**WEB RACE CONDITIONS**

Race conditions in web security refer to a type of vulnerability that occurs when multiple threads, processes, or operations in a computer program access shared resources or data at the same time, which can lead to a “collision” that causes unintended and malicious behavior in an application. These vulnerabilities can arise in a multitude of ways, especially in web applications, when multiple users or processes interact with the application's resources simultaneously.

Here's how a race condition might manifest in the context of web security:

1. **Authentication and Authorization**: Imagine a web application where a user's access rights are determined by a combination of their authentication status and their authorization level. If the application doesn't handle these checks properly and multiple requests from different users or processes arise simultaneously, there could be a race condition. For instance, a user might be able to access restricted content by timing their requests in a way that exploits the race condition.
2. **Account Balances**: In a banking application, if multiple transactions are processed simultaneously that affect an account's balance, such as deposits and withdrawals, a race condition might lead to incorrect balances. For example, two simultaneous withdrawal transactions might lead to an overdraft situation that wasn't intended.
3. **File Operations**: When multiple users are allowed to access and modify the same file or resource simultaneously, race conditions can lead to data corruption or inconsistent states. For instance, if a file is being written to by one process while another process is reading from it, the reading process might get incomplete or corrupted data.
4. **Session Management**: If a web application uses session tokens for user authentication and doesn't manage them properly, concurrent requests might lead to users' sessions being mixed up. This could potentially allow one user to access another user's session data.

The most well-known type of race condition to detect and exploit is bypassing a limited overrun race condition through the business logic of the application. "Limited overrun race conditions" in the context of web security refers to the practice of preventing or minimizing the negative impact of race conditions that could lead to unauthorized access, data corruption, or other security vulnerabilities. This involves designing and implementing measures to ensure that multiple concurrent processes or users do not inadvertently compromise the security or integrity of a web application's resources or data.

Let's consider an example of code from a web application that has a rate-limiting mechanism in place to prevent users from making too many requests in a short period. However, due to a race condition, an attacker might be able to bypass this rate-limiting and make more requests than they should.

A screen shot of a computer screen

Description automatically generated

The web application has an endpoint that allows users to perform a certain action, such as sending messages. The application enforces a rate limit of 10 requests per minute for this action. The vulnerability lies in the fact that the rate limit check, action execution, and request count increment are not performed atomically. If two or more requests from the same user arrive simultaneously, they might pass the rate limit check before the request count is incremented, allowing them to exceed the rate limit.

**Attack Scenario**: An attacker sends 10 requests to the ‘perform\_action’ endpoint simultaneously. Due to the race condition, all requests pass the rate limit check because the request count hasn't been updated yet. The attacker can perform the action 10 times, effectively bypassing the rate limit.

**Mitigation**: To mitigate this race condition, you need to have the rate limit check, action execution, and request count increment be performed as a single atomic operation. You should also use synchronization mechanisms like locks or semaphores to ensure that only one request can modify the request count at a time.

Here's how the code could be updated to mitigate the race condition:

A screen shot of a computer

Description automatically generated

**By using a lock to synchronize access to the rate limit check and request count update, the race condition is mitigated. Only one request will be able to perform these operations at a time, preventing the attacker from bypassing the rate limit through concurrent requests.**

There are also ways of manually testing for web race conditions via bypassing rate limiting by using Burp Suite. For the following demonstrations we will use labs from Portswigger Academy to demonstrate our methodology and exploitation of web race conditions. According to an article from Portswigger, you should have Burp Suite version 2023.9 as it, “adds powerful new capabilities to Burp Repeater that enable you to easily send a group of parallel requests in a way that greatly reduces the impact of one of these factors, namely network jitter”.

