# PortSwigger Race Conditions Lab Notes

1. Limit overrun race conditions

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.

For a faster and more convenient way to trigger the race condition, we recommend that you solve this lab using the [Trigger race conditions](https://github.com/PortSwigger/bambdas/blob/main/CustomAction/ProbeForRaceCondition.bambda) custom action. This is only available in Burp Suite Professional.

**Note**

Solving this lab requires Burp Suite 2023.9 or higher.

 Solution - Burp Suite Professional

**Predict a potential collision**

1. Log in and buy the cheapest item possible, making sure to use the provided discount code so that you can study the purchasing flow.
2. Consider that the shopping cart mechanism and, in particular, the restrictions that determine what you are allowed to order, are worth trying to bypass.
3. In Burp, from the proxy history, identify all endpoints that enable you to interact with the cart. For example, a POST /cart request adds items to the cart and a POST /cart/coupon request applies the discount code.
4. Try to identify any restrictions that are in place on these endpoints. For example, observe that if you try applying the discount code more than once, you receive a Coupon already applied response.
5. Make sure you have an item to your cart, then send the GET /cart request to Burp Repeater.
6. In Repeater, try sending the GET /cart request both with and without your session cookie. Confirm that without the session cookie, you can only access an empty cart. From this, you can infer that:
   * The state of the cart is stored server-side in your session.
   * Any operations on the cart are keyed on your session ID or the associated user ID.

This indicates that there is potential for a collision.

1. Consider that there may be a race window between when you first apply a discount code and when the database is updated to reflect that you've done this already.

**Benchmark the behavior**

1. Make sure there is no discount code currently applied to your cart.
2. Send the request for applying the discount code (POST /cart/coupon) to Repeater.
3. In Repeater, send the POST /cart/coupon request twice.
4. Observe that the first response confirms that the discount was successfully applied, but the second response rejects the code with the same Coupon already applied message.

**Probe for clues**

1. Remove the discount code from your cart.
2. In **Repeater**, open the **Custom actions** side panel.
3. Click **New > From template**, then select **Trigger race condition**.
4. Save the template to the **Custom actions** side panel without making any modifications.
5. Click  beside the **Trigger race condition** custom action. The request is sent 20 times in parallel.
6. In the browser, refresh your cart and confirm that the 20% reduction has been applied more than once, resulting in a significantly cheaper order.

**Prove the concept**

1. Remove the applied codes and the arbitrary item from your cart and add the leather jacket to your cart instead.
2. Resend the group of POST /cart/coupon requests in parallel.
3. Refresh the cart and check the order total:
   * If the order total is still higher than your remaining store credit, remove the discount codes and repeat the attack.
   * If the order total is less than your remaining store credit, purchase the jacket to solve the lab.

 Solution - Burp Suite Community Edition

**Predicting a potential collision**

1. Log in and buy the cheapest item possible, making sure to use the provided discount code so that you can study the purchasing flow.
2. Consider that the shopping cart mechanism and, in particular, the restrictions that determine what you are allowed to order, are worth trying to bypass.
3. In Burp, from the proxy history, identify all endpoints that enable you to interact with the cart. For example, a POST /cart request adds items to the cart and a POST /cart/coupon request applies the discount code.
4. Try to identify any restrictions that are in place on these endpoints. For example, observe that if you try applying the discount code more than once, you receive a Coupon already applied response.
5. Make sure you have an item to your cart, then send the GET /cart request to Burp Repeater.
6. In Repeater, try sending the GET /cart request both with and without your session cookie. Confirm that without the session cookie, you can only access an empty cart. From this, you can infer that:
   * The state of the cart is stored server-side in your session.
   * Any operations on the cart are keyed on your session ID or the associated user ID.

This indicates that there is potential for a collision.

1. Consider that there may be a race window between when you first apply a discount code and when the database is updated to reflect that you've done this already.

