# Race Conditions

So in this exercise, please refer to the binder contents of “Race Conditions" for more information regarding options.

**Scenario:**
You call a restaurant to reserve a table for a crucial business lunch. You are familiar with the restaurant and its setup. One particular table number 17 is the preferable choice, considering it has a nice view and it is relatively isolated. You call to make a reservation for table 17 and the host confirms it is free, placing a “Reserved” tag on the table, at the same time another customer is talking with another host making a reservation for the same table.

**This summarizes race conditions.**

Why did this happen then?

1. More than one host was taking reservations
2. Took the host a few minutes to process eg (The booking tag)
3. There is at least a 1 minute window for another client to reserve a reserved table.

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

**Example A:**
A bank account has £100
Two threads try withdraw the money at the same time.
Thread 1 checks the balance and sees £100, withdrawing £45
Thread 2 checks the balance and also sees £100, withdrawing £35

We cannot 100% guarantee which thread will update the remaining balance first, however lets assume that it is thread 1. Thread 1 will put the remaining balance to £55. Afterwards thread 2 might set the remaining balance to £65 if not correctly handled. Thread 2 calculated the £65 should remain in the account after the withdrawal because balance was £100 then thread 2 checked it. In other words the user made **2** withdrawals but the account balance was dedicated for the **2nd thread because it was the last to process the request.**

**Example B:**
A bank account has £75
Two threads try withdraw money at the same time
Thread 1 checks the balance and sees £75 withdrawing £50
Before thread 1 updates the balance
Thread 2 checks the balance incorrectly seeing £75 withdrawing £50.

Thread 2 will proceed with the withdrawal although such as transaction should have been declined. This is called a **Time-of-Check to Time-of-Use (TOCTOU) vulnerability.**