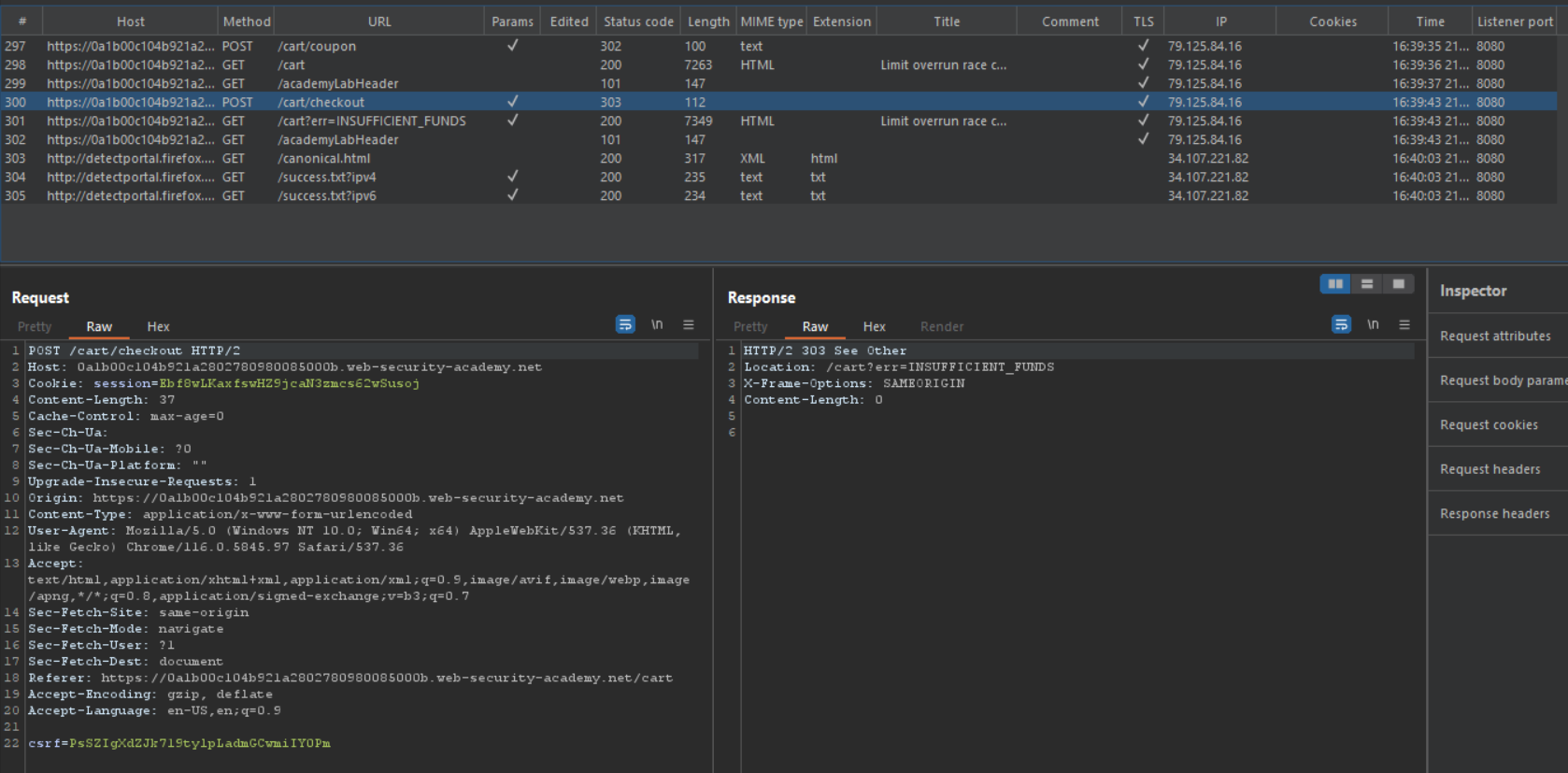
First, we will look at how to detect and exploit limit overrun race conditions with burp repeater. We demonstrate how to accomplish this by completing the following Portswigger Academy lab, “Limit Overrun Race Conditions”. The goals of the lab are as follows: “This lab's purchasing flow contains a race condition that enables you to purchase items for an unintended price. To solve the lab, successfully purchase a Lightweight L33t Leather Jacket. You can log in to your account with the following credentials: wiener:peter.”.

Step 1: We want to look for places that we can send many requests at the same time to test the rate limiting.

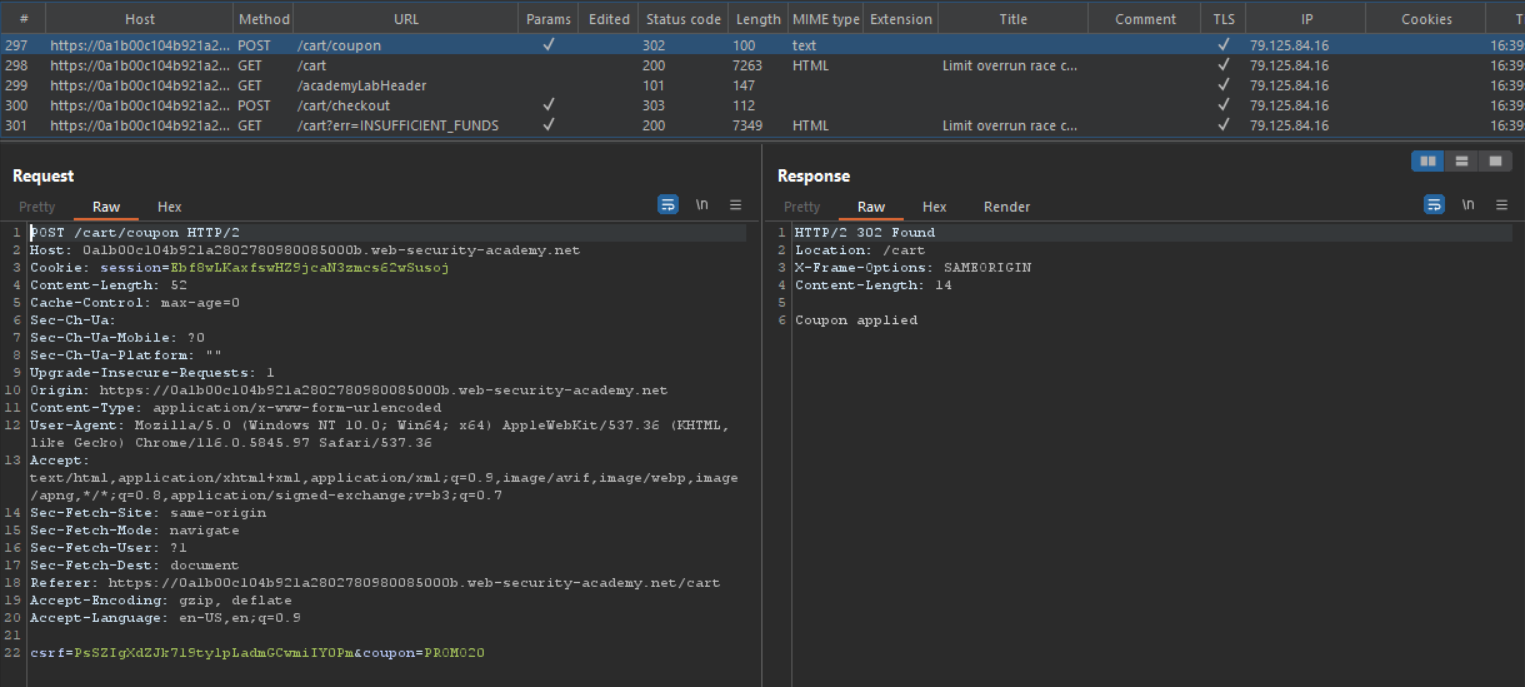
Step 2: Once we login, we can immediately see the Lightweight L33t Leather Jacket we need to purchase on the homepage. Add it to the cart.