**Benchmarking the behavior**

1. Make sure there is no discount code currently applied to your cart.
2. Send the request for applying the discount code (POST /cart/coupon) to Repeater.
3. In Repeater, add the new tab to a group. For details on how to do this, see [Creating a new tab group](https://portswigger.net/burp/documentation/desktop/tools/repeater/groups#creating-a-new-tab-group).
4. Right-click the grouped tab, then select **Duplicate tab**. Create 19 duplicate tabs. The new tabs are automatically added to the group.
5. Send the group of requests in sequence, using separate connections to reduce the chance of interference. For details on how to do this, see [Sending requests in sequence](https://portswigger.net/burp/documentation/desktop/tools/repeater/send-group#sending-requests-in-sequence).
6. Observe that the first response confirms that the discount was successfully applied, but the rest of the responses consistently reject the code with the same **Coupon already applied** message.

**Probing for clues**

1. Remove the discount code from your cart.
2. In Repeater, send the group of requests again, but this time in parallel, effectively applying the discount code multiple times at once. For details on how to do this, see [Sending requests in parallel](https://portswigger.net/burp/documentation/desktop/tools/repeater/send-group#sending-requests-in-parallel).
3. Study the responses and observe that multiple requests received a response indicating that the code was successfully applied. If not, remove the code from your cart and repeat the attack.
4. In the browser, refresh your cart and confirm that the 20% reduction has been applied more than once, resulting in a significantly cheaper order.

**Proving the concept**

1. Remove the applied codes and the arbitrary item from your cart and add the leather jacket to your cart instead.
2. Resend the group of POST /cart/coupon requests in parallel.
3. Refresh the cart and check the order total:
   * If the order total is still higher than your remaining store credit, remove the discount codes and repeat the attack.
   * If the order total is less than your remaining store credit, purchase the jacket to solve the lab.
4. Bypassing rate limits via race conditions

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:

1. Work out how to exploit the race condition to bypass the rate limit.
2. Successfully brute-force the password for the user carlos.
3. Log in and access the admin panel.
4. Delete the user carlos.

You can log in to your account with the following credentials: wiener:peter.

You should use the following list of potential passwords:

**Passwords**

**Note**

* 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](https://portswigger.net/bappstore/9abaa233088242e8be252cd4ff534988).
* 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.

 Solution

**Predict a potential collision**

1. Experiment with the login function by intentionally submitting incorrect passwords for your own account.
2. Observe that if you enter the incorrect password more than three times, you're temporarily blocked from making any more login attempts for the same account.
3. Try logging in using another arbitrary username and observe that you see the normal Invalid username or password message. This indicates that the rate limit is enforced per-username rather than per-session.
4. Deduce that the number of failed attempts per username must be stored server-side.
5. Consider that there may be a race window between:
   * When you submit the login attempt.
   * When the website increments the counter for the number of failed login attempts associated with a particular username.

**Benchmark the behavior**

1. From the proxy history, find a POST /login request containing an unsuccessful login attempt for your own account.
2. Send this request to Burp Repeater.
3. In Repeater, add the new tab to a group. For details on how to do this, see [Creating a new tab group](https://portswigger.net/burp/documentation/desktop/tools/repeater/groups#creating-a-new-tab-group).
4. Right-click the grouped tab, then select **Duplicate tab**. Create 19 duplicate tabs. The new tabs are automatically added to the group.
5. Send the group of requests in sequence, using separate connections to reduce the chance of interference. For details on how to do this, see [Sending requests in sequence](https://portswigger.net/burp/documentation/desktop/tools/repeater/send-group#sending-requests-in-sequence).
6. Observe that after two more failed login attempts, you're temporarily locked out as expected.

**Probe for clues**

1. Send the group of requests again, but this time in parallel. For details on how to do this, see [Sending requests in parallel](https://portswigger.net/burp/documentation/desktop/tools/repeater/send-group#sending-requests-in-parallel)
2. Study the responses. Notice that although you have triggered the account lock, more than three requests received the normal Invalid username and password response.
3. Infer that if you're quick enough, you're able to submit more than three login attempts before the account lock is triggered.