Example code:

```
import threading

x = 0  # Shared variable

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

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

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

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

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

These two threads start together but do nothing except print the value on the screen. Consequently one would expect them to finish simultaneously but there is no guarantee which thread will finish first.

Eg:

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

Both threads finished, but at differing times. In this case thread 2 finished first. Running it again:

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

This time thread 1 finished first!

So why does this happen?

**Parallel execution:**
Web servers may execute man requests in parallel to handle concurrent interactions. If these requests access and modify shared resources or application states without synchronization it can lead to race conditions and unexpected behaviors.

**Database operation:**
Concurrent database operations such as read, modify and write sequences can introduce race conditions. Eg. Two users attempting to update the same record simultaneously may result in inconsistencies in the data or conflicts. The solution to this is proper locking mechanisms and transaction isolation.

**Third-Party Libraries and Services:**
Nowadays web applications often integrate with third party libraries eg, APIs and other services. If these external components are not designed to handle concurrent access properly, race conditions may occur when multiple requests or operations interact simultaneously.

**Client-Server Model:**

**Client:** Program or application that initiates a request for a service. When we browse a web page for instance, our browser requests the web page file.

**Server:** Server provides the program or services requests. This is the response to the incoming request. The web server will respond to an incoming HTTP GET request and sends a HTML page or file to the requesting client.

**Typical Web Application**

**Presentation tier:** This tier consists of web browsers on the client side. Renders HTML, CSS or JavaScript code.

**Application tier:** Web application business logic and functionality. Receives client requests, processes them and interacts with the data tier. Uses languages such as Node.js and PHP.

**Data tier:** Responsible for storing and manipulating application data. Typical operation include, updating, deleting and searching existing records. Typically uses a Database Management system (**DBMS**). Examples include MySQL and PostgreSQL.

**States:**
Validating and conducting money transfer
Validating coupon codes and applying discounts

**Validating and conducting money transfers:**

1. User clicks “Confirm transfer” button.
2. Application queries database to confirm that the account balance can cover the transfer amount.
3. Database responds to the query
   If the amount is within the account limits, it will conduct the transaction.
   If the amount is beyond the account limits, the application shows an error message.

**Validating coupon codes and applying discounts:**

1. User enters coupon code.