Step 3: Go to your cart. Add the promo code we were given and then click, ‘place order’.

Step 4: Once we try to purchase the jacket, we see there is an error message stating, “Not enough store credit for this purchase”. Go to Burp and locate the two POST requests that were sent. The second POST request returned a 303 SEE OTHER response code with an error message stating insufficient funds.



Step 5: Now go to the first POST request (/cart/coupon) and notice how this request was from when we applied our PROMOCODE. Send this request to repeater about 20 times so we can knock off the price to under $50.

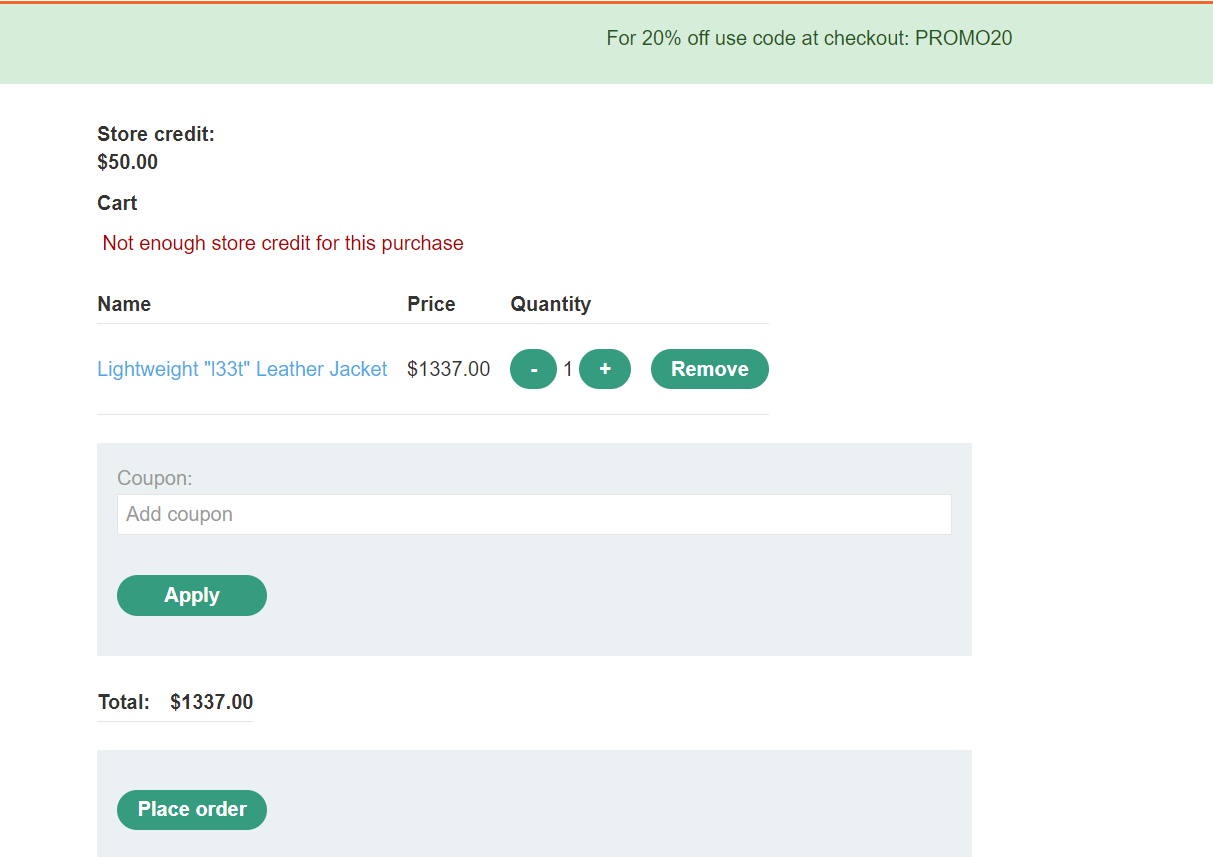


Step 6: First click the 3 dots at the top right corner in Burp, then select *create new group*, the add all 20 tabs to the group and select a color of your choosing.