**Prove the concept**

1. Still in Repeater, highlight the value of the password parameter in the POST /login request.
2. Right-click and select **Extensions > Turbo Intruder > Send to turbo intruder**.
3. In Turbo Intruder, in the request editor, notice that the value of the password parameter is automatically marked as a payload position with the %s placeholder.
4. Change the username parameter to carlos.
5. From the drop-down menu, select the examples/race-single-packet-attack.py template.
6. In the Python editor, edit the template so that your attack queues the request once using each of the candidate passwords. For simplicity, you can copy the following example:

***def queueRequests(target, wordlists):***

***# as the target supports HTTP/2, use engine=Engine.BURP2 and concurrentConnections=1 for a single-packet attack***

***engine = RequestEngine(endpoint=target.endpoint,***

***concurrentConnections=1,***

***engine=Engine.BURP2***

***)***

***# assign the list of candidate passwords from your clipboard***

***passwords = wordlists.clipboard***

***# queue a login request using each password from the wordlist***

***# the 'gate' argument withholds the final part of each request until engine.openGate() is invoked***

***for password in passwords:***

***engine.queue(target.req, password, gate='1')***

***# once every request has been queued***

***# invoke engine.openGate() to send all requests in the given gate simultaneously***

***engine.openGate('1')***

***def handleResponse(req, interesting):***

***table.add(req)***

1. Note that we're assigning the password list from the clipboard by referencing wordlists.clipboard. Copy the list of candidate passwords to your clipboard.
2. Launch the attack.
3. Study the responses.
   * If you have no successful logins, wait for the account lock to reset and then repeat the attack. You might want to remove any passwords from the list that you know are incorrect.
   * If you get a 302 response, notice that this login appears to be successful. Make a note of the corresponding password from the **Payload** column.

A screenshot of a computer

AI-generated content may be incorrect.

1. Wait for the account lock to reset, then log in as carlos using the identified password.
2. Access the admin panel and delete the user carlos to solve the lab.
3. Multi-endpoint race conditions

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 into your account with the following credentials: wiener:peter.

**Note**

Solving this lab requires Burp Suite 2023.9 or higher.

**Tip**

When experimenting, we recommend purchasing the gift card as you can later redeem this to avoid running out of store credit.

 Solution

**Predict a potential collision**

1. Log in and purchase a gift card so you can study the purchasing flow.
2. Consider that the shopping cart mechanism and, in particular, the restrictions that determine what you are allowed to order, are worth trying to bypass.
3. From the proxy history, identify all endpoints that enable you to interact with the cart. For example, a POST /cart request adds items to the cart and a POST /cart/checkout request submits your order.
4. Add another gift card to your cart, then send the GET /cart request to Burp Repeater.
5. In Repeater, try sending the GET /cart request both with and without your session cookie. Confirm that without the session cookie, you can only access an empty cart. From this, you can infer that:
   * The state of the cart is stored server-side in your session.
   * Any operations on the cart are keyed on your session ID or the associated user ID.

This indicates that there is potential for a collision.

1. Notice that submitting and receiving confirmation of a successful order takes place over a single request/response cycle.
2. Consider that there may be a race window between when your order is validated and when it is confirmed. This could enable you to add more items to the order after the server checks whether you have enough store credit.

**Benchmark the behavior**

1. Send both the POST /cart and POST /cart/checkout request to Burp Repeater.
2. In Repeater, add the two tabs to a new group. For details on how to do this, see [Creating a new tab group](https://portswigger.net/burp/documentation/desktop/tools/repeater/groups#creating-a-new-tab-group)
3. Send the two requests in sequence over a single connection a few times. Notice from the response times that the first request consistently takes significantly longer than the second one. For details on how to do this, see [Sending requests in sequence](https://portswigger.net/burp/documentation/desktop/tools/repeater/send-group#sending-requests-in-sequence).
4. Add a GET request for the homepage to the start of your tab group.
5. Send all three requests in sequence over a single connection. Observe that the first request still takes longer, but by "warming" the connection in this way, the second and third requests are now completed within a much smaller window.
6. Deduce that this delay is caused by the back-end network architecture rather than the respective processing time of the each endpoint. Therefore, it is not likely to interfere with your attack.
7. Remove the GET request for the homepage from your tab group.
8. Make sure you have a single gift card in your cart.
9. In Repeater, modify the POST /cart request in your tab group so that the productId parameter is set to 1, that is, the ID of the **Lightweight L33t Leather Jacket**.
10. Send the requests in sequence again.
11. Observe that the order is rejected due to insufficient funds, as you would expect.

