

Race Conditions

Introduction

Let's say we are tasked with testing the security of an online shopping web application. Many questions pop up. Can we reuse a single \$10 gift card to pay for a \$100 item? Can we apply the same discount to our shopping cart multiple times? The answer is maybe! If the system is susceptible to a race condition vulnerability, we can do all this and more.

This room introduces the race conditions vulnerability. A race condition is a situation in computer programs where the timing of events influences the behaviour and outcome of the program. It typically happens when a variable gets accessed and modified by multiple threads. Due to a lack of proper lock mechanisms and synchronization between the different threads, an attacker might abuse the system and apply a discount multiple times or make money transactions beyond their balance.

Learning Objectives

After completing this room, you will learn about the following:

- Race conditions vulnerability
- Using Burp Suite Repeater to exploit race conditions

Along the way, you will also learn about:

- Threads and multi-threading
- State diagrams

Learning Prerequisites

To follow this room, we recommend familiarity with the HTTP protocol, web applications, and Burp Suite. The following rooms and modules are recommended to fill any knowledge gaps.

- [How the Web Works](#)
- [Packets and Frames](#)
- [Burp Suite: The Basics](#)

Multi-Threading

In this task, we will provide a quick overview of the following terms:

- Program
- Process
- Thread

- Multi-threading

Programs

A program is a set of instructions to achieve a specific task. You need to execute the program to accomplish what you want. Unless you execute it, it won't do anything and remains a set of static instructions.

Think of it as a recipe; you just downloaded a new coffee recipe that includes a variety of herbs, such as cardamom and cinnamon. These are the instructions:

1. Combine brewed coffee, cardamom, cinnamon, and cloves (if using) in a saucepan.
2. Heat the mixture over low heat for 5 minutes, stirring occasionally. Do not boil.
3. Strain the coffee into your mug.
4. Add milk if desired, and sweeten to taste with honey or sugar.

Unless someone carries out the above instructions, no coffee will be served!

Compare this to our minimal Flask (Python) "Hello, World!" server. The code below dictates that the app will listen on port 8080 and respond with a minimal greeting HTML page that contains "Hello, World!" However, we must run these instructions (program) before we expect to get any greeting pages.

Please note that the Flask code below is shown for demonstrative purposes only. We didn't provide an environment to run it as it is outside the scope of this room.

```
# Import the Flask class from the flask module
from flask import Flask

# Create an instance of the Flask class representing the application
app = Flask(__name__)

# Define a route for the root URL ('/')
@app.route('/')
def hello_world():
    # This function will be executed when the root URL is accessed
    # It returns a string containing HTML code for a simple web page
    return '<html><head><title>Greeting</title></head><body><h1>Hello, World!</h1></body></html>'

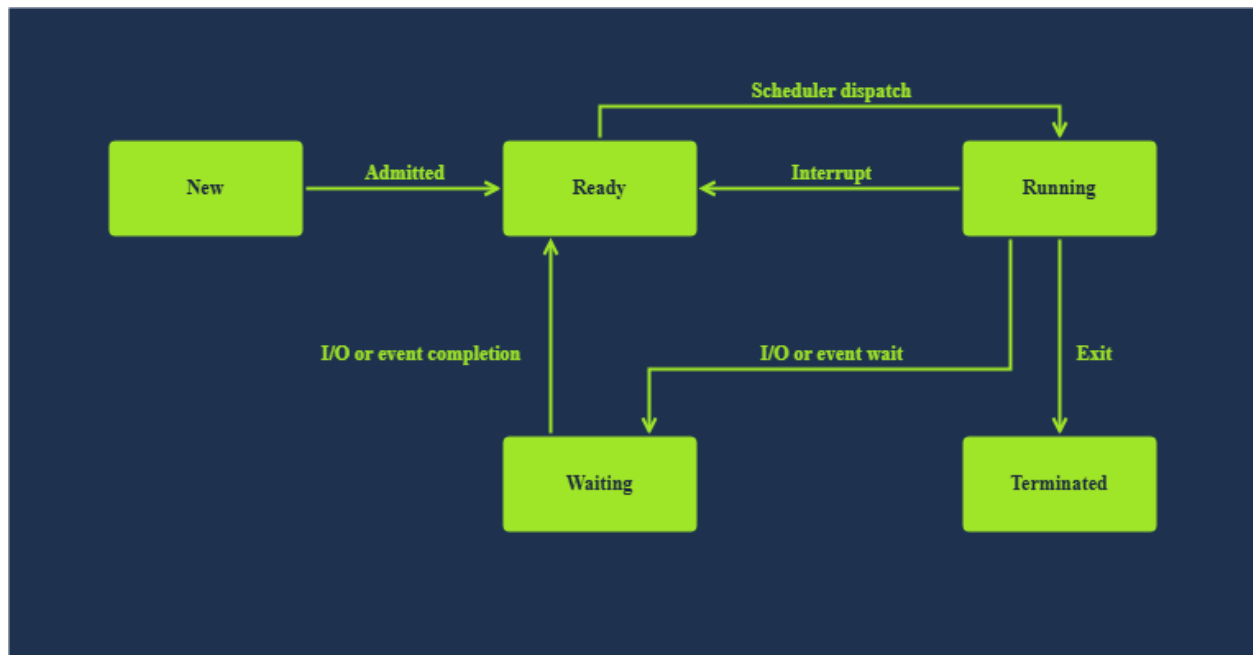
# This checks if the script is being run directly (as the main program)
# and not being imported as a module
if __name__ == '__main__':
    # Run the Flask application
    # The host='0.0.0.0' allows the server to be accessible from any IP address
    # The port=8080 specifies the port number on which the server will listen
    app.run(host='0.0.0.0', port=8080)
```

Processes

One afternoon, you decide to try out this new coffee recipe you downloaded online. You start going through the recipe one step at a time. You are in the process of making this coffee recipe. While in the “process” of “executing” the “instructions,” you might get interrupted by an urgent call. Or you might work on another “job” while waiting for the water to heat. Interruptions and waiting are generally unavoidable. The act of carrying out the recipe instructions to make coffee is similar to the process of executing program instructions.

A process is a program in execution. In some literature, you might come across the term job. Both terms refer to the same thing, although the term process has superseded the term job. Unlike a program, which is static, a process is a dynamic entity. It holds several key aspects, in particular:

- Program: The executable code related to the process
- Memory: Temporary data storage
- State: A process usually hops between different states. After it is in the New state, i.e., just created, it moves to the Ready state, i.e., ready to run once given CPU time. Once the CPU allocates time for it, it goes to the Running state. Furthermore, it can be in the Waiting state pending I/O or event completion. Once it exits, it moves to the Terminated state.

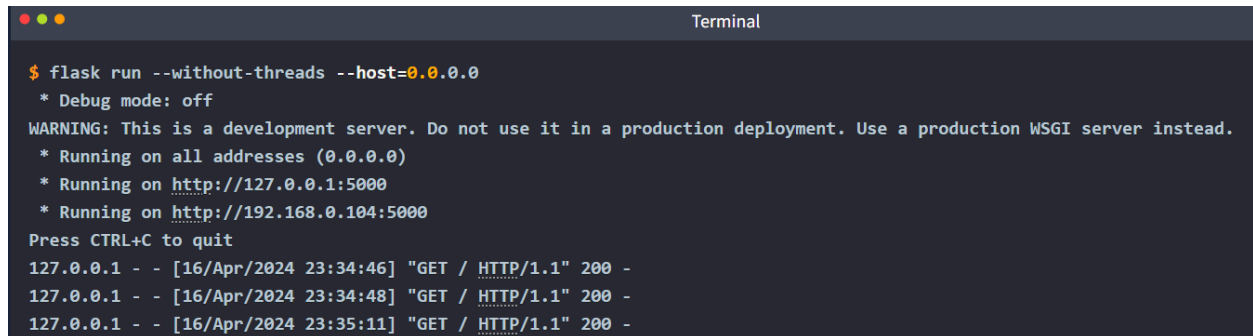


If you run the Flask code above, a process will be created, and it will listen for incoming connections at port 8080. In other words, it will spend most of its time in the Waiting state. When it receives an HTTP GET / request, it will switch to the Ready state, waiting

for its turn to run based on the CPU scheduling. Once in the Running state, it sends the HTML page to the client and returns to the Waiting state.

From the server's perspective, the app is servicing clients sequentially, i.e., client requests are processed one at a time. (Note that Flask is multi-threaded by default since version 1.0. We used the argument `--without-threads` to force it to run single-threaded.)

Please note that the Flask command shown below is for demonstrative purposes only.

