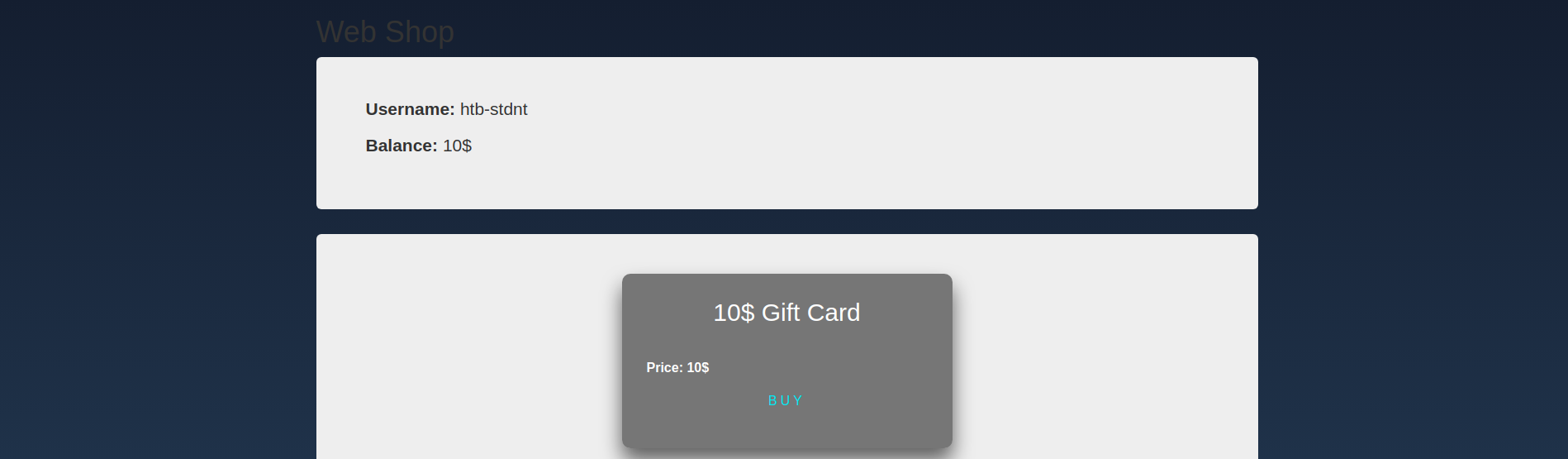
# Race Conditions

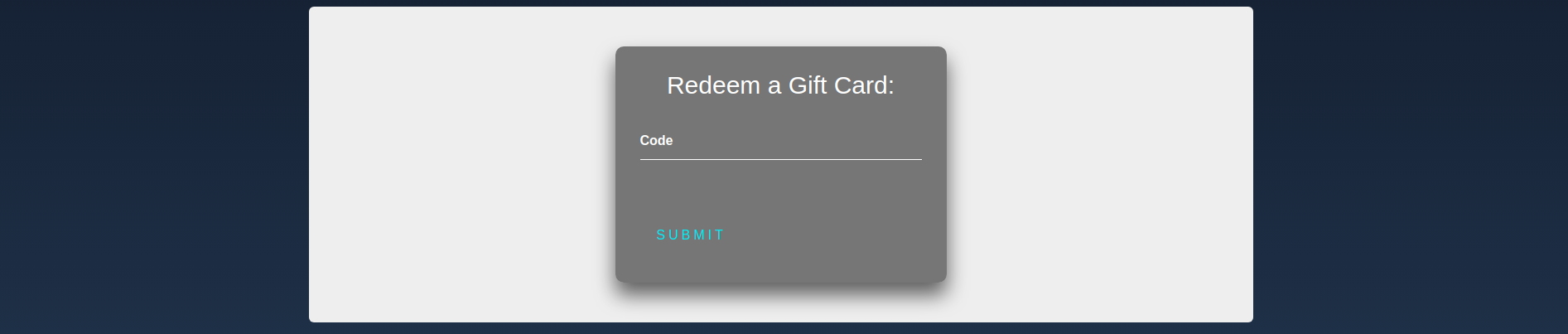
Race conditions in web applications arise when the developers do not account for the simultaneous execution of certain control paths due to multithreading. In particular, this also includes single-threaded languages like PHP if the web server itself supports multithreading. Since many web servers spawn multiple worker threads by default, the prerequisites are met for most default web server configurations. Let us discuss how we can identify race conditions, how we can exploit them, and how we can prevent them.