**Prove the concept**

1. Remove the jacket from your cart and add another gift card.
2. In Repeater, try sending the requests again, but this time in parallel. For details on how to do this, see [Sending requests in parallel](https://portswigger.net/burp/documentation/desktop/tools/repeater/send-group#sending-requests-in-parallel).
3. Look at the response to the POST /cart/checkout request:
   * If you received the same "insufficient funds" response, remove the jacket from your cart and repeat the attack. This may take several attempts.
   * If you received a 200 response, check whether you successfully purchased the leather jacket. If so, the lab is solved.

🔑 What is the race condition here?

The vulnerability comes from how the checkout process works in the backend:

1. Step 1 – Verification:  
   The server checks your basket → calculates the total → compares it with your credit balance.
2. Step 2 – Confirmation:  
   If you have enough credit, it finalizes the order and deducts money.

⚡ Problem: these two steps are not atomic (not locked together). This means that between verification and deduction, another request can slip in and add an expensive item without being checked against your balance.

⚙️ How the attack works

1. Log in as wiener:peter → you have $100 credit.
2. Add a cheap item (gift card, $10) and go to checkout.
3. Capture two endpoints in Burp:
   * Add jacket → adds the expensive jacket to your basket.
   * Checkout → finalizes the purchase.
4. Send both requests at the same time (race condition attack).
   * The checkout verifies only the $10 item.
   * Before confirmation, the “add jacket” request sneaks in.
   * Result: checkout approves both items even though you only had enough money for the $10.

🧾 Outcome

* You end up buying the jacket + gift card for the price of the gift card.

1. Single-endpoint race conditions

This lab's email change feature contains a race condition that enables you to associate an arbitrary email address with your account.

Someone with the address carlos@ginandjuice.shop has a pending invite to be an administrator for the site, but they have not yet created an account. Therefore, any user who successfully claims this address will automatically inherit admin privileges.

To solve the lab:

1. Identify a race condition that lets you claim an arbitrary email address.
2. Change your email address to carlos@ginandjuice.shop.
3. Access the admin panel.
4. Delete the user carlos

You can log in to your own account with the following credentials: wiener:peter.

You also have access to an email client, where you can view all emails sent to @exploit-<YOUR-EXPLOIT-SERVER-ID>.exploit-server.net addresses.

**Note**

Solving this lab requires Burp Suite 2023.9 or higher.

 Solution

**Predict a potential collision**

1. Log in and attempt to change your email to anything@exploit-<YOUR-EXPLOIT-SERVER-ID>.exploit-server.net. Observe that a confirmation email is sent to your intended new address, and you're prompted to click a link containing a unique token to confirm the change.
2. Complete the process and confirm that your email address has been updated on your account page.
3. Try submitting two different @exploit-<YOUR-EXPLOIT-SERVER-ID>.exploit-server.net email addresses in succession, then go to the email client.
4. Notice that if you try to use the first confirmation link you received, this is no longer valid. From this, you can infer that the website only stores one pending email address at a time. As submitting a new email address edits this entry in the database rather than appending to it, there is potential for a collision.

**Benchmark the behavior**

1. Send the POST /my-account/change-email request to Repeater.
2. In Repeater, add the new tab to a group. For details on how to do this, see [Creating a new tab group](https://portswigger.net/burp/documentation/desktop/tools/repeater/groups#creating-a-new-tab-group).
3. Right-click the grouped tab, then select **Duplicate tab**. Create 19 duplicate tabs. The new tabs are automatically added to the group.
4. In each tab, modify the first part of the email address so that it is unique to each request, for example, test1@exploit-<YOUR-EXPLOIT-SERVER-ID>.exploit-server.net, test2@..., test3@... and so on.
5. Send the group of requests in sequence over separate connections. For details on how to do this, see [Sending requests in sequence](https://portswigger.net/burp/documentation/desktop/tools/repeater/send-group#sending-requests-in-sequence).
6. Go back to the email client and observe that you have received a single confirmation email for each of the email change requests.