A terminal window titled "Terminal" with a dark background. It shows the command `$ flask run --without-threads --host=0.0.0.0` and its output. The output includes a warning about using a development server, information about running on all addresses and specific ports, and a list of incoming HTTP requests.

```
$ flask run --without-threads --host=0.0.0.0
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://192.168.0.104:5000
Press CTRL+C to quit
127.0.0.1 - - [16/Apr/2024 23:34:46] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [16/Apr/2024 23:34:48] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [16/Apr/2024 23:35:11] "GET / HTTP/1.1" 200 -
```

Threads

Let's wrap up with another coffee analogy! Consider the case of a commercial espresso machine in a coffee shop. Let's say it has two portafilters. At the start of the work day, the barista turns on the espresso machine, and whenever a customer orders an espresso, one portafilter is used to prepare an espresso shot for them. Does another customer order an espresso? No problem, the second portafilter to the rescue! The warmed-up espresso machine is the process; each new order is assigned a portafilter; that's the analogy for the thread.

A thread is a lightweight unit of execution. It shares various memory parts and instructions with the process.

In many cases, we need to replicate the same process repeatedly. Think of a web server serving thousands of users the same page (or a personalized page). We can adopt one of two main approaches:

- Serial: One process is running; it serves one user after the other sequentially. New users are enqueued.
- Parallel: One process is running; it creates a thread to serve every new user. New users are only enqueued after the maximum number of running threads is reached.

The previous app can run with four threads using Gunicorn. Gunicorn, also called the “Green Unicorn”, is a Python WSGI HTTP server. WSGI stands for Web Server Gateway Interface, which bridges web servers and Python web applications. In particular, Gunicorn can spawn multiple worker processes to handle incoming requests simultaneously. By running gunicorn with the `--workers=4` option, we are specifying that we want four workers ready to tackle clients’ requests; moreover, `--threads=2` indicates that each worker process can spawn two threads.

Please note that the Gunicorn command shown below is for demonstrative purposes only. We didn’t provide an environment to run it as it is outside the scope of this room.

```
gunicorn --workers=4 --threads=2 -b 0.0.0.0:8080 app:app
[2024-04-16 23:35:59 +0300] [507149] [INFO] Starting gunicorn 21.2.0
[2024-04-16 23:35:59 +0300] [507149] [INFO] Listening at: http://0.0.0.0:8080 (507149)
[2024-04-16 23:35:59 +0300] [507149] [INFO] Using worker: gthread
[2024-04-16 23:35:59 +0300] [507150] [INFO] Booting worker with pid: 507150
[2024-04-16 23:35:59 +0300] [507151] [INFO] Booting worker with pid: 507151
[2024-04-16 23:35:59 +0300] [507152] [INFO] Booting worker with pid: 507152
[2024-04-16 23:35:59 +0300] [507153] [INFO] Booting worker with pid: 507153
```

It is worth noting the following:

- It is impossible to run more than one copy of this process as it binds itself to TCP port 8080. A TCP or UDP port can only be tied to one process.
- Process can be configured with any number of threads, and the HTTP requests arriving at port 8080 will be sent to the different threads.

Answer the questions below:

You downloaded an instruction booklet on how to make an origami crane. What would this instruction booklet resemble in computer terms?

Answer: **Program**

What is the name of the state where a process is waiting for an I/O event?

Answer: **Waiting**

Race Conditions

Real World Analogy

Picture the following situation. You call a restaurant to reserve a table for a crucial business lunch. You are familiar with the restaurant and its setup. One particular table, number 17, is your preferred choice, considering it has a nice view and is relatively isolated. You call to make a reservation for Table 17; the host confirms it is free as no “Reserved” tag is placed on it. At the same time, another customer is talking with another host and making a reservation for the same table.

Who really reserved the table? That’s a race condition.

Why did this happen? This unlucky situation happened because more than one host was taking reservations; furthermore, it took the host a few minutes to fetch the “Reserved” tag and put it on the table after updating the daily reservation book. There is at least a one-minute window for another client to reserve a reserved table.

Similarly, when one thread checks a value to perform an action, another thread might change that value before the action takes place.

Example A

Let’s consider this scenario:

- A bank account has \$100.
- Two threads try to withdraw money at the same time.
- Thread 1 checks the balance (sees \$100) and withdraws \$45.
- Before Thread 1 updates the balance, Thread 2 also checks the balance (incorrectly sees \$100) and withdraws \$35.

We cannot be 100% certain which thread will get to update the remaining balance first; however, let’s assume that it is Thread 1. Thread 1 will set the remaining balance to \$55. Afterwards, Thread 2 might set the remaining balance to \$65 if not appropriately handled. (Thread 2 calculated that \$65 should remain in the account after the withdrawal because the balance was \$100 when Thread 2 checked it.) In other words, the user made two withdrawals, but the account balance was deducted only for the second one because Thread 2 said so!

Example B

Let’s consider another scenario:

- A bank account has \$75.
- Two threads try to withdraw money at the same time.
- Thread 1 checks the balance (sees \$75) and withdraws \$50.
- Before Thread 1 updates the balance, Thread 2 checks the balance (incorrectly sees \$75) and withdraws \$50.

Thread 2 will proceed with the withdrawal, although such a transaction should have been declined.

Examples A and B demonstrate a Time-of-Check to Time-of-Use (TOCTOU) vulnerability.

Example Code

Consider the following Python code with two threads simulating a task completion by 10% increments.

```
import threading

x = 0 # Shared variable

def increase_by_10():
    global x
    for i in range(1, 11):
        x += 1
        print(f"Thread {threading.current_thread().name}: {i}0% complete, x = {x}")

# Create two threads
thread1 = threading.Thread(target=increase_by_10, name="Thread-1")
thread2 = threading.Thread(target=increase_by_10, name="Thread-2")

# Start the threads
thread1.start()
thread2.start()

# Wait for both threads to finish
thread1.join()
thread2.join()

print("Both threads have finished completely.")
```

These two threads start together; they do nothing except print a value on the screen. Consequently, one would expect them to finish simultaneously, or at least the result to be consistent. However, in the program above, there is no guarantee which thread will finish first and how early it will be. Below is the first execution output:

```
python t3_race_to_100.py
...
Thread Thread-1: 40% complete, x = 10
Thread Thread-2: 70% complete, x = 11
Thread Thread-1: 50% complete, x = 12
Thread Thread-2: 80% complete, x = 13
Thread Thread-1: 60% complete, x = 14
Thread Thread-1: 70% complete, x = 16
Thread Thread-2: 90% complete, x = 15
Thread Thread-2: 100% complete, x = 17
Thread Thread-1: 80% complete, x = 18
Thread Thread-1: 90% complete, x = 19
Thread Thread-1: 100% complete, x = 20
Both threads have finished completely.
```

Below is a second execution output:


```
python t3_race_to_100.py
...
Thread Thread-1: 70% complete, x = 10
Thread Thread-2: 40% complete, x = 11
Thread Thread-1: 80% complete, x = 12
Thread Thread-2: 50% complete, x = 13
Thread Thread-1: 90% complete, x = 14
Thread Thread-2: 60% complete, x = 15
Thread Thread-1: 100% complete, x = 16
Thread Thread-2: 70% complete, x = 17
Thread Thread-2: 80% complete, x = 18
Thread Thread-2: 90% complete, x = 19
Thread Thread-2: 100% complete, x = 20
Both threads have finished completely.
```

Running this program multiple times will lead to different results. In the first attempt, Thread-2 reached 100 first; however, in the second attempt, Thread-2 reached 100 second. We have no control over the output. If the security of our application relies on one thread finishing before the other, then we need to set mechanisms in place to ensure proper protection. Consider the following two examples to better understand the bugs' gravity when we leave things to chance.

On the AttackBox, you can save the above Python code and run it multiple times to observe the outcome. For instance, if you saved it as race.py, you can run the script using the python race.py command.

Causes

As we saw in the last program, two threads were changing the same variable. Whenever the thread was given CPU time, it rushed to increase x by 1. Consequently, these two threads were "racing" to increment the same variable. This program shows a straightforward example happening on a single host.

Generally speaking, a common cause of race conditions lies in shared resources. For example, when multiple threads concurrently access and modify the same shared data.

Examples of shared data are a database record and an in-memory data structure.

There are many subtle causes, but we will mention three common ones:

- Parallel Execution: Web servers may execute multiple requests in parallel to handle concurrent user interactions. If these requests access and modify shared resources or application states without proper synchronization, it can lead to race conditions and unexpected behaviour.
- Database Operations: Concurrent database operations, such as read-modify-write sequences, can introduce race conditions. For example, two users attempting to update the same record simultaneously may result in inconsistent data or conflicts. The solution lies in enforcing proper locking mechanisms and transaction isolation.
- Third-Party Libraries and Services: Nowadays, web applications often integrate with third-party libraries, APIs, and other services. If these external components are not designed to handle concurrent access properly, race conditions may occur when multiple requests or operations interact with them simultaneously.