A screenshot of a computer

Description automatically generated

Step 7: Next step, which is very important, is to remove the PROMOCODE because we cannot submit the one that we have already applied twice. Then click place order. You should be returned with an error message that states, “Not enough store credit for this purchase.”.

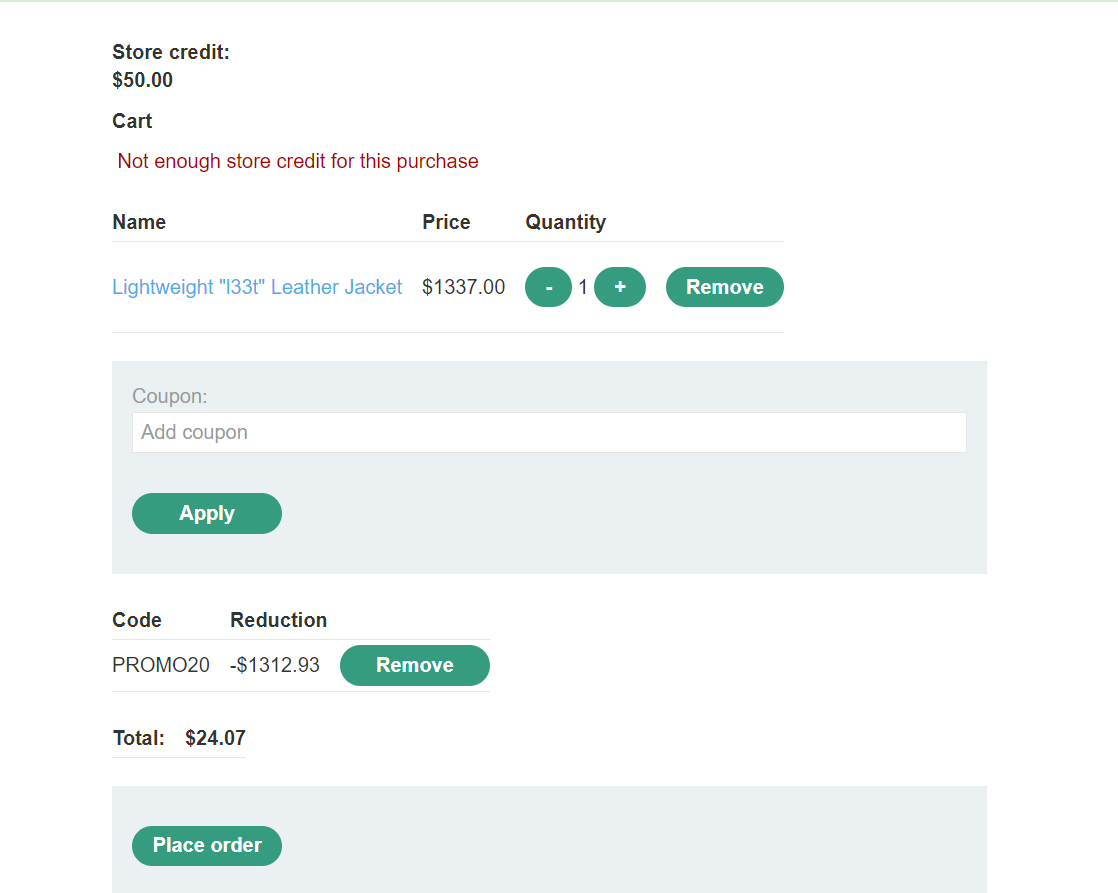


Step 8: Then we go back to burp and before we hit send, click on the down arrow, and select, “send group in parallel (single-packet attack)”. This will allow us to send multiple requests concurrently in order to use the coupon not just once but 20 TIMES!!!! Notice how in the HTTP response for every HTTP request in your created group it says, “Coupon applied “ versus “Coupon already applied”.

A screenshot of a computer program

Description automatically generated

Step 9: Now go back to the application and refresh the page. As we can see, the total price of our jacket is $24.07, we successfully applied a lot more than a 20% discount to our order. Our race condition exploitation was successful so now you can click place order and the lab will be completed.