## Code Review - Identifying the Vulnerability

For this section, we will analyze the source code of a simple webshop for race condition vulnerabilities. Since the source code is more complex than the last few sections, let us start by getting an overview of the web application. After logging in, we are greeted with a simple webshop with our initial balance of 10$:



Further down, there is a form to redeem gift card codes to increase our balance:



Since this directly influences our balance, let us investigate how gift card codes are implemented. Redeeming a code results in the following request:

Code: http

POST /shop.php HTTP/1.1  
Host: racecondition.htb  
Content-Length: 23  
Content-Type: application/x-www-form-urlencoded  
Cookie: PHPSESSID=qvvchpk8h4qnotbniqqffd1nuv  
  
redeem=7204884880747967

The PHP code calls the function redeem\_gift\_card with the code provided in the redeem POST parameter and our username taken from the session variable, which looks like this:

Code: php

function redeem\_gift\_card($username, $code) {  
 $gift\_card\_balance = check\_gift\_card\_balance($code);  
  
 if ($gift\_card\_balance === 0) {  
 return "Invalid Gift Card Code!";  
 }  
  
 // update user balance  
 $user = fetch\_user\_data($username);  
 $new\_balance = $user['balance'] + $gift\_card\_balance;  
 update\_user\_balance($username, $new\_balance);  
  
 // invalidate code  
 invalidate\_gift\_card($code);  
  
 return "Successfully redeemed gift card. Your new balance is: " . $new\_balance . '$';  
}

At a high-level abstraction, the function works as follows:

* Check if the code is valid and fetch the balance from the database
* Return if the code is invalid
* Fetch the user's current balance from the database and add the gift card's balance
* Update the user's balance
* Invalidate the code

The code assumes synchronous actions since there are no locks or other mechanisms that would prevent race conditions. To illustrate this, let us consider what happens if the same HTTP request redeeming the code is sent two times in quick succession and different web server threads handle these requests. The two threads will simultaneously execute the redeem\_gift\_card function with the same code. If both threads pass the check\_gift\_card\_balance function before the other thread invalidates the code and one of the threads fetches the user's balance after the other thread has already updated the user's balance, the same gift card code will be applied twice, such that the balance is increased twice with the same code. This is a classical TOCTOU scenario since the gift card balance is checked before it is used (invalidated).

To illustrate this further, have a look at the following sequence of events consisting of the important steps of the redeem\_gift\_card function for both threads for a 10$ gift card:

|  |  |  |
| --- | --- | --- |
| **Thread 1** | **Thread 2** | **User's Balance** |
| redeem\_gift\_card("htb-stdnt", 7204884880747967) | - | 0$ |
| check\_gift\_card\_balance(7204884880747967) | - | 0$ |
| fetch\_user\_data("htb-stdnt") | - | 0$ |
| update\_user\_balance("htb-stdnt", 10$) | - | 10$ |
| - | redeem\_gift\_card("htb-stdnt", 7204884880747967) | 10$ |
| - | check\_gift\_card\_balance(7204884880747967) | 10$ |
| - | fetch\_user\_data("htb-stdnt") | 10$ |
| - | update\_user\_balance("htb-stdnt", 20$) | 20$ |
| invalidate\_gift\_card(7204884880747967) | - | 20$ |
| - | invalidate\_gift\_card(7204884880747967) | 20$ |

Due to multithreading, the two functions are executed simultaneously, making the above sequence of events possible. The timing needs to be just right so the first thread does not invalidate the code when the second thread checks its validity. Thus, exploitation of race conditions may require many attempts to get the timing right. In this case, we can exploit the race condition to apply the same gift card code multiple times to increase our balance.

## Debugging the Application Locally

We need to run the web application on a multi-threaded server to test our assumption about the race condition vulnerability. Thus, we cannot use PHP's built-in single-threaded web server. For a simple deployment option, we can use Docker. Since the source code comes with a Dockerfile, we can simply build the docker container and subsequently run it using the following commands:

Race Conditions

yovecio@htb[/htb]$ docker build -t race\_condition .  
yovecio@htb[/htb]$ docker run -p 8000:80 race\_condition  
  
 \* Starting MySQL database server mysqld  
 ...done.  
 \* Starting Apache httpd web server apache2  
AH00558: apache2: Could not reliably determine the server's fully qualified domain name, using 172.17.0.2. Set the 'ServerName' directive globally to suppress this message  
 \*

Afterward, we can access the web application at <http://localhost:8000>. Before jumping straight into the exploitation of the race condition, we first need to discuss how PHP handles session files and applies file locks since that significantly influences our exploit.