Answer the questions below:

**Does the presented Python script guarantee which thread will reach 100% first?
(Yea/Nay)**

Answer: **Nay**

In the second execution of the Python script, what is the name of the thread that reached 100% first?

Answer: **Thread-1**

Web Application Architecture

Let's visit web application architecture to explain how race conditions are possible.

Client-Server Model

Web applications follow a client-server model:

- Client: The client is the program or application that initiates the request for a service. For example, when we browse a web page, our web browser requests the web page (file) from a web server.
- Server: The server is the program or system that provides these services in response to incoming requests. For instance, the web server responds to an

incoming HTTP GET request and sends an HTML page (or file) to the requesting web browser (client).

Generally speaking, the client-server model runs over a network. The client sends its request over the network, and the server receives it and processes it before sending back the required resource.

Typical Web Application

A web application follows a multi-tier architecture. Such architecture separates the application logic into different layers or tiers. The most common design uses three tiers:

- Presentation tier: In web applications, this tier consists of the web browser on the client side. The web browser renders the HTML, CSS, and JavaScript code.
- Application tier: This tier contains the web application's business logic and functionality. It receives client requests, processes them, and interacts with the data tier. It is implemented using server-side programming languages such as Node.js and PHP, among many others.
- Data tier: This tier is responsible for storing and manipulating the application data. Typical database operations include creating, updating, deleting, and searching existing records. It is usually achieved using a database management system (DBMS); examples of DBMS include MySQL and PostgreSQL.

States

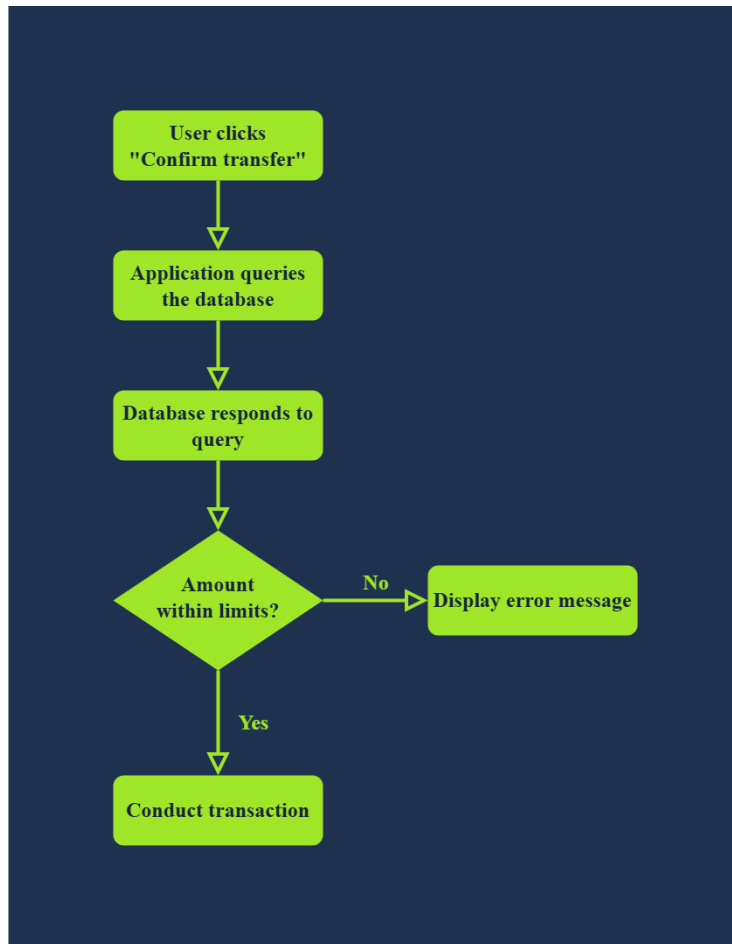
Let's visit some examples from business logic before diving deeper. We will consider the following examples:

- Validating and conducting money transfer
- Validating coupon codes and applying discounts

Validating and Conducting Money Transfer

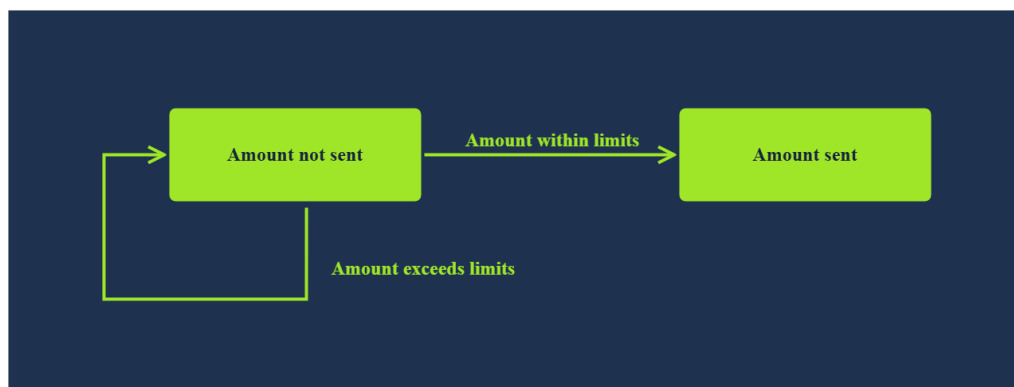
Consider the example of transferring money to a friend or your other account. The program will progress as follows:

- The user clicks on the "Confirm Transfer" button
- The application queries the database to confirm that the account balance can cover the transfer amount
- The database responds to the query
 - If the amount is within the account limits, the application conducts the transaction
 - If the amount is beyond the account limits, the application shows an error message



In an ideal scenario, the code above leads to two program states:

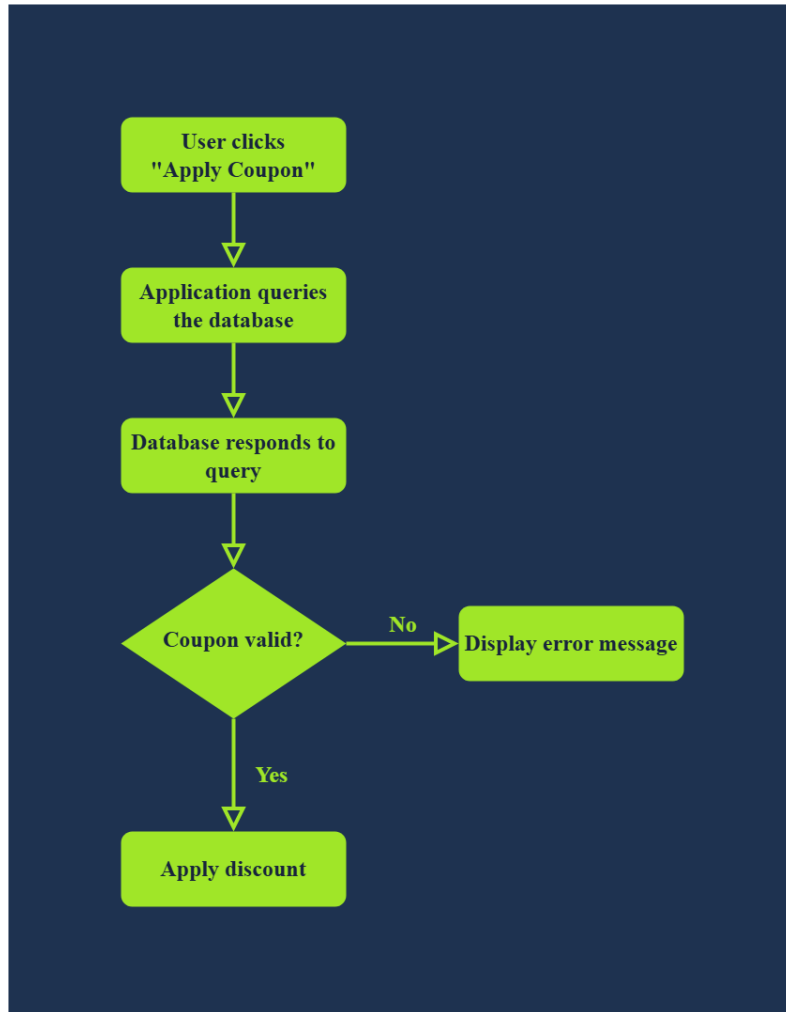
- Amount not sent
- Amount sent



Validating coupon codes and applying discounts

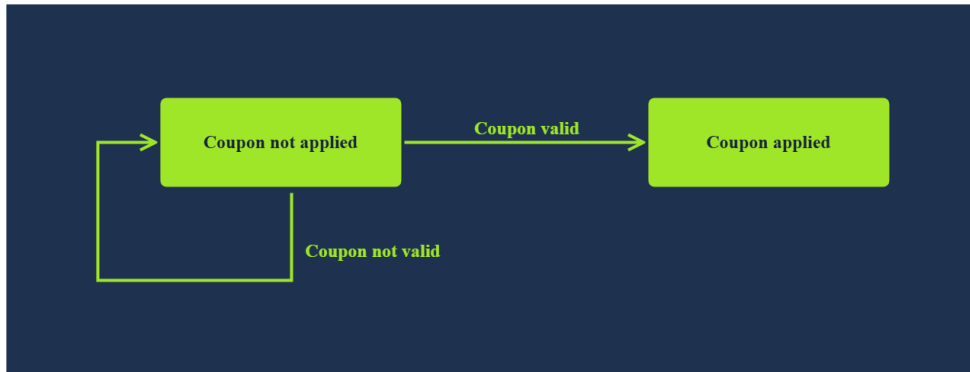
Let's consider the example of applying a discount coupon. The user goes to their shopping cart and adds a coupon to get a discount. The steps might be something along the following lines:

- The user enters a coupon code
- The application queries the database to determine whether the coupon code is valid and whether any constraints exist
- The database responds with validity and constraints
 - The discount is applied if the code is valid and there are no constraints on applying it for this user.
 - An error message is displayed if the code is invalid or there are constraints on applying it for this user.



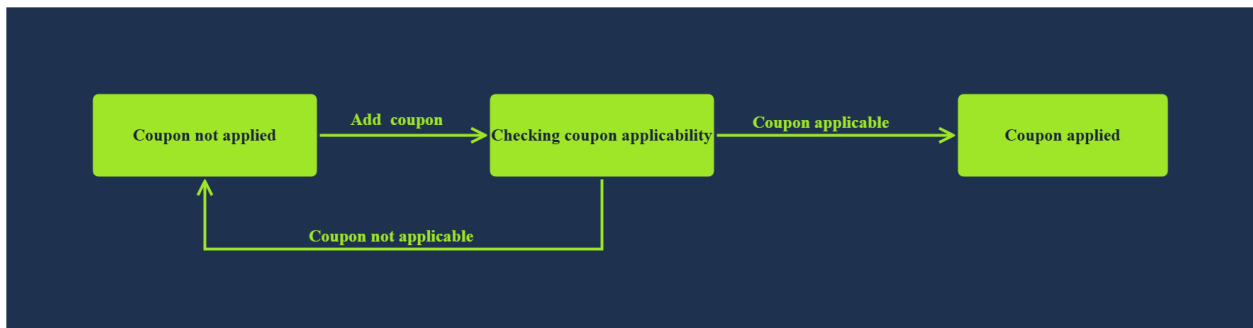
The above code leads to a few program states:

- Coupon not applied
- Coupon applied

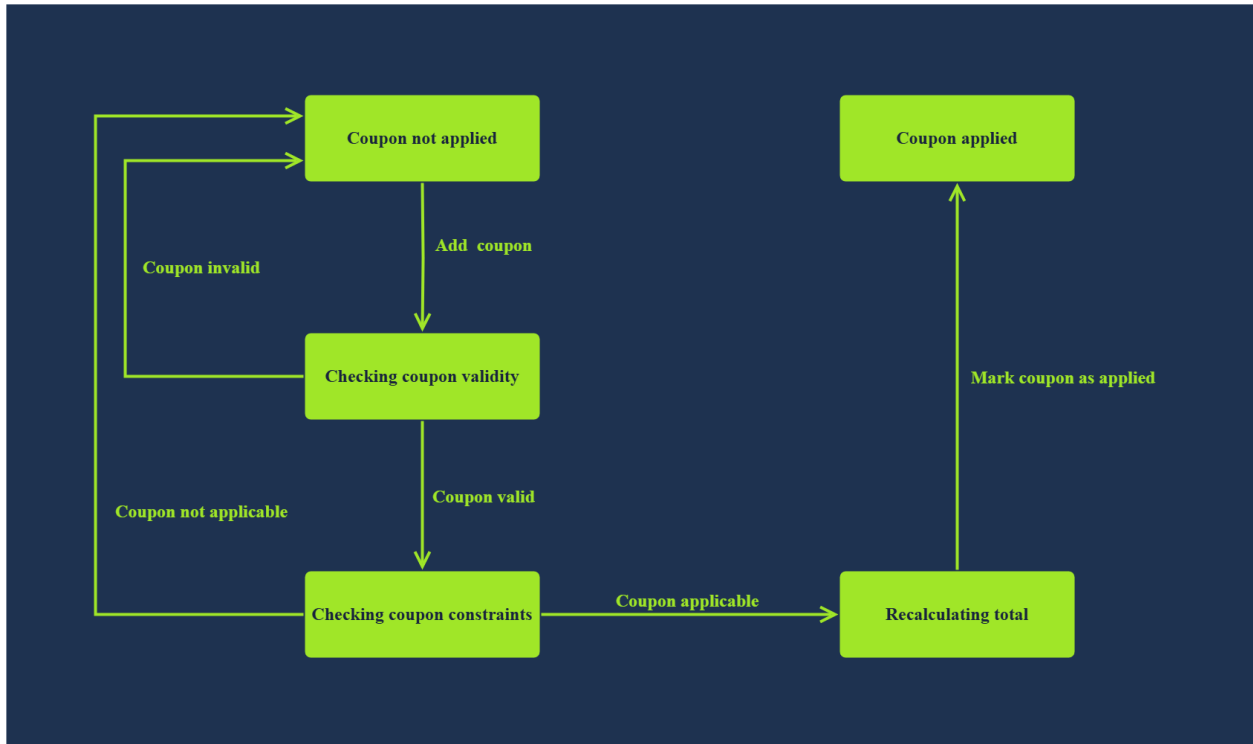


Two States? Think Again

Let's continue our analysis of applying a discount coupon. Ideally, we expect two states: Coupon not applied and Coupon applied. However, this is too simplistic to depict real sophisticated scenarios. We can add an intermediary state: Checking coupon applicability.

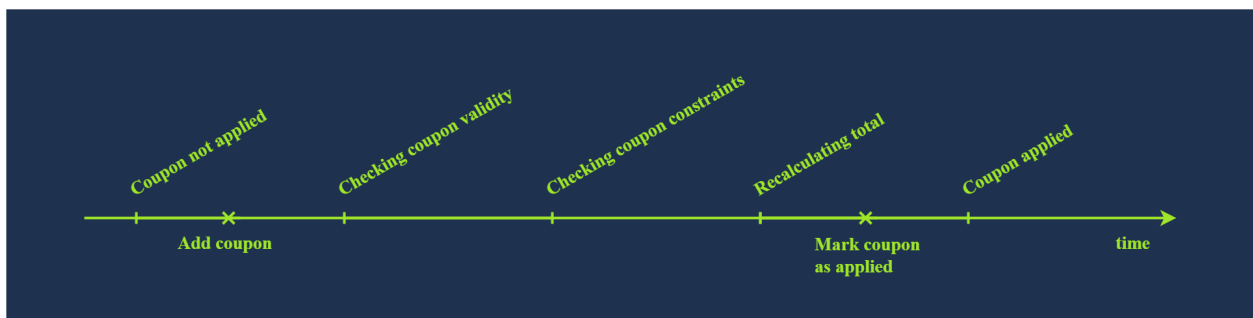


Depending on how the application is developed, we can expect more states. For example, Checking coupon applicability might involve two states: Checking coupon validity and Checking coupon constraints. A coupon might be valid, but existing constraints prevent it from being applied. Similarly, Coupon applied might be divided into two states, one of which is Recalculating total.



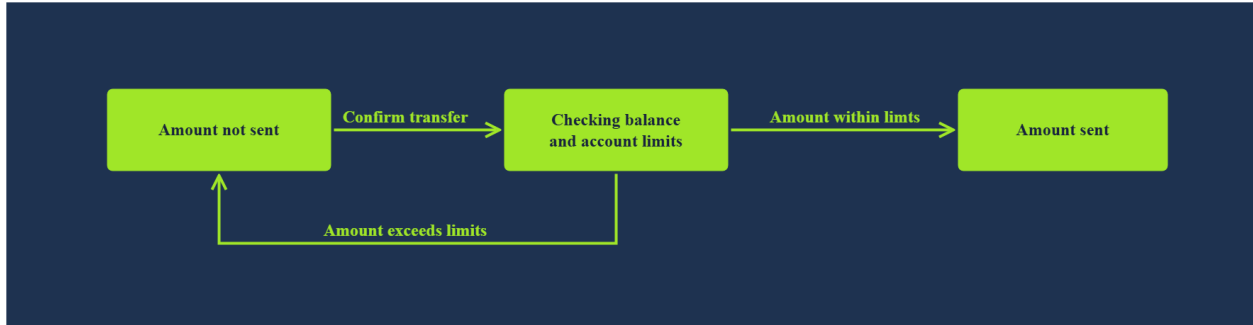
Why is this important for race conditions?