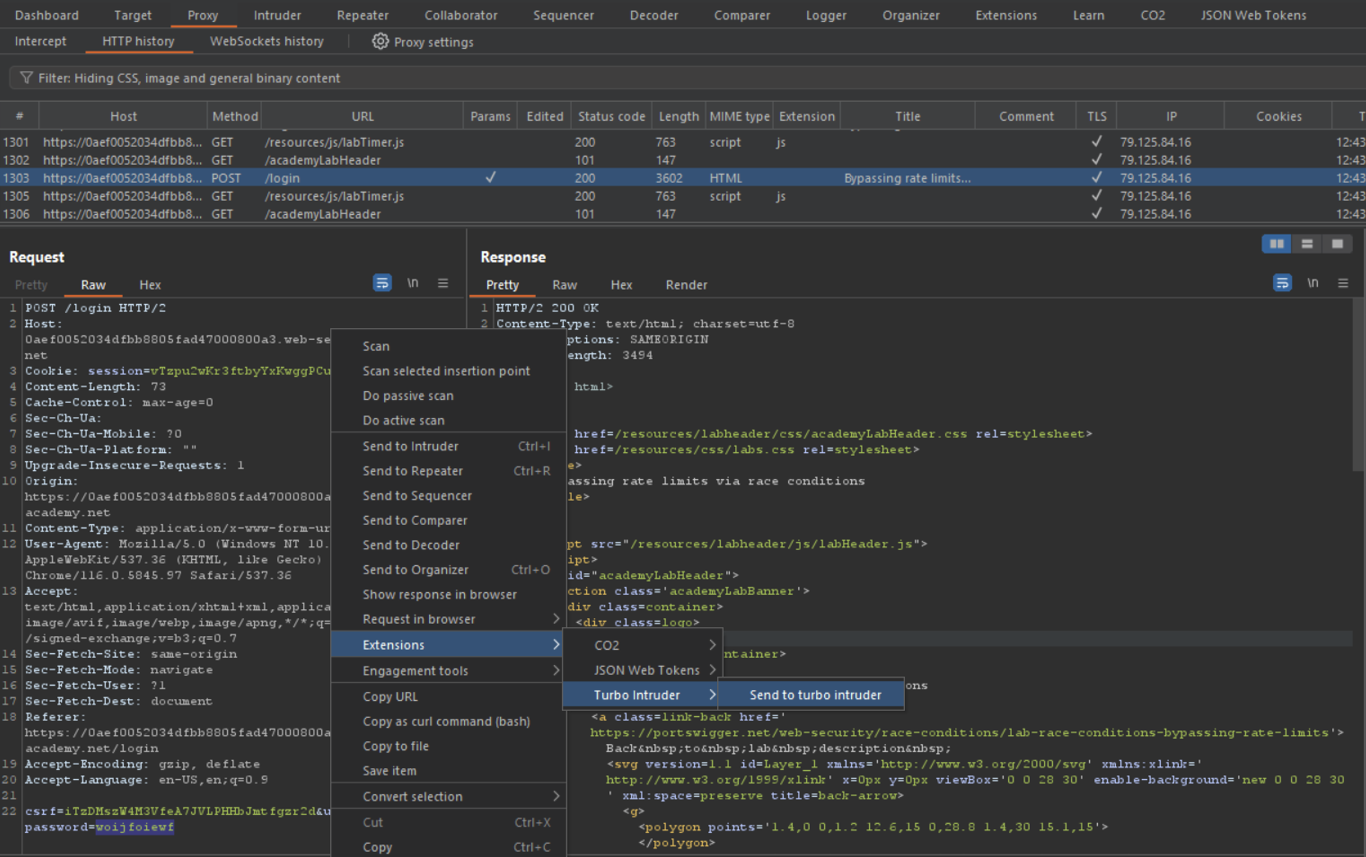


In addition of the new single-packet attack in Burp Repeater, the latest version of the Turbo Intruder extension also supports this technique. Turbo Intruder requires some proficiency in Python, but is better suited to more complex attacks, such as ones that require multiple retries or an extremely large number of requests. In our next example, we will exploit a race condition that allows us to bypass the rate limit and successfully brute force the password for the user. We will use a lab from Portswigger Academy, “Lab: Bypassing rate limits via race conditions”, to demonstrate how this exploit works.