## Exploitation

#### PHP Session Files and File Locks

Without going into too much detail, PHP stores the session information in session files on the web server's filesystem. As such, during the execution of PHP files, the web server needs to read and write to these files. To ensure that no undefined or unsafe state is reached, PHP uses file locks on session files whenever the [session\_start](https://www.php.net/manual/en/function.session-start.php) function is used to prevent multiple file writes at the same time due to multithreading. File locks are implemented on the operating system level, ensuring that only a single thread can access the file at any time. If a second thread attempts to access a file while another file holds the file lock, the second process has to wait until the first thread is finished. Thus, simultaneous file accesses are prevented. These file locks are held until the end of the PHP file, i.e., until the response is sent or until the [session\_write\_close](https://www.php.net/manual/en/function.session-write-close.php) function is called.

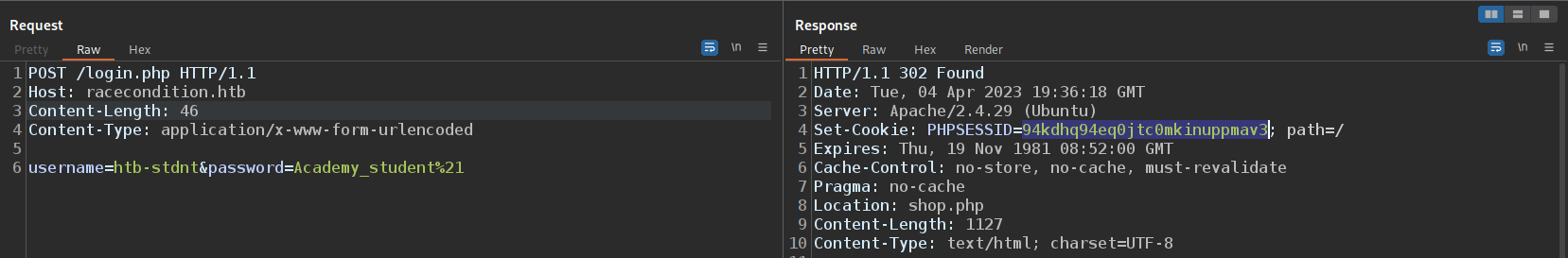
Therefore, these file locks indirectly prevent the exploitation of race conditions if session variables are used in the vulnerable PHP file. The race condition is only accessible after logging in, so session variables are used. If we attempt to send multiple requests using the same PHP session, the file locks will prevent simultaneous execution. Thus, threads must wait for the file locks before execution, effectively resulting in a single-thread scenario. This prevents any exploitation of a race condition vulnerability.

So how do we solve this problem? We can simply use different sessions in our exploit. Suppose we log in many times and record the session IDs. In that case, we can assign each request in our exploit different session IDs, meaning each thread accesses a different session file, and there is no need to wait for file locks, making simultaneous execution viable. Let us explore how to exploit the race condition above.