In the state diagram above, we can see that we pass through multiple states before the coupon is marked as applied. Let's draw the states on a time axis, as shown below.



There is a time window between the instant we try to add a coupon and the instant where the coupon is marked as applied and cannot be applied again. As long as the coupon is not marked as applied, most likely, no controls prevent it from being accepted repeatedly. We might be able to apply it multiple times during this time window.

This situation is similar when considering the states for the program making a money transfer. Although ideally speaking, it would be two states, considering the business logic, we can easily update the diagram to include three states. The reason is that we expect some time spent checking the account balance and limits; although this amount of time might be brief, it is not zero. If we dig deeper, we can uncover more "hidden" states.



However, even if the web application is vulnerable, we still have one challenge to overcome: timing. Even in vulnerable applications, this “window of opportunity” is relatively short; therefore, exploiting it necessitates that our requests reach the server simultaneously. In practice, we aim to get our repeated requests to reach the server only milliseconds apart.

How can we get our duplicated requests to reach the server within this short window? We need a tool such as Burp Suite.

Answer the questions below:

How many states did the original state diagram of “validating and conducting money transfer” have?

Answer: **2**

How many states did the updated state diagram of “validating and conducting money transfer” have?

Answer: **3**

How many states did the final state diagram of “validating coupon codes and applying discounts” have?

Answer: **5**

Exploiting Race Conditions

Click on the Start Machine button on the right to start the attached VM. Click on the Start AttackBox button at the top to start the AttackBox. On the AttackBox, browse to http://MACHINE_IP:8080.

These are the credentials for two users:

- User1: 07799991337
- Password: pass1234

And

- User2: 07113371111
- Password: pass1234

This web application belongs to a mobile operator and allows phone credit transfer. In this demo, we will check if the system is susceptible to a race condition vulnerability and try to exploit it by transferring more credit than we have in our account.

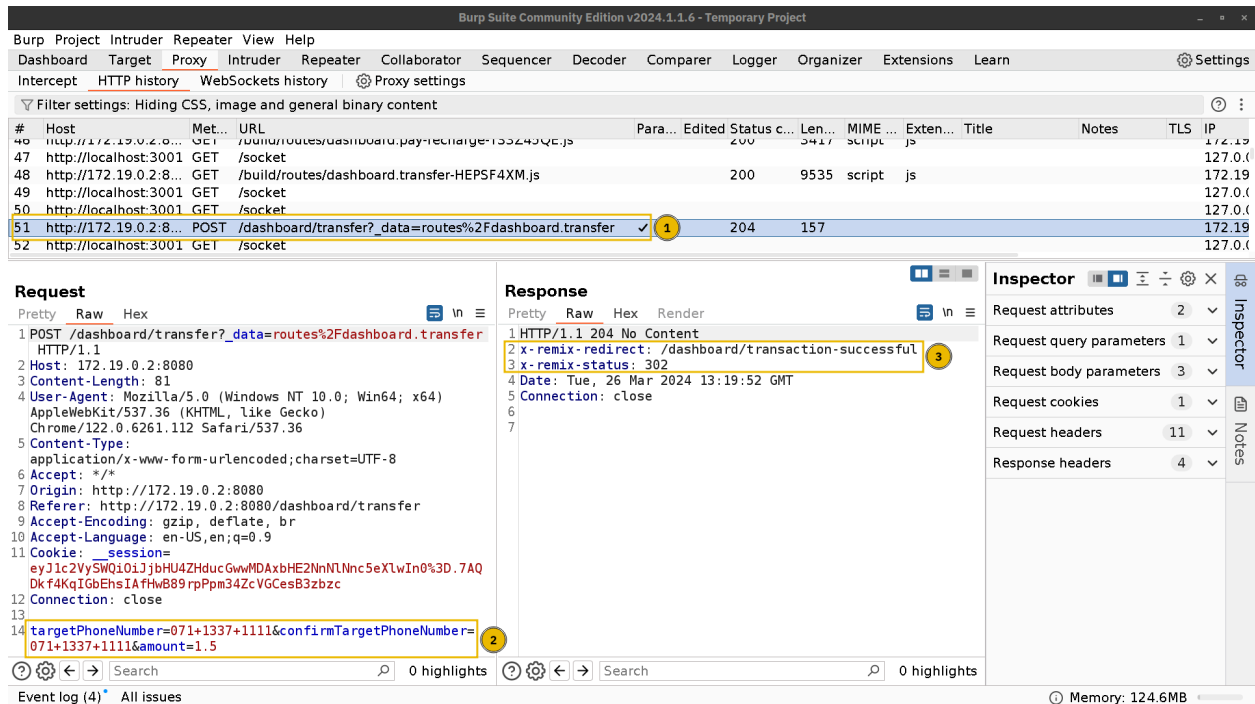
First, we need to explore and study how the target web application receives HTTP requests and how it responds to them. Using Burp Suite Proxy, click Open browser under the Intercept tab. (If you get an error message about enabling the browser's sandbox, you must manually change the settings. In this case, click Settings on the top right of your Burp Suite window, click on the Burp's browser under Tools, and check Allow Burp's browser to run without a sandbox.) Using the bundled browser, we can browse the target site and study how it processes our HTTP requests, notably the POST HTTP requests and related responses. The HTTP history tab logs every HTTP request and the respective response.

Log in to either of the accounts and click the Pay & Recharge button. Let's make a credit transfer: click the Transfer button and enter the mobile number of the other account along with the amount you want to transfer. You can try to transfer an amount that exceeds your current balance and a small amount, such as \$1, to see how the system responds in each case.

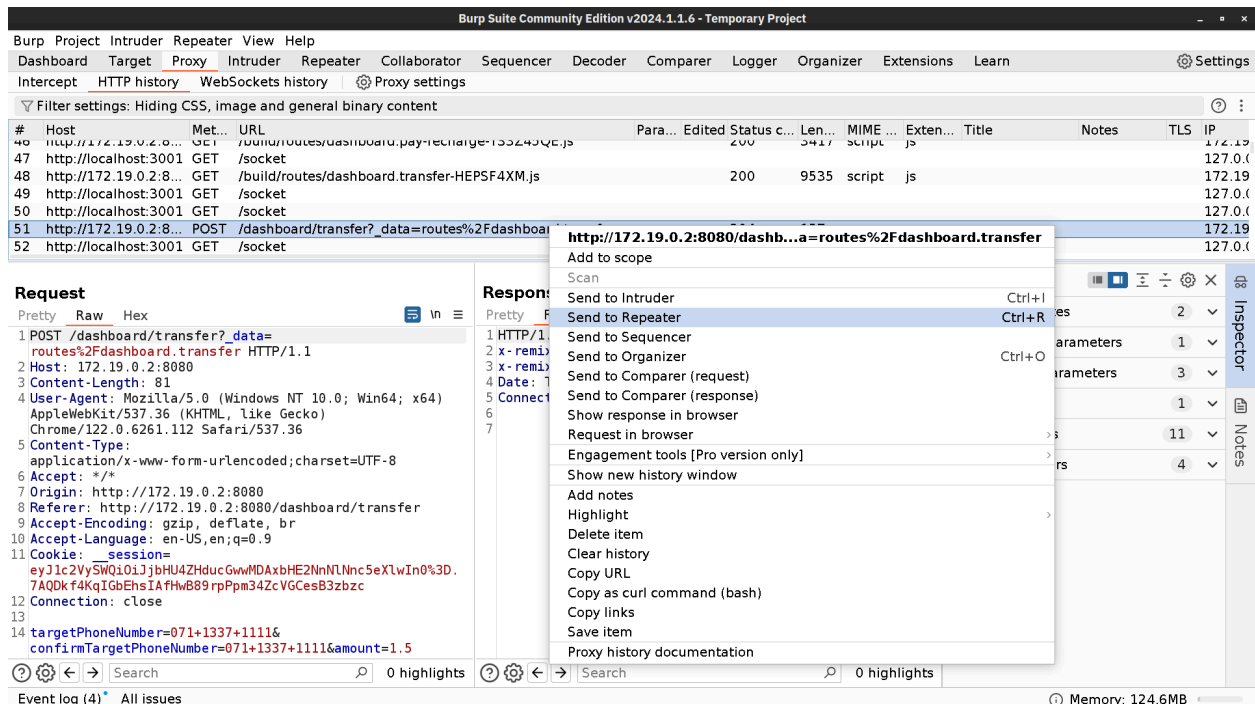
Burp Suite: Repeater

In the image below, we can see:

1. A POST request
2. The details show the target phone number and a transfer amount of \$1.5
3. In the response, we can infer that the transaction is successful



Now that we have seen how the system reacts to valid and invalid requests, let's see if we can exploit a race condition. Right-click on the POST request you want to duplicate and choose Send to Repeater.