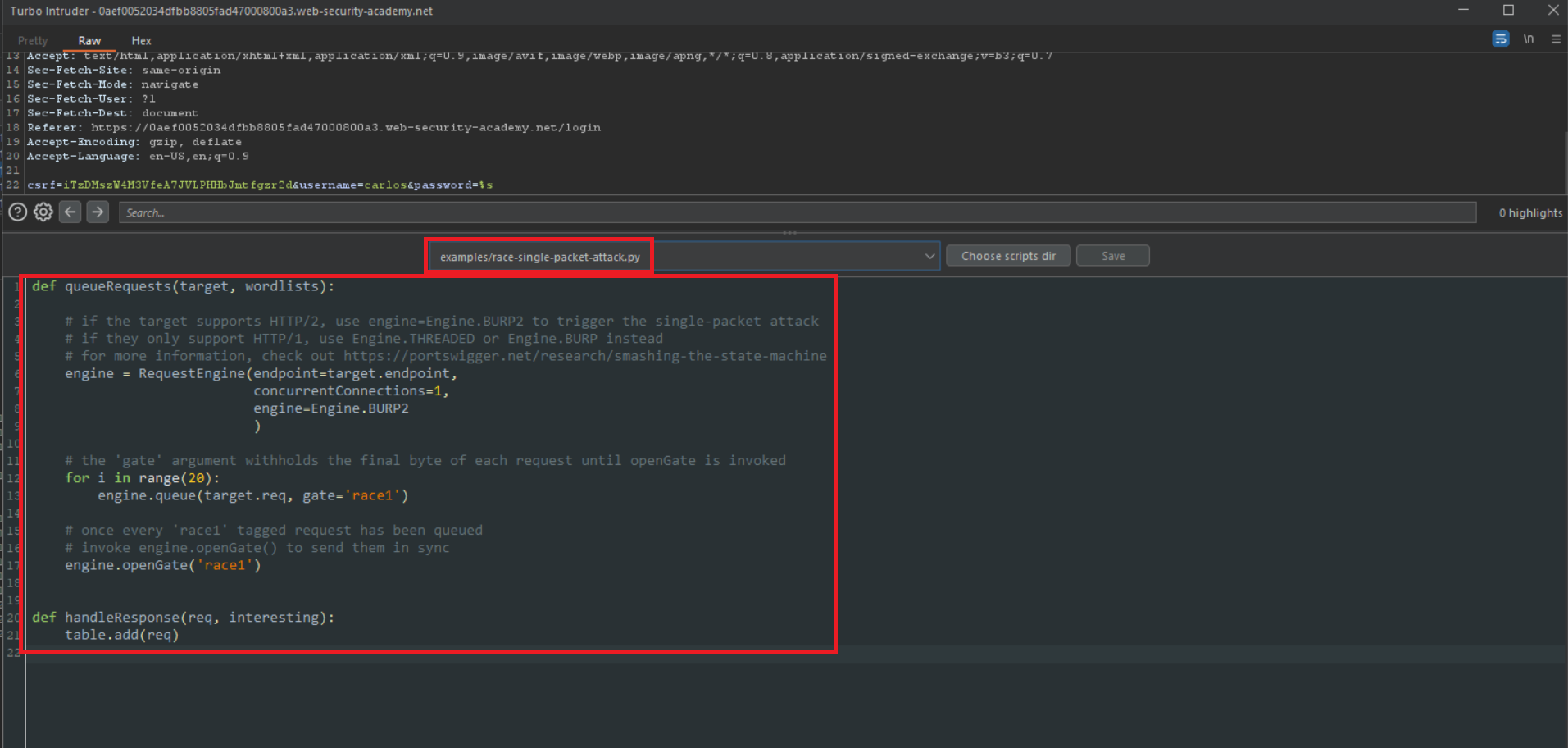
The goal of this lab is as follows: “This lab's login mechanism uses rate limiting to defend against brute-force attacks. However, this can be bypassed due to a race condition. To solve the lab: Work out how to exploit the race condition to bypass the rate limit. Successfully brute-force the password for the user Carlos. Log in and access the admin panel. Delete the user Carlos. You can log in to your account with the following credentials: wiener:peter. Solving this lab requires Burp Suite 2023.9 or higher. You should also use the latest version of the Turbo Intruder, which is available from the BApp Store. You have a time limit of 15 mins. If you don't solve the lab within the time limit, you can reset the lab. However, Carlos's password changes each time.”. We have also been provided a password list that can be used for the brute-force attack of this lab.

Step 1: Input an invalid login with the username as Carlos and the password as something arbitrary.