2. Application queries database to determine whether the coupon is valid and whether any constraints exist.
3. Database responds with validity and constraints.
   The discount is applied if the code is valid and there is no constraints.
   An error message is displayed if the code is invalid or there are constraints.

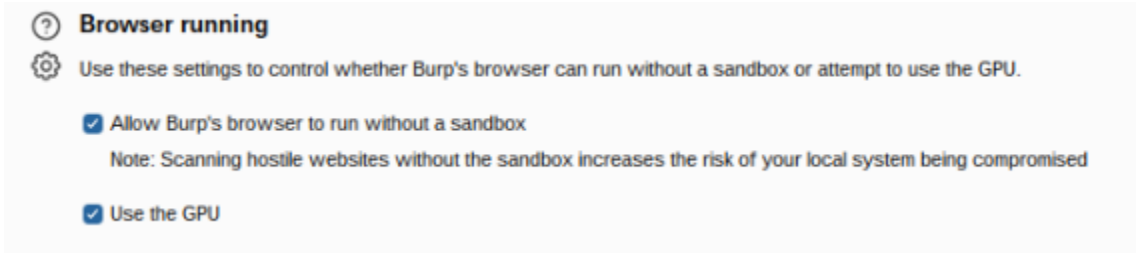
# EXPLOITING RACE CONDITIONS PRACTICAL

We have our orders again, to use 2 accounts to transfer the money between the two, using the repeater and trying to achieve a total sum in one of the accounts £100. I am still quite green with Burp, but I am eager to do this.

Our necessary credentials provided to us by THM:


First things first, launch Burp, and get FoxyProxy up and running. First we need to intercept, otherwise we will not have a target and this will be pretty much impossible! But by intercepting our target HTML we can ensure we are attacking the correct information, I like to think of it as grabbing a webpage.

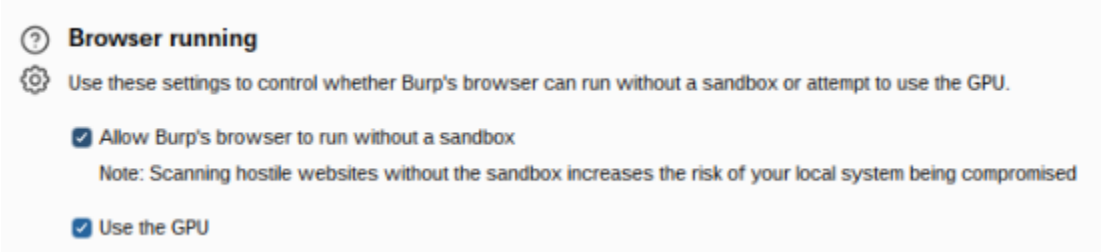
We have also been instructed to run Burp without a sandbox, which we can do in the settings.



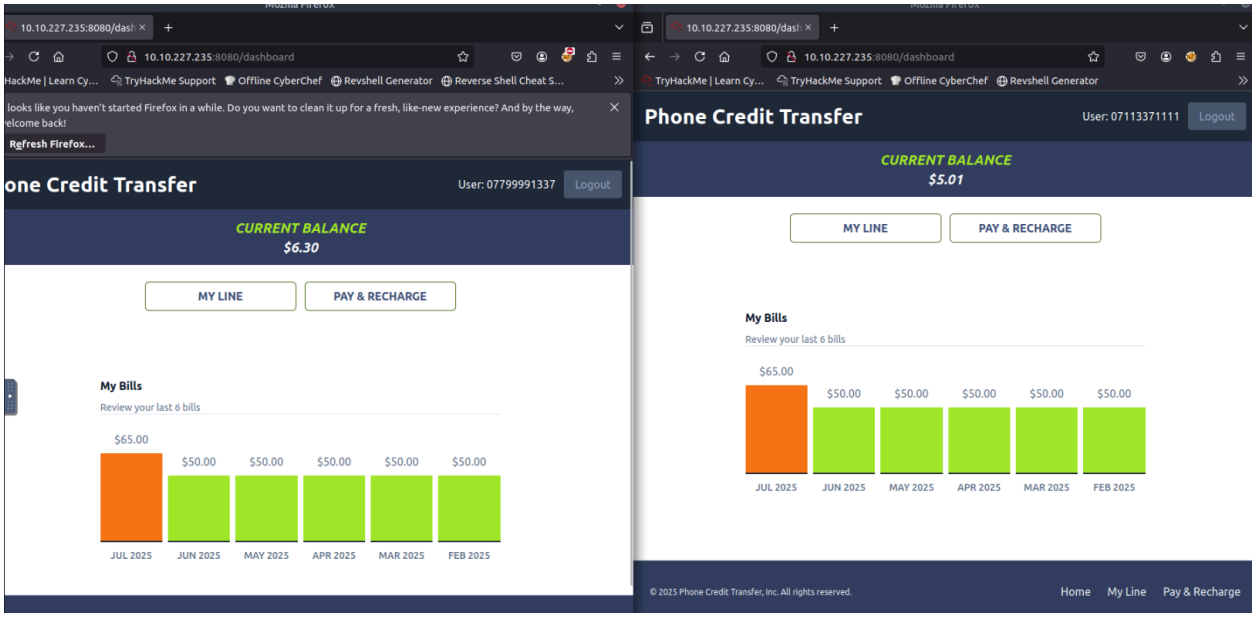

And there we go! I can close that down and now continue.

What I chose to do first was to log into account 1 just to take a look around, after all one thing I have learnt over my short time on THM is to obtain as much information prior to carrying out an attack, because you may just expose weaknesses just be exploring what is right in front of you, which can enhance your attack scope.