**Probe for clues**

1. In Repeater, send the group of requests again, but this time in parallel, effectively attempting to change the pending email address to multiple different values at the same time. For details on how to do this, see [Sending requests in parallel](https://portswigger.net/burp/documentation/desktop/tools/repeater/send-group#sending-requests-in-parallel).
2. Go to the email client and study the new set of confirmation emails you've received. Notice that, this time, the recipient address doesn't always match the pending new email address.
3. Consider that there may be a race window between when the website:
   1. Kicks off a task that eventually sends an email to the provided address.
   2. Retrieves data from the database and uses this to render the email template.
4. Deduce that when a parallel request changes the pending email address stored in the database during this window, this results in confirmation emails being sent to the wrong address.

**Prove the concept**

1. In Repeater, create a new group containing two copies of the POST /my-account/change-email request.
2. Change the email parameter of one request to anything@exploit-<YOUR-EXPLOIT-SERVER-ID>.exploit-server.net.
3. Change the email parameter of the other request to carlos@ginandjuice.shop.
4. Send the requests in parallel.
5. Check your inbox:
   * If you received a confirmation email in which the address in the body matches your own address, resend the requests in parallel and try again.
   * If you received a confirmation email in which the address in the body is carlos@ginandjuice.shop, click the confirmation link to update your address accordingly.
6. Go to your account page and notice that you now see a link for accessing the admin panel.
7. Visit the admin panel and delete the user carlos to solve the lab.

🔑 The Vulnerability

* The app lets a user update their email.
* When you request an email change:
  1. The server stores the new email temporarily.
  2. It sends a confirmation link to that email.
  3. Once you click confirm, the change is finalized.

⚡ Bug: the app doesn’t properly lock the “update email” action. If you send two email update requests at the same time, they both get partially processed.

⚙️ How the Race Condition Works

1. Send two concurrent requests to update email:
   * Request A → change to hack1@labid.web-security-academy.net
   * Request B → change to hack2@labid.web-security-academy.net
2. Both requests are processed at nearly the same time:
   * The system creates two pending email-change operations.
   * It sends two confirmation links (one for hack1, one for hack2).
3. Confirm mismatch:
   * If you confirm request A’s link, it sometimes finalizes request B’s update (depending on timing).
   * This overlap means one confirmation can “complete” another request.
4. Exploit with admin’s email:
   * Replace one of the requests with the administrator’s real email address.
   * Because of the collision, the confirmation step applies to the admin email instead of your own.
5. Result:
   * Your account now has the admin’s email associated with it.
   * The system assumes your user is an admin → you get access to the admin panel.

🧾 Outcome

* After hijacking the admin’s email, the attacker logs in as usual.
* Now the account is treated like an admin account.
* The attacker visits the admin panel and deletes Carlos to solve the lab.

1. Exploiting time-sensitive vulnerabilities

This lab contains a password reset mechanism. Although it doesn't contain a race condition, you can exploit the mechanism's broken cryptography by sending carefully timed requests.

To solve the lab:

1. Identify the vulnerability in the way the website generates password reset tokens.
2. Obtain a valid password reset token for the user carlos.
3. Log in as carlos.
4. Access the admin panel and delete the user carlos.

You can log into your account with the following credentials: wiener:peter.

**Note**

Solving this lab requires Burp Suite 2023.9 or higher.