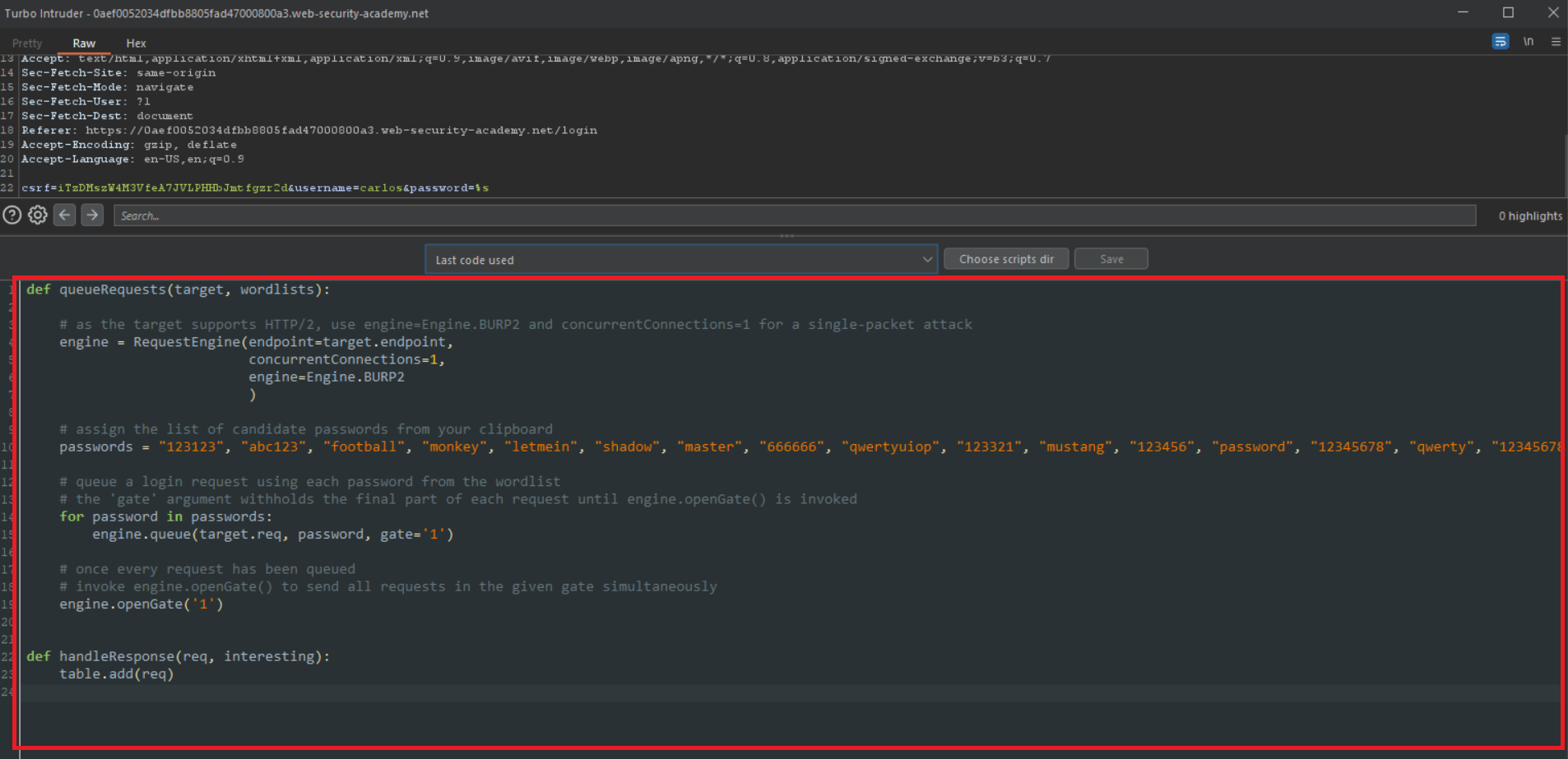
Step 2: Find the invalid login attempt with burp and send it to burp intruder with the password field in the http request highlighted as that will be our insertion point for our brute-force attack.



Step 3: In turbo intruder, we select the default examples/race-single-packet-attack.py script.



Step 4: As seen below we made the following editions to the script:

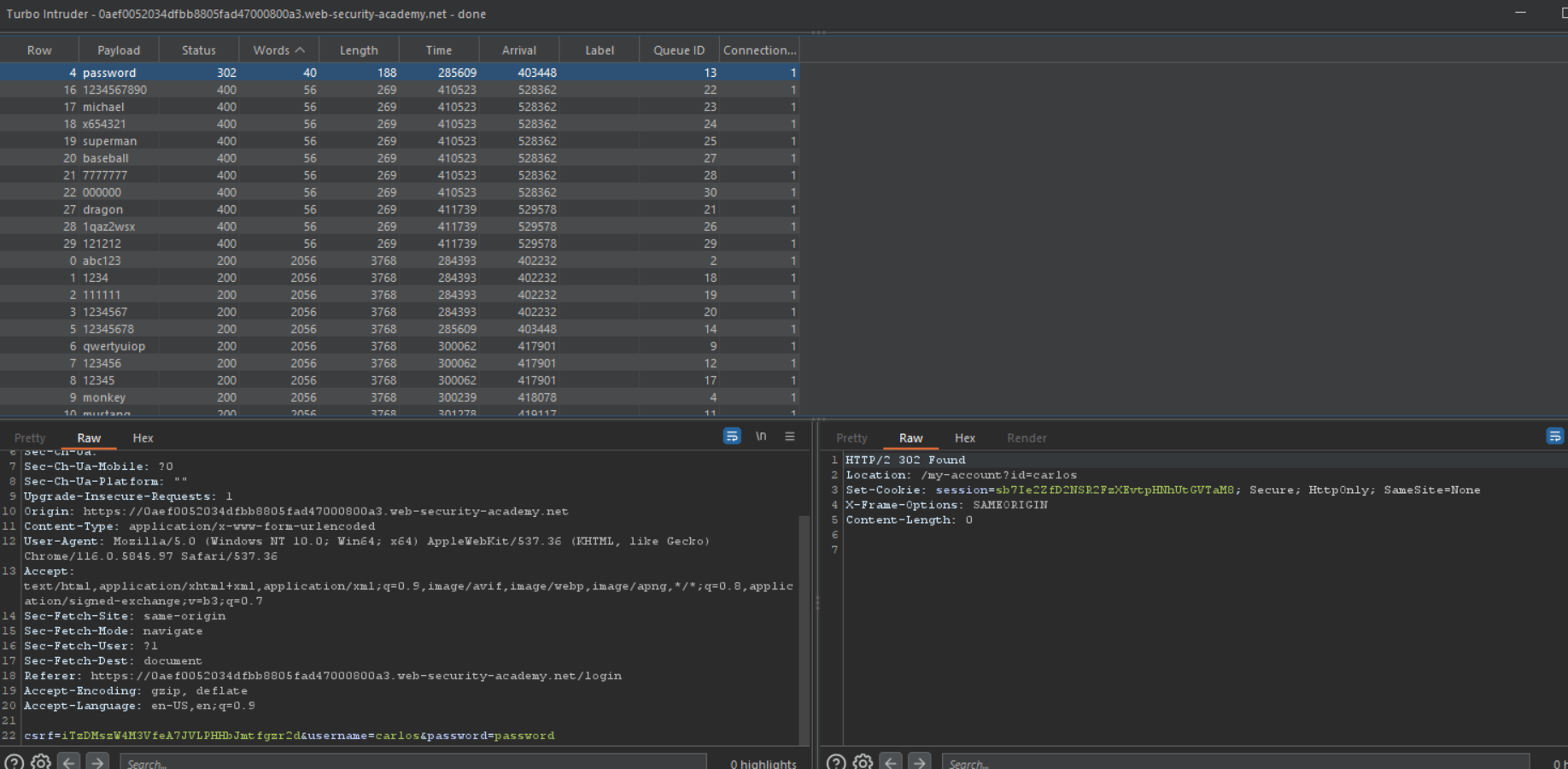


As seen in these 2 python scripts, we made some editions to the default script shown in step 3. The default script in step 3 and our edited script in step 4 are similar in structure and purpose because both of their main functionalities involve sending requests concurrently. However, the first script introduces the concept of “racing requests”, while the second script focuses on a straightforward brute-force attack. Let’s break down the changes made between the 2 scripts:

**Changes Made:**

1. **Racing Requests vs. Brute-Force Attack:**
   * In the first script:
     + The script introduces the concept of "racing" requests, where it sends multiple requests in a way that they will reach the server simultaneously, exploiting potential concurrency issues.
     + A loop (**for i in range(20)**) is used to enqueue requests with the **gate** tag **'race1'**, and then **engine.openGate('race1')** is used to send all these tagged requests in sync.
   * In the second script:
     + The script focuses on a more straightforward brute-force attack on a login page using a list of candidate passwords.
     + It iterates through the **passwords** list, queuing login requests with each password, and then sends all the requests simultaneously using **engine.openGate('1')**.
2. **Password Source:**
   * In both scripts, the list of candidate passwords is used from a different source:
     + In the first script, the password candidates are generated or selected internally since there's no specific source mentioned in the code.
     + In the second script, the password candidates are extracted from the **passwords list**.
3. **Request Queuing Mechanism:**
   * Both scripts use the **engine.queue()** function to queue requests. However, the requests differ in nature:
     + In the first script, requests are crafted to exploit concurrency vulnerabilities, but the details are not provided in the code snippet.
     + In the second script, requests are straightforward login requests with username and password parameters.

Step 5: As we can see by the 302 status code, the words, and the length of the payload with the password, “password”, this is the password for Carlos account, and we have successfully exploited a race condition that allowed us to brute-force Carlos account via bypassing the rate limits.



Step 6: Login to Carlos account (you may have to wait for the account lockout policy wait time to be over). Once logged in, go to the admin panel, and delete Carlos account.

A screenshot of a computer

Description automatically generated

**PREVENTION**

Preventing web race conditions involves a combination of careful design, proper synchronization, and well-structured code. While it may not always be possible to eliminate all potential race conditions, following these best practices significantly reduces their probability and impact on the web application's security and functionality:  
  
Synchronization Mechanisms:  
Use synchronization mechanisms like locks, semaphores, and mutexes to control access to shared resources or critical sections of code. These mechanisms ensure that only one thread or process can access the resource at a time, preventing concurrent modifications.  
  
Database Transactions:  
When working with databases, use transactions to group related database operations together. Transactions ensure that either all operations in the transaction are completed, or none of them are, helping maintain data consistency.  
  
Immutable Data Structures:  
Whenever possible, use immutable data structures or techniques to eliminate the need for data modification after creation. Immutable data reduces the risk of race conditions related to data changes.  
  
Session Isolation:  
Properly manage user sessions to prevent session-related race conditions. Use unique session tokens for each user and ensure that session data is isolated to prevent data leakage.  
  
Rate Limiting:  
Implement rate limiting mechanisms to restrict the number of requests a user can make within a certain time frame. Rate limiting helps prevent abuse and potential race conditions arising from excessive concurrent requests.  
  
Caching Strategies:  
Develop caching strategies that include cache validation and proper cache expiration policies. This helps avoid serving outdated or incorrect cached content to users.  
  
Concurrency Models:  
Choose appropriate concurrency models and frameworks for your application. Some frameworks provide built-in mechanisms for managing concurrency, which can help reduce the chances of race conditions.  
  
Error Handling:  
Implement proper error handling to gracefully manage unexpected situations that might arise due to race conditions. This helps prevent application crashes or unexpected behavior.  
  
Design for Single Responsibility:  
Follow the principle of "Single Responsibility" in your code design. Divide your code into smaller, focused components that perform distinct tasks. This can help reduce the complexity and potential for race conditions.