Right! This is quite useful, we have billing information, “my line” and “pay & recharge” I like pay and recharge, playing with money real or fake is always fun.

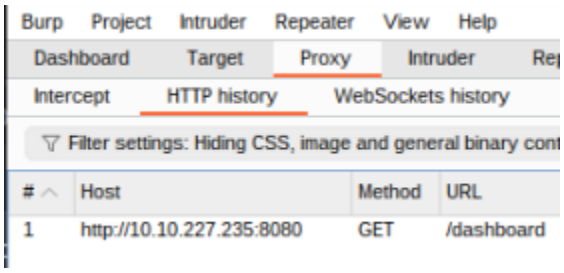



Firstly I was instructed to turn on the setting which allows Burps Browser to run without a sandbox.
So let's first break down the 2 accounts by logging in to them. Because there is 2 accounts I am going to open them both up simultaneously to inspect them both side by side.



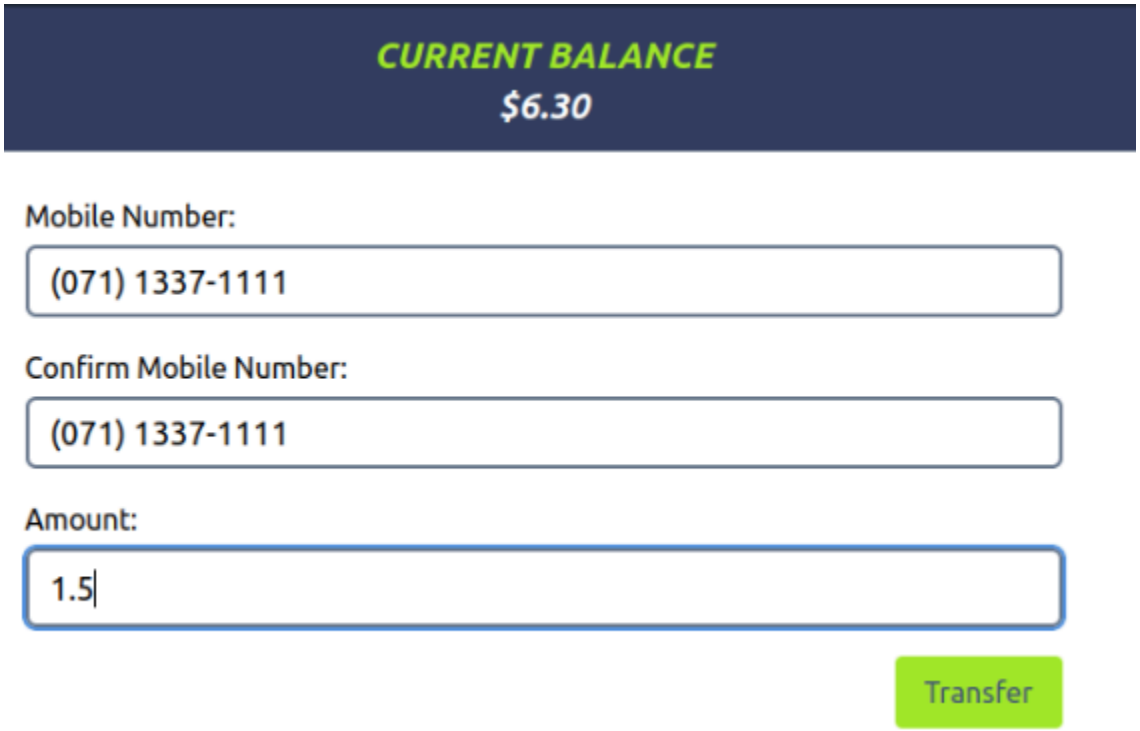

Two accounts totalling $11.31 combined total. We will see if I can exploit this.

So firstly



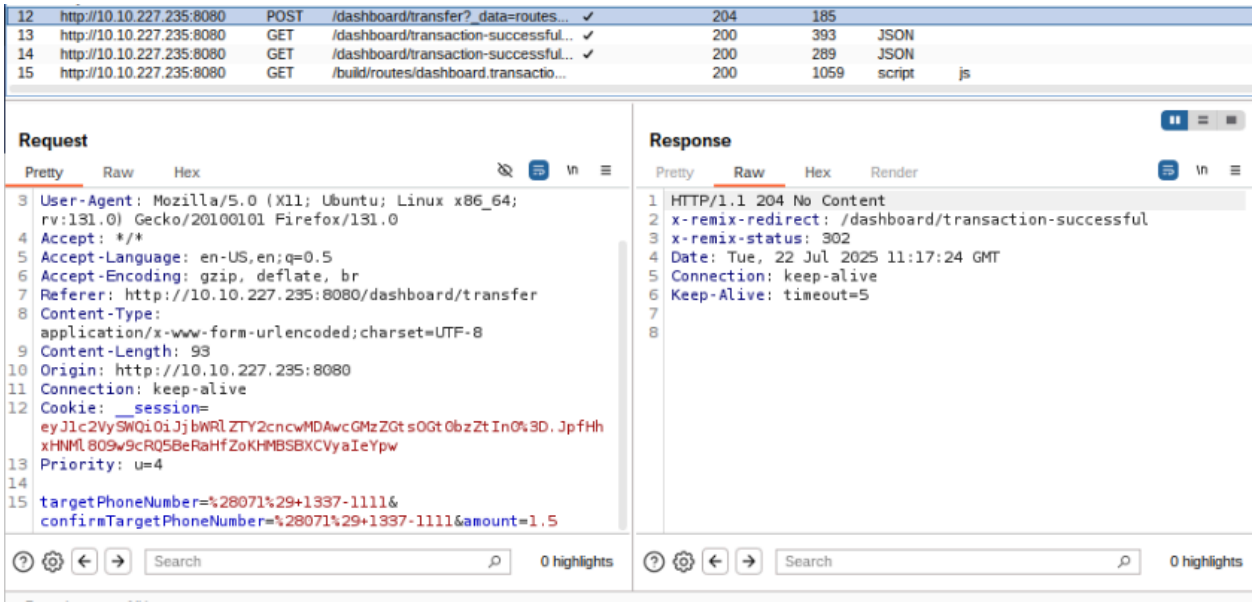

I will focus on the HTTP history and try and prompt a "POST" request because that will allow me to see the transmission from the server instead! Which is able to be exploited.

So to do that, I will send money from account 1 to account 2. Sending $1.50 as per the same instruction of THM.



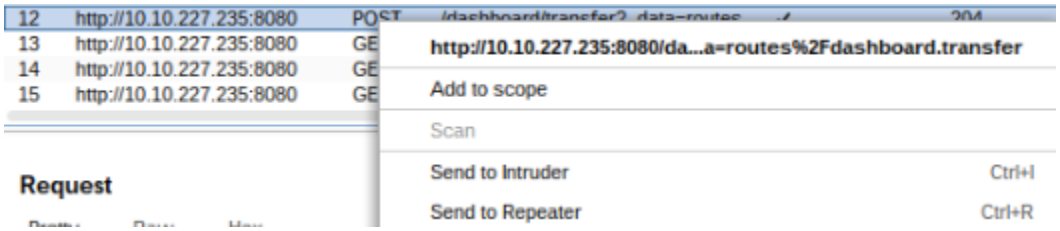

As we can see here, we are in account 1, sending money to account 2 with account twos information and the value in which we are sending, hopefully in doing this it provides a response POST request on the HTTP history which we can navigate in a much finer scope.


Excellent, so the transaction went through and as we can see we went from $6.30, to $4.80. So it's successfully registered on account ones balance.

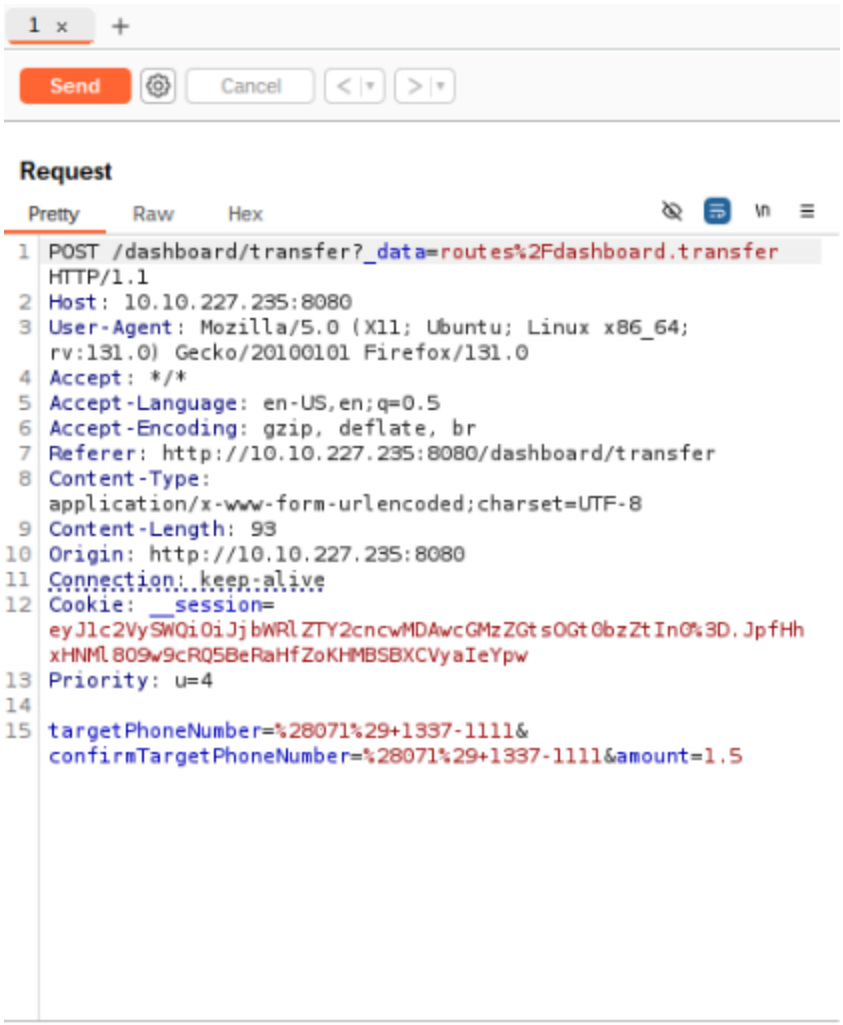



On Burp Suite navigating the the HTTP history I can now see a post request with the details / information of the recent transaction, the target phone number being the account 2, and the value of which I am sending which is 1.5 or $1.50.

