Nicola Mores, 04/05/2024

Challenge: RandomPasswordGenerator 2.0 (RPG2)

Background

This challenge is about Reverse Engineering, a technique in which an attacker can try to understand how a system or a program works, without having direct access to the high-level code it uses, but only to the ELF file it produces. In real life this technique can be used in order to produce malware, modify and crack softwares, by bypassing the licensing mechanisms.

Vulnerability

In this challenge we have two main vulnerabilities: the predictability, given a particular seed, of the C `rand()` function and the weak seed used in the implementation. The `rand()` function will always generate the same sequence of numbers given the same seed value [1]. So, knowing the seed used, it is possible to predict the future "random" values. In addition to this, in our implementation the server uses a PID (process ID, an identifier given to every process running in a system) as its seed. This is a bad seed choice due to the limited amount of values that PID can have: 2^{15}-1 by default on 32 bit systems and up to 2^{22}-1 on 64 bit ones [2][3]. These two vulnerabilities could lead to brute force attacks. In order to prevent these attacks, we could use a more secure random generator, use more secure seeds or, since the main problem is that we were able to easily find out how the random was initiated and used, apply some anti reversing techniques such as obfuscation and anti disassembly.

Solution

In order to complete this challenge we can try to use a decompiler, like Ghidra, in a way to reason about how the software works by looking at one of its possible decompiled code. After disassembling and analyzing the program with Ghidra, since it seems obfuscation wasn't applied this time, we can start by exploring the main function. Inside this function we can notice how the program uses the C random function and in particular how it uses the PID of the process, retrieved with the `getpid()` function, as an initial seed through the `srand()` function call. Then we can see how it calls an internal function `rand_pass()`, that we can imagine actually creates the random password, and stores its return value in a variable, let's call it `generated_pswrd`. Then it asks the user to guess the password and will print the challenge's flag if the guess is correct, otherwise it will tell us what was the correct answer. This procedure will be executed 2 times, due to the for loop, after which the process will end. Let's have a look at the `rand_pass()` function in order to understand how the password is generated. There we can see how it creates a variable, assigns to it some allocated memory, based on the function parameter, and will use it as return value. This suggests that this variable is the generated password and that the function parameter (0x10 or 16 in our main call) will be its length. Our assumption is also supported by the for loop that will iterate for `param_1` times. In the for loop the program will get a random number from the `rand()` call and then use it in order to calculate, through a mathematical function, a char that will be stored in the i^{th} cell of memory previously allocated. Since we now know how the program works, we now have to find a way to get to the flag, and the only plausible way to do so is to break the (pseudo-) random number generator (PRNG) and correctly "guess" the password in two tries. We know that, given a specific seed, the random function provided by C will always generate the same set of numbers, so we might use this property, in addition to the bad choice of the seed, to break the PRNG by brute force. Let's start by coding a program that implements the same `rand_pass()` function and that gets the 1^{st} password from the server. Now we can just code a loop that uses `rand_pass()` to create a password and check if it is the same as the received one and if it is not starts a new iteration, incrementing the guessed seed by one. When we will find a collision in the passwords it means that we have found the seed used by the server side PRNG and so we will be able to predict the future "random" values and thus the future password generated. We can now just create another password, without modifying the PRNG, and send it to the server, receiving in this way the flag. Here is a possible python implementation that exploit the system vulnerabilities [4]:

```python
if(REMOTE):
    conn = pwn.remote(IP_ADDRESS, PORT)

# Retrieve first Password
conn.recvuntil(b":")
conn.sendline(b"1")

conn.recvuntil(b": ")
firstPsw = conn.recvline(keepends=False).decode()
print(f"First Password: {firstPsw}")

for i in range(32768):              # Iterate from 0 to (Default) Max PID possibile in Linux

    print(f"Try number {i}")
    libc.srand(i)                   # Initiate srand with seed equal to selected PID

    myPsw = ""                      # Create psw in the same way as rand_pass()
    for j in range(16):
        randNum = libc.rand()
        myPsw += chr(randNum % 0x5e + ord('!'))

    print(myPsw)

    if(myPsw == firstPsw):          # Check if myPsw is equal to the received one
        break

guessedPsw = ""                     # Create another psw using the same seed
for i in range(16):
    randNum = libc.rand()
    guessedPsw += chr(randNum % 0x5e + ord('!'))

conn.recvuntil(b":")                # Send our guessed psw and retrieve the flag
conn.sendline(guessedPsw.encode())

conn.recvuntil(b":\n")
flag = (conn.recvline(keepends=False)).decode()
print(f"Flag: {flag}")

conn.close()
```


[1] rand() Documentation:
https://en.cppreference.com/w/c/numeric/random/rand
[2] PID values in linux:
https://www.oreilly.com/library/view/mastering-linux-kernel/9781785883057/0ae8de90-e954-44d4-9c71-5bb0a8b8cb61.xhtml#:~:text=PIDs%20in%20Linux%20are%20of,sys%2Fkernel%2Fpid_max%20interface.
[3] PID values in linux:
https://unix.stackexchange.com/questions/16883/what-is-the-maximum-value-of-the-process-id
[4] My code implementation:
https://gitfront.io/r/NikoMrs/PrSpp9XjTGu2/EthicalHacking2024/blob/Assignment6/solution.py