In the Repeater tab, as shown in the numbered screenshots below:

1. Click on the + icon next to the received request tab and select Create tab group
2. Assign a group name, and include the tab of the request you just sent to the importer before clicking Create

3. Right-click on the request tab and choose Duplicate tab (If this option is not available in your version, you can press CTRL+R multiple times instead)
4. As a starting point, we will duplicate it 20 times
5. Next to the Send button, the arrow pointed downwards will bring a menu to decide how you want to send the duplicated requests

Next, we will exploit the target application by sending the duplicated request. Using the built-in options in Burp Suite Repeater, the drop-down arrow offers the following choices:

- Send group in sequence (single connection)
- Send group in sequence (separate connections)
- Send group in parallel

Sending Request Group in Sequence

Sending the group in sequence provides two options:

- Send group in sequence (single connection)
- Send group in sequence (separate connections)

Send Group in Sequence over a Single Connection

This option establishes a single connection to the server and sends all the requests in the group's tabs before closing the connection. This can be useful for testing for potential client-side desync vulnerabilities.

Send Group in Sequence over Separate Connections

As the name suggests, this option establishes a TCP connection, sends a request from the group, and closes the TCP connection before repeating the process for the subsequent request.

We tested this option to attack the web application. The screenshot below shows 21 TCP connections for the different POST requests in the group we sent.

- The first group (labelled 1) comprises five successful requests. We could confirm that they were successful by checking the respective responses. Furthermore, we noticed that each took around 3 seconds, as indicated by the duration (labelled 3).
- The second group (labelled 2) shows sixteen denied requests. The duration was around four milliseconds. It is interesting to check the Relative Start time as well.

Wireshark - Conversations - Send-group-sequence-separate-connection.pcapng

Conversation Settings

☐ Name resolution

☐ Absolute start time

☒ Limit to display filter

Copy

Follow Stream...

Graph...

Protocol

Address A

Port A

IPv4

TCP

21

UDP

192.168.124.8

36318

172.18.0.2

8080

10

2 kB

2

10

100.00%

5

1 kB

5

495 bytes

12.048695

3.0178

2.738 bits/s

192.168.124.8

36332

172.18.0.2

8080

10

2 kB

3

10

100.00%

5

1 kB

5

495 bytes

15.070648

3.0280

2.729 bits/s

192.168.124.8

49962

172.18.0.2

8080

10

2 kB

4

10

100.00%

5

1 kB

5

495 bytes

18.102499

3.0292

2.728 bits/s

192.168.124.8

49978

172.18.0.2

8080

10

2 kB

5

10

100.00%

5

1 kB

5

495 bytes

21.135373

3.0273

2.729 bits/s

192.168.124.8

49980

172.18.0.2

8080

10

2 kB

6

10

100.00%

5

1 kB

5

495 bytes

24.165785

3.0305

2.726 bits/s

192.168.124.8

49984

172.18.0.2

8080

10

2 kB

7

10

100.00%

5

1 kB

5

522 bytes

27.200325

0.0053

1.554 kbps

192.168.124.8

49990

172.18.0.2

8080

10

2 kB

8

10

100.00%

5

1 kB

5

522 bytes

27.210097

0.0047

192.168.124.8

50002

172.18.0.2

8080

10

2 kB

9

10

100.00%

5

1 kB

5

522 bytes

27.217852

0.0044

192.168.124.8

50016

172.18.0.2

8080

10

2 kB

10

10

100.00%

5

1 kB

5

522 bytes

27.225338

0.0039

192.168.124.8

50032

172.18.0.2

8080

10

2 kB

11

10

100.00%

5

1 kB

5

522 bytes

27.232535

0.0049

192.168.124.8

50046

172.18.0.2

8080

10

2 kB

12

10

100.00%

5

1 kB

5

522 bytes

27.240391

0.0043

192.168.124.8

50062

172.18.0.2

8080

10

2 kB

13

10

100.00%

5

1 kB

5

522 bytes

27.247532

0.0040

192.168.124.8

50064

172.18.0.2

8080

10

2 kB

14

10

100.00%

5

1 kB

5

522 bytes

27.254428

0.0046

192.168.124.8

50074

172.18.0.2

8080

10

2 kB

15

10

100.00%

5

1 kB

5

522 bytes

27.261638

0.0039

192.168.124.8

50086

172.18.0.2

8080

10

2 kB

16

10

100.00%

5

1 kB

5

522 bytes

27.268885

0.0042

192.168.124.8

50088

172.18.0.2

8080

10

2 kB

17

10

100.00%

5

1 kB

5

522 bytes

27.276484

0.0045

192.168.124.8

50090

172.18.0.2

8080

10

2 kB

18

10

100.00%

5

1 kB

5

522 bytes

27.353520

0.0035

192.168.124.8

50096

172.18.0.2

8080

10

2 kB

19

10

100.00%

5

1 kB

5

522 bytes

27.363537

0.0041

192.168.124.8

50100

172.18.0.2

8080

10

2 kB

20

10

100.00%

5

1 kB

5

522 bytes

27.369552

0.0042

192.168.124.8

50104

172.18.0.2

8080

10

2 kB

21

10

100.00%

5

1 kB

5

522 bytes

27.378191

0.0039

192.168.124.8

50112

172.18.0.2

8080

10

2 kB

22

10

100.00%

5

1 kB

5

522 bytes

27.391244

0.0042

The screenshot below shows the whole TCP connection for a request. We can confirm that the POST request was sent in a single packet.

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

tcp.stream eq 18

| No. | Time | Source | Destination | Protocol | Length | Info |
|-----|----------|---------------|---------------|----------|--------|--|
| 181 | 15.32665 | 192.168.124.8 | 172.18.0.2 | TCP | 74 | 50100 → 8080 [SYN] Seq=0 Win=32120 Len=0 MSS=1460 SACK_PERM TSval=1692094924 TSecr=0 WS=128 |
| 182 | 15.32698 | 172.18.0.2 | 192.168.124.8 | TCP | 74 | 8080 → 50100 [SYN, ACK] Seq=0 Ack=1 Win=31856 Len=0 MSS=1460 SACK_PERM TSval=144016941 TSecr=1692094924 WS=128 |
| 183 | 15.32100 | 192.168.124.8 | 172.18.0.2 | TCP | 66 | 50100 → 8080 [ACK] Seq=1 Ack=1 Win=32128 Len=0 TSval=1692094924 TSecr=144016941 |
| 184 | 15.32130 | 192.168.124.8 | 172.18.0.2 | HTTP | 761 | POST /dashboard/transfer?data=routes%2Fdashboard/transfer HTTP/1.1 (application/x-www-form-urlencoded) |
| 185 | 15.32134 | 172.18.0.2 | 192.168.124.8 | TCP | 66 | 8080 → 50100 [ACK] Seq=1 Ack=696 Win=31872 Len=0 TSval=144016941 TSecr=1692094924 |
| 186 | 15.32374 | 192.168.124.8 | 172.18.0.2 | HTTP | 250 | HTTP/1.1 204 No Content |
| 187 | 15.32376 | 192.168.124.8 | 172.18.0.2 | TCP | 66 | 50100 → 8080 [ACK] Seq=696 Ack=185 Win=32000 Len=0 TSval=1692094927 TSecr=144016944 |
| 188 | 15.32391 | 172.18.0.2 | 192.168.124.8 | TCP | 66 | 8080 → 50100 [FIN, ACK] Seq=185 Ack=696 Win=31872 Len=0 TSval=144016944 TSecr=1692094927 |
| 189 | 15.32498 | 192.168.124.8 | 172.18.0.2 | TCP | 66 | 50100 → 8080 [FIN, ACK] Seq=696 Ack=186 Win=32000 Len=0 TSval=1692094928 TSecr=144016944 |
| 190 | 15.32510 | 172.18.0.2 | 192.168.124.8 | TCP | 66 | 8080 → 50100 [ACK] Seq=186 Ack=697 Win=31872 Len=0 TSval=144016945 TSecr=1692094928 |

Send Request Group in Parallel

Choosing to send the group's requests in parallel would trigger the Repeater to send all the requests in the group at once. In this case, we notice the following, as shown in the screenshot below:

- In the Relative Start column, we notice that all 21 packets were sent within a window of 0.5 milliseconds (labelled 1).
- All 21 requests were successful; they resulted in a successful credit transfer. Each request took around 3.2 seconds to complete (labelled 2).