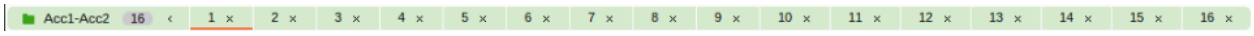
So in order to manipulate this request I will right click the POST request and send to repeater.



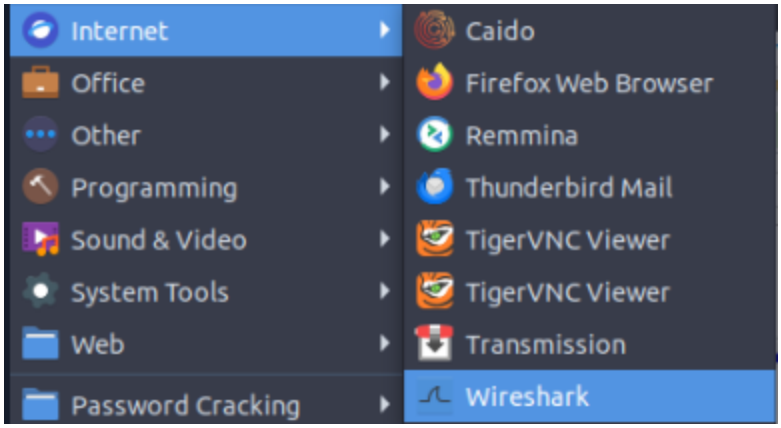

By following the hot-key short cut this is “Ctrl+R”.

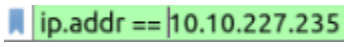



Now it is in the repeater I can now see that I have one tab open, or “one sequence” which I could send on its own but that won’t really get us anywhere. So my aim now is to group it and duplicate the request with the intention of sending multiple requests at once to try overload the web servers processing abilities, with the hope more than 1 request makes it through, thus increasing the amount of money we have sent in 1 request.

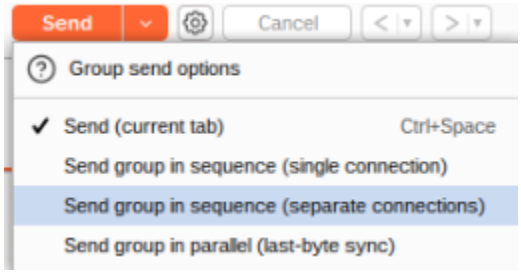



So I have called it “Acc1-Acc2” , representing it as account 1 sending account 2 money. I also duplicated the request 15 times. I am also going to go to wireshark and monitor the traffic so I can see how the web server behaves when the request is sent.




Now that is configured I will now prepare to send the requests, but! Select the drop down menu to see the options we have in which to send the requests.




Sending the group in sequence provides two options:

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

**Send Group in Sequence over a Single Connection**

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

**Send Group in Sequence over Separate Connections**

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

## Send Request Group in Parallel

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

