random since generated by the `rand()` function.
- **File IO:** To even enter the password, the user needs to point the program to the right input.
- **No arbitrary execution:** The executed instruction depends of the value used to point to the correct input (step 2) to enter the password. Therefore, the user shouldn't be able to execute an arbitrary instruction.

A. Lock 1: Random password

Our first lock was inspired by a CTF binary challenge named `random` on `pwnable.kr`. In `access_to_the_peanutcave_reboot` 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 the value `0x9000dd09` (good dog).

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.

B. Lock 2: File IO Lock

Our second lock was inspired by a CTF binary challenge named `fd` on `pwnable.kr`. 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 to access `execve`.

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 `buffer2` (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 the privilege escalation:

- 1) Compile the code with: `gcc -o access_to_the_peanutcave_reboot access_to_the_peanutcave_reboot.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: `access_to_the_peanutcave_reboot 4660`. This includes the first argument as `0x1234`, the value it needs to be to clear the second lock.
- 3) On the first prompt type `4220229742` which will correctly XOR with the random number to provide `0x9000dd09` and unlock the first lock
- 4) On the second prompt type a padding of size 32 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 3 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 (`1804289383`) we can perform and XOR with `0x9000dd09` from line 26 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 `4220229742`.

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 20 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, the `argv[1]` needs to be equal to `4660` in order to the file descriptor used in the `read()`. Without it, the program will simply stop since there will be no other valid file descriptor number. Otherwise, it continues smoothly.

When the program is running, there will be two prompts. On the first prompt, we have to input the value we got from XORing `rand` earlier, `4220229742`. This will result in the comparison being true, and will open the lock.

After passing the lock, our program is designed to execute the command specified by the first argument, which must be `4660` in order to be able to reach this state. We will need to make sure that `buffer2` 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 32 before inputting `"/bin/shh"` (we could also have use `gdb` to verify how far `buffer` is from `buffer2` on the stack. We then redid our execution and put in our correct buffer and `"/bin/shh"` on the second prompt. This successfully launches a shell.

VI. CONCLUSION

In this work we have introduced a new challenge binary `access_to_the_peanutcave_reboot`, a binary with a series of 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.