Wireshark - Conversations - Send-group-parallel.pcapng

Conversation Settings

☐ Name resolution

☐ Absolute start time

☒ Limit to display filter

Copy

Follow Stream...

Graph...

Protocol

| Ethernet 1 | IPv4 1 | IPv6 | TCP 21 | UDP | | | | | | | | | | | | | | | | |
|---------------|--------|--------|------------|--------|---------|-------|-----------|---------------|------------------|---------------|-------------|---------------|-------------|-----------|----------|--------------|--|--|--|--|
| Address A | | Port A | Address B | Port B | Packets | Bytes | Stream ID | Total Packets | Percent Filtered | Packets A → B | Bytes A → B | Packets B → A | Bytes B → A | Rel Start | Duration | Bits/s A → B | | | | |
| 192.168.124.8 | | 43900 | 172.18.0.2 | 8080 | 12 | 2 kB | 2 | 12 | 100.00% | 6 | 1 kB | 6 | 561 bytes | 3.216826 | 3.2033 | 2.747 bits/s | | | | |
| 192.168.124.8 | | 43902 | 172.18.0.2 | 8080 | 12 | 2 kB | 3 | 12 | 100.00% | 6 | 1 kB | 6 | 561 bytes | 3.216839 | 3.1894 | 2.759 bits/s | | | | |
| 192.168.124.8 | | 43916 | 172.18.0.2 | 8080 | 12 | 2 kB | 4 | 12 | 100.00% | 6 | 1 kB | 6 | 561 bytes | 3.216865 | 3.2178 | 2.734 bits/s | | | | |
| 192.168.124.8 | | 43920 | 172.18.0.2 | 8080 | 12 | 2 kB | 5 | 12 | 100.00% | 6 | 1 kB | 6 | 561 bytes | 3.216899 | 3.2532 | 2.705 bits/s | | | | |
| 192.168.124.8 | | 43934 | 172.18.0.2 | 8080 | 12 | 2 kB | 6 | 12 | 100.00% | 6 | 1 kB | 6 | 561 bytes | 3.216905 | 3.2318 | 2.722 bits/s | | | | |
| 192.168.124.8 | | 43940 | 172.18.0.2 | 8080 | 12 | 2 kB | 7 | 12 | 100.00% | 6 | 1 kB | 6 | 561 bytes | 3.216938 | 3.2387 | 2.717 bits/s | | | | |
| 192.168.124.8 | | 43948 | 172.18.0.2 | 8080 | 12 | 2 kB | 8 | 12 | 100.00% | 6 | 1 kB | 6 | 561 bytes | 3.216947 | 3.1267 | 2.814 bits/s | | | | |
| 192.168.124.8 | | 43958 | 172.18.0.2 | 8080 | 12 | 2 kB | 9 | 12 | 100.00% | 6 | 1 kB | 6 | 561 bytes | 3.216969 | 3.1331 | 2.808 bits/s | | | | |
| 192.168.124.8 | | 43970 | 172.18.0.2 | 8080 | 12 | 2 kB | 10 | 12 | 100.00% | 6 | 1 kB | 6 | 561 bytes | 3.216991 | 3.1751 | 2.771 bits/s | | | | |
| 192.168.124.8 | | 43986 | 172.18.0.2 | 8080 | 12 | 2 kB | 11 | 12 | 100.00% | 6 | 1 kB | 6 | 561 bytes | 3.216999 | 3.1477 | 2.795 bits/s | | | | |
| 192.168.124.8 | | 44000 | 172.18.0.2 | 8080 | 12 | 2 kB | 12 | 12 | 100.00% | 6 | 1 kB | 6 | 561 bytes | 3.217031 | 3.1678 | 2.777 bits/s | | | | |
| 192.168.124.8 | | 44006 | 172.18.0.2 | 8080 | 12 | 2 kB | 13 | 12 | 100.00% | 6 | 1 kB | 6 | 561 bytes | 3.217045 | 3.2108 | 2.740 bits/s | | | | |
| 192.168.124.8 | | 44018 | 172.18.0.2 | 8080 | 12 | 2 kB | 14 | 12 | 100.00% | 6 | 1 kB | 6 | 561 bytes | 3.217069 | 3.1396 | 2.802 bits/s | | | | |
| 192.168.124.8 | | 44028 | 172.18.0.2 | 8080 | 12 | 2 kB | 15 | 12 | 100.00% | 6 | 1 kB | 6 | 561 bytes | 3.217092 | 3.2245 | 2.729 bits/s | | | | |
| 192.168.124.8 | | 44040 | 172.18.0.2 | 8080 | 12 | 2 kB | 16 | 12 | 100.00% | 6 | 1 kB | 6 | 561 bytes | 3.217104 | 3.1543 | 2.789 bits/s | | | | |
| 192.168.124.8 | | 44054 | 172.18.0.2 | 8080 | 12 | 2 kB | 17 | 12 | 100.00% | 6 | 1 kB | 6 | 561 bytes | 3.217136 | 3.2460 | 2.711 bits/s | | | | |
| 192.168.124.8 | | 44064 | 172.18.0.2 | 8080 | 12 | 2 kB | 18 | 12 | 100.00% | 6 | 1 kB | 6 | 561 bytes | 3.217140 | 3.2592 | 2.700 bits/s | | | | |
| 192.168.124.8 | | 44072 | 172.18.0.2 | 8080 | 12 | 2 kB | 19 | 12 | 100.00% | 6 | 1 kB | 6 | 561 bytes | 3.217170 | 3.1607 | 2.784 bits/s | | | | |
| 192.168.124.8 | | 44078 | 172.18.0.2 | 8080 | 12 | 2 kB | 20 | 12 | 100.00% | 6 | 1 kB | 6 | 561 bytes | 3.217195 | 3.2659 | 2.694 bits/s | | | | |
| 192.168.124.8 | | 44080 | 172.18.0.2 | 8080 | 12 | 2 kB | 21 | 12 | 100.00% | 6 | 1 kB | 6 | 561 bytes | 3.217208 | 3.1956 | 2.753 bits/s | | | | |
| 192.168.124.8 | | 44090 | 172.18.0.2 | 8080 | 12 | 2 kB | 22 | 12 | 100.00% | 6 | 1 kB | 6 | 561 bytes | 3.217371 | 3.1820 | 2.765 bits/s | | | | |

By paying close attention to the screenshot above, we notice that each request led to 12 packets; however, in the previous attempt (send in sequence), we see that each request required only 10 packets. Why did this happen?

According to Sending Grouped HTTP Requests documentation, when sending in parallel, Repeater implements different techniques to synchronize the requests' arrival at the target, i.e., they arrive within a short time frame. The synchronization technique depends on the HTTP protocol being used:

- In the case of HTTP/2+, the Repeater tries to send the whole group in a single packet. In other words, a single TCP packet would carry multiple requests.
- In the case of HTTP/1, the Repeater resorts to last-byte synchronization. This trick is achieved by withholding the last byte from each request. Only once all packets are sent without the last-byte are the last-byte of all the requests sent. The screenshot below shows our POST request sent over two packets.

race-send-parallel.pcapng

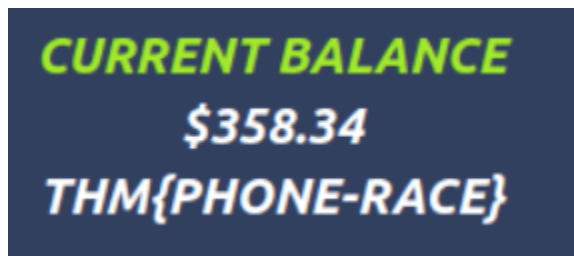
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