 Solution

**Study the behavior**

1. Study the password reset process by submitting a password reset for your own account and observe that you're sent an email containing a reset link. The query string of this link includes your username and a token.
2. Send the POST /forgot-password request to Burp Repeater.
3. In Repeater, send the request a few times, then check your inbox again.
4. Observe that every reset request results in a link with a different token.
5. Consider the following:
   * The token is of a consistent length. This suggests that it's either a randomly generated string with a fixed number of characters, or could be a hash of some unknown data, which may be predictable.
   * The fact that the token is different each time indicates that, if it is in fact a hash digest, it must contain some kind of internal state, such as an RNG, a counter, or a timestamp.
6. Duplicate the Repeater tab and add both tabs to a new group. For details on how to do this, see [Creating a new tab group](https://portswigger.net/burp/documentation/desktop/tools/repeater/groups#creating-a-new-tab-group)
7. Send the pair of reset requests in parallel a few times. For details on how to do this, see [Sending requests in parallel](https://portswigger.net/burp/documentation/desktop/tools/repeater/send-group#sending-requests-in-parallel).
8. Observe that there is still a significant delay between each response and that you still get a different token in each confirmation email. Infer that your requests are still being processed in sequence rather than concurrently.

**Bypass the per-session locking restriction**

1. Notice that your session cookie suggests that the website uses a PHP back-end. This could mean that the server only processes one request at a time per session.
2. Send the GET /forgot-password request to Burp Repeater, remove the session cookie from the request, then send it.
3. From the response, copy the newly issued session cookie and CSRF token and use them to replace the respective values in one of the two POST /forgot-password requests. You now have a pair of password reset requests from two different sessions.
4. Send the two POST requests in parallel a few times and observe that the processing times are now much more closely aligned, and sometimes identical.

**Confirm the vulnerability**

1. Go back to your inbox and notice that when the response times match for the pair of reset requests, this results in two confirmation emails that use an identical token. This confirms that a timestamp must be one of the inputs for the hash.
2. Consider that this also means the token would be predictable if you knew the other inputs for the hash function.
3. Notice the separate username parameter. This suggests that the username might not be included in the hash, which means that two different usernames could theoretically have the same token.
4. In Repeater, go to the pair of POST /forgot-password requests and change the username parameter in one of them to carlos.
5. Resend the two requests in parallel. If the attack worked, both users should be assigned the same reset token, although you won't be able to see this.
6. Check your inbox again and observe that, this time, you've only received one new confirmation email. Infer that the other email, hopefully containing the same token, has been sent to Carlos.
7. Copy the link from the email and change the username in the query string to carlos.
8. Visit the URL in the browser and observe that you're taken to the form for setting a new password as normal.
9. Set the password to something you'll remember and submit the form.
10. Try logging in as carlos using the password you just set.
    * If you can't log in, resend the pair of password reset emails and repeat the process.
    * If you successfully log in, visit the admin panel and delete the user carlos to solve the lab.

🔑 Core Vulnerability

* The password reset mechanism relies on CSRF tokens + reset tokens tied to sessions.
* The app doesn’t generate them safely → if you send multiple reset requests in parallel, sometimes two requests get issued with the same reset token.
* This “collision” creates a race condition that lets you steal or reuse another user’s valid reset token.

⚙️ How It Was Exploited

1. Map the flow:
   * /forgot-password → request reset.
   * Email contains reset link with a token.
   * /reset-password → submit new password.
   * Confirmation endpoint finalizes change.
2. Find the weakness:
   * Tokens are generated per-session (PHPSESSID + CSRF).
   * By sending multiple reset requests at the same time, the attacker notices sometimes both responses get the same token.
3. Exploit it on target (Carlos):
   * Attacker sends simultaneous reset requests for Carlos.
   * They catch one of the valid reset tokens in their own response.
   * Even though Carlos also gets the email, the attacker now has a working token too.
4. Complete the takeover:
   * Use the stolen token with a fresh session + CSRF.
   * Reset Carlos’s password to something the attacker controls.
   * Log in as Carlos → go to the admin panel → delete Carlos (lab goal).

📝 Key Takeaways

* Race condition: multiple requests for the same action (password reset) cause duplicate or reused tokens.
* Impact: attacker can hijack password reset tokens meant for another user.
* Root cause: token generation not atomic, not strongly bound to a single request/session.