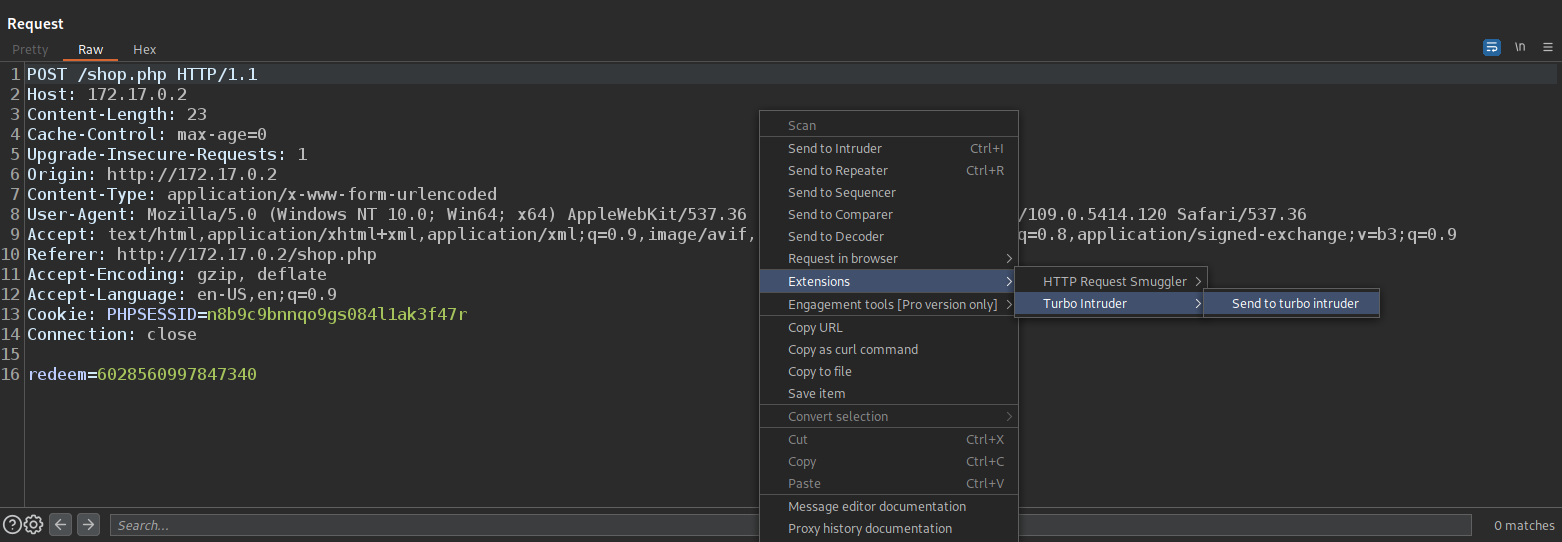
#### Burp Turbo Intruder

We will use the Burp extension Turbo Intruder to exploit the race condition. It can be installed in Burp by going to Extensions > BApp Store and installing the Turbo Intruder extension.

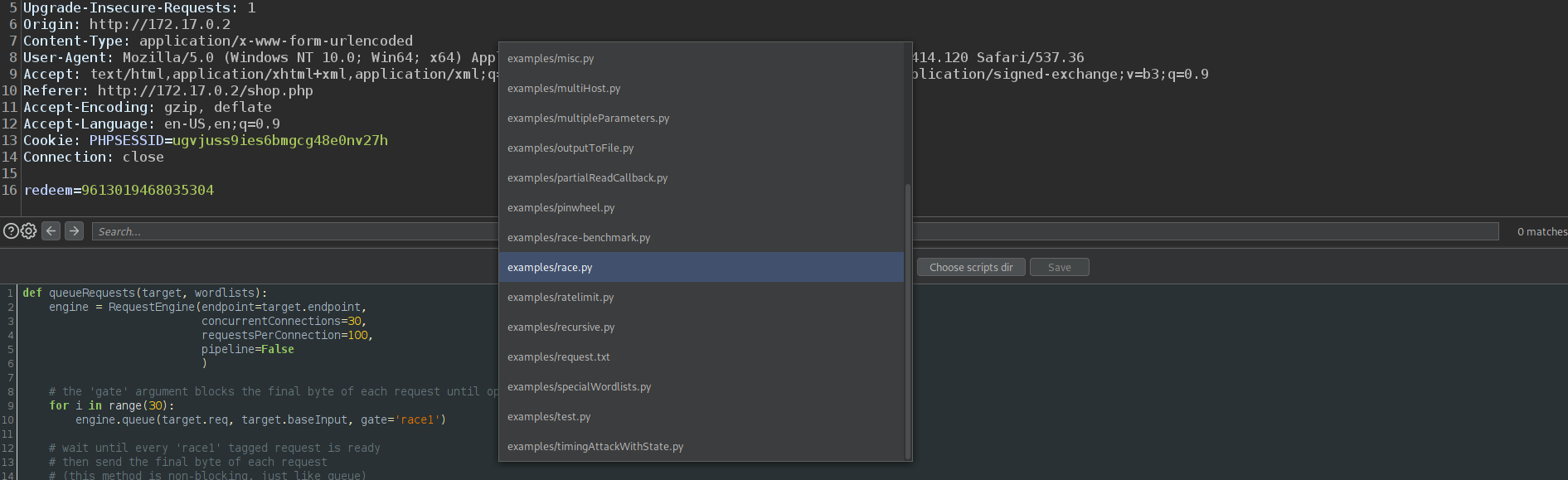
In the first step, we must generate multiple valid session IDs to avoid running into the file lock issue described above. To do so, we can send the login request to Burp Repeater, send it about 5 times, and take note of the five different PHPSESSID cookies:



To exploit the race condition, we will buy a gift card, intercept the request to redeem the code and drop it so it is not redeemed on the backend. We can then send the request to redeem the code to Turbo Intruder from Burp's HTTP history:



This opens the request in a Burp Turbo Intruder window. From the drop-down menu in the middle of the window, we will select the examples/race.py script:



**Note:** This script does not exist in the latest version of Turbo Intruder. If you are already familiar with Turbo Intruder, feel free to use any other script as a baseline and adjust it to your needs. Otherwise, you can find the race.py script in the Turbo Intruder GitHub repository [here](https://github.com/PortSwigger/turbo-intruder/blob/b5c6e2d614cf8db0e9b02a32dd06119161888e17/resources/examples/race.py). You can simply copy and paste it into the Turbo Intruder window and continue from there.