tcp.stream eq 18

| No. | Time | Source | Destination | Protocol | Length | Info |
|-----|----------|------------------------|-------------|----------|---|------|
| 19 | 0.000369 | 192.168... 172.18.0... | TCP | 74 | 44078 → 8080 [SYN] Seq=0 Win=32120 Len=0 MSS=1460 SACK_PERM TSval=1692642813 TSecr=0 WS=128 | |
| 67 | 0.001169 | 172.18.0... 192.168... | TCP | 74 | 8080 → 44078 [SYN, ACK] Seq=0 Ack=1 Win=31856 Len=0 MSS=1460 SACK_PERM TSval=144564827 TSecr=1692642813 WS=128 | |
| 74 | 0.001177 | 192.168... 172.18.0... | TCP | 66 | 44078 → 8080 [ACK] Seq=1 Ack=1 Win=32128 Len=0 TSval=1692642814 TSecr=144564827 | |
| 104 | 0.001570 | 192.168... 172.18.0... | TCP | 74 | 8080 → 44078 [PSH, ACK] Seq=1 Ack=1 Win=32128 Len=695 TSval=1692642814 TSecr=144564827 [TCP segment of a reassembled PDU] | |
| 105 | 0.001544 | 172.18.0... 192.168... | TCP | 66 | 8080 → 44078 [ACK] Seq=1 Ack=696 Win=31872 Len=0 TSval=144564827 TSecr=1692642814 | |
| 140 | 0.103117 | 192.168... 172.18.0... | HTTP | 67 | POST /dashboard/transfer?data=routes%2Fdashboard.transfer HTTP/1.1 (application/x-www-form-urlencoded) | |
| 147 | 0.103341 | 172.18.0... 192.168... | TCP | 66 | 8080 → 44078 [ACK] Seq=1 Ack=697 Win=31872 Len=0 TSval=144564929 TSecr=1692642916 | |
| 248 | 3.265787 | 172.18.0... 192.168... | HTTP | 223 | HTTP/1.1 204 No Content | |
| 249 | 3.265835 | 192.168... 172.18.0... | TCP | 66 | 44078 → 8080 [ACK] Seq=697 Ack=158 Win=32088 Len=0 TSval=1692646078 TSecr=144568091 | |
| 250 | 3.266039 | 172.18.0... 192.168... | TCP | 66 | 8080 → 44078 [FIN, ACK] Seq=158 Ack=697 Win=31872 Len=0 TSval=144568092 TSecr=1692646078 | |
| 251 | 3.266214 | 192.168... 172.18.0... | TCP | 66 | 44078 → 8080 [FIN, ACK] Seq=697 Ack=159 Win=32088 Len=0 TSval=1692646079 TSecr=144568092 | |
| 252 | 3.266287 | 172.18.0... 192.168... | TCP | 66 | 8080 → 44078 [ACK] Seq=159 Ack=698 Win=31872 Len=0 TSval=144568092 TSecr=1692646079 | |

Answer the questions below:

You need to get either of the accounts to get more than \$100 of credit to get the flag. What is the flag that you obtained?



Answer: **THM{PHONE-RACE}**

Detection and Mitigation

Detection

Detecting race conditions from the business owner's perspective can be challenging. If a few users redeemed the same gift card multiple times, it would most likely go unnoticed unless the logs are actively checked for certain behaviours. Considering that race conditions can be used to exploit even more subtle vulnerabilities, it is clear that

we need the help of penetration testers and bug bounty hunters to try to discover such vulnerabilities and report their findings.

Penetration testers must understand how the system behaves under normal conditions when enforced controls are enforced. The controls can be: use once, vote once, rate once, limit to balance, and limit to one every 5 minutes, among others. The next step would be to try to circumvent this limit by exploiting race conditions. Figuring out the different system's states can help us make educated guesses about time windows where a race condition can be exploited. Tools such as Burp Suite Repeater can be a great starting point.

Mitigation

We will list a few mitigation techniques.

- Synchronization Mechanisms: Modern programming languages provide synchronization mechanisms like locks. Only one thread can acquire the lock at a time, preventing others from accessing the shared resource until it's released.
- Atomic Operations: Atomic operations refer to indivisible execution units, a set of instructions grouped together and executed without interruption. This approach guarantees that an operation can finish without being interrupted by another thread.
- Database Transactions: Transactions group multiple database operations into one unit. Consequently, all operations within the transaction either succeed as a group or fail as a group. This approach ensures data consistency and prevents race conditions from multiple processes modifying the database concurrently.

Answer the questions below:

Make sure you have taken note of the above.

No Answer Needed

Challenge Web App

This room introduced race conditions and various situations leading to such vulnerabilities. System complexity and geographical spread can lead to diverse unforeseen situations, including vulnerabilities related to race conditions. To discover and exploit such conditions, it is vital that we first observe how the system behaves under normal conditions and then try to find out how it behaves when we try to exploit the timing. With the currently available tools, we have plenty of techniques to try.

Challenge

Following what you have learned, it is time to attempt discovering and exploiting a race condition without guidance.

Click on the Start Machine button on the right to start the attached VM. Click on the Start AttackBox button at the top to start the AttackBox if you haven't done that already. On the AttackBox, browse to `http://MACHINE_IP:5000`.

These are the credentials for the three users:

Name: Rasser Cond

- Username: 4621
- Password: blueApple

Name: Zavodni Stav

- Username: 6282
- Password: whiteHorse

Name: Warunki Wycigu

- Username: 9317
- Password: greenOrange

This web application belongs to a bank and allows clients to transfer online money. You need to get one of the accounts to amass more than \$1000.

Answer the questions below:

What flag did you obtain after getting an account's balance above \$1000?

Log in to one of the accounts and send a transfer to another account.

Transfer Funds

You send

100

5.00 USD

Total fees

Warunki Wycigu gets

95.00

Continue

Capture the transfer in Burp.

The screenshot shows the Burp Suite interface with the 'Intercept on' button active. Below the toolbar, a table lists captured requests:

| Time | Type | Direction | Method | URL |
|-------------|------|-----------|--------|---|
| 05:54:51.31 | HTTP | → Request | GET | http://ciscobinary.openh264.org/openh264-linux64-652bdb7719f30b52b08e506645a7322ff1b2cc6f.zip |
| 05:55:06.31 | HTTP | → Request | POST | http://10.10.11.252:5000/transfer/9317 |

The selected request is expanded, showing the following details:

Request

Pretty Raw Hex

3 User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:131.0) Gecko/20100101 Firefox/131.0

4 Accept: */*

5 Accept-Language: en-US,en;q=0.5

6 Accept-Encoding: gzip, deflate, br

7 Referer: http://10.10.11.252:5000/transfer/9317

8 Content-Type: multipart/form-data; boundary=-----179438609514922351491366395761

9 Content-Length: 443

10 Origin: http://10.10.11.252:5000

11 Connection: keep-alive

12 Cookie: session=.eJwZLskhDAQRuFXWf46iAhEyBPYb7fdqCMGdCLJxBTiuYus5Sn0d4KSLizqRLKeYDUkNkiRg5P2w54YaCUZGbZu66I0ENqewI80P4n7fJU0G0wUY_bhEZAXp9z7EBL_GT2aqqtwXTcC6Sa1.aIr26g.zBy97Emhqv_RuVaYjGuzFpZyCAI

13 Priority: u=0

14

15 -----179438609514922351491366395761

16 Content-Disposition: form-data; name="fund_being_transferred"

17

18 100

19 -----179438609514922351491366395761

20 Content-Disposition: form-data; name="calculatedfee"

21

22 5.00

23 -----179438609514922351491366395761

24 Content-Disposition: form-data; name="receiver_amount"

25

26 95.00

Send the transfer to the repeater and duplicate the tabs to equal \$1000 or more.

Group 5 21 x 107 x 108 x 109 x 110 x 111 x 112 x 113 x 114 x 115 x 116 x 117 x 118 x 119 x 120 x 121 x 122 x 123 x 124 x 125 x 126 x

127 x

Send group (parallel) Cancel < >

Target: http://10.10.11.252:5000

Request

Pretty Raw Hex

```
1 POST /transfer/9317 HTTP/1.1
2 Host: 10.10.11.252:5000
3 User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:131.0) Gecko/20100101
  Firefox/131.0
4 Accept: */*
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate, br
7 Referer: http://10.10.11.252:5000/transfer/9317
8 Content-Type: multipart/form-data;
  boundary=-----179438609514922351491366395761
9 Content-Length: 449
10 Origin: http://10.10.11.252:5000
11 Connection: keep-alive
12 Cookie: session=
  .eJvtzLskhDAQRuFWf461AheyBPYb7fdqCMGdCLJx8T1uyusS9n0d4KSLizqRlKeyDUKk1Rg5P2vS
  4YaCUZG8Zu6G1OGNqew180P4n7fJUOG0wUY_bhEZAhp9z7EB1_6T2aqqtwKtc6Sai.aIr26g.zBy97
  Emhqv_RuVaYjGuzFpZyCAI
13 Priority: u=0
14
15 -----179438609514922351491366395761
16 Content-Disposition: form-data; name="fund_being_transferred"
17
18 100
19 -----179438609514922351491366395761
20 Content-Disposition: form-data; name="calculatedfee"
21
22 5.00
23 -----179438609514922351491366395761
```

Response

Inspector

Request attributes 2

Request query parameters 0


Request body parameters 3

Request cookies 1

Request headers 12

Ready

Send the group in parallel and forward through the interceptor.

 Dashboard

Warunki Wycigu

Balance

\$3167.5 USD

THM{BANK-RED-FLAG}

Answer: THM{BANK-RED-FLAG}