The turbo intruder window consists of two main parts: the HTTP request at the top and the exploit script at the bottom. The script at the bottom is written in Python, and we can modify it according to our needs. Turbo Intruder inserts a payload into the request wherever a %s is specified. In our case, we need to add different session cookies to the requests to avoid running into the file lock issue. Therefore, we will modify the request at the top by replacing the session cookie with the value %s such that the corresponding line looks like this:

Code: http

<SNIP>  
Cookie: PHPSESSID=%s  
<SNIP>

Here is the final request:

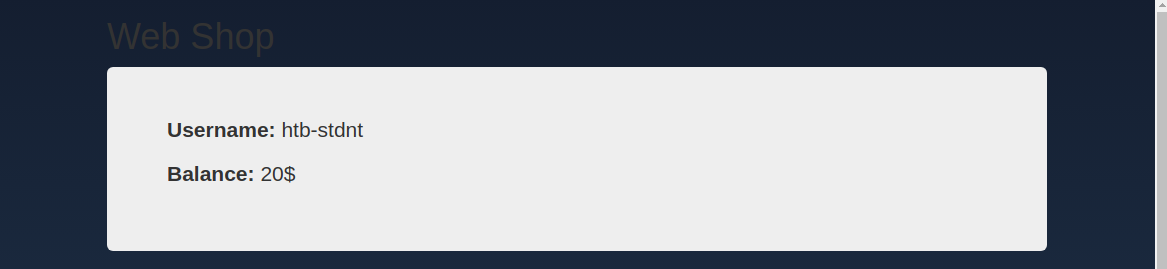


Now we have to specify the payload, which is the second parameter of the engine.queue function call. Thus, we modify the exploit script to look like this by inserting the valid session cookies we obtained in the previous step:

Code: python

def queueRequests(target, wordlists):  
 engine = RequestEngine(endpoint=target.endpoint,  
 concurrentConnections=30,  
 requestsPerConnection=100,  
 pipeline=False  
 )  
  
 # the 'gate' argument blocks the final byte of each request until openGate is invoked  
 for sess in ["p5b2nr48govua1ieljfdecppjg", "48ncr9hc1rjm361fp7h17110ar", "0411kdhfmca5uqiappmc3trgcg", "m3qv0d1qu7omrtm2rooivr7lc4", "onerh3j83jopd5ul8scjaf14rr"]:  
 engine.queue(target.req, sess, gate='race1')  
  
 # wait until every 'race1' tagged request is ready  
 # then send the final byte of each request  
 # (this method is non-blocking, just like queue)  
 engine.openGate('race1')  
  
 engine.complete(timeout=60)  
  
  
def handleResponse(req, interesting):  
 table.add(req)

Finally, we can start the attack by clicking the Attack button at the bottom of the Turbo Intruder window. After a few seconds, we can stop the attack and look at our balance by refreshing the browser window. If the attack was successful, we should have successfully redeemed the same code multiple times, increasing our balance by more than 10$:



Turbo Intruder is highly customizable since we can adjust the Python script according to our needs. For more details, check out the [Turbo Intruder documentation](https://portswigger.net/research/turbo-intruder-embracing-the-billion-request-attack). We can also use Turbo Intruder to send multiple different requests if the race condition involves different endpoints. The examples/test.py script is a good starting point to see how additional requests can be queued within the Python code. Alternatively, we can write our own custom Python script to exploit the race condition.

## Prevention & Patching

Now that we have seen how to exploit race condition vulnerabilities let us discuss how to prevent them. Since race conditions can arise in different contexts, prevention depends on the concrete vulnerability. For instance, if the race condition arises due to simultaneous file accesses, it can be prevented by implementing file locks similar to the PHP session file locks. In our case, the race condition exists because of simultaneous database accesses from multiple threads. To prevent this, we need to implement SQL locks. They work similarly to file locks. There are READ locks which allow the current session to read the table but not write to it. Other sessions are still allowed read access to the table but write access is prevented. Furthermore, there are WRITE locks that allow the current session read and write access to the table and prevent all access to the table by other sessions. Thus, our race condition can be prevented by obtaining a WRITE lock on the users table since the user's balance is updated and a WRITE lock on the active\_gift\_cards table since the gift card code is removed. We can achieve this by executing the following SQL query:

Code: sql

LOCK TABLES active\_gift\_cards WRITE, users WRITE;

After the code has been redeemed, we can release the locks by executing the following query:

Code: sql

UNLOCK TABLES;

This prevents simultaneous access to the database by multiple threads, thus preventing the race condition vulnerability. For more details, check out the SQL documentation on locks [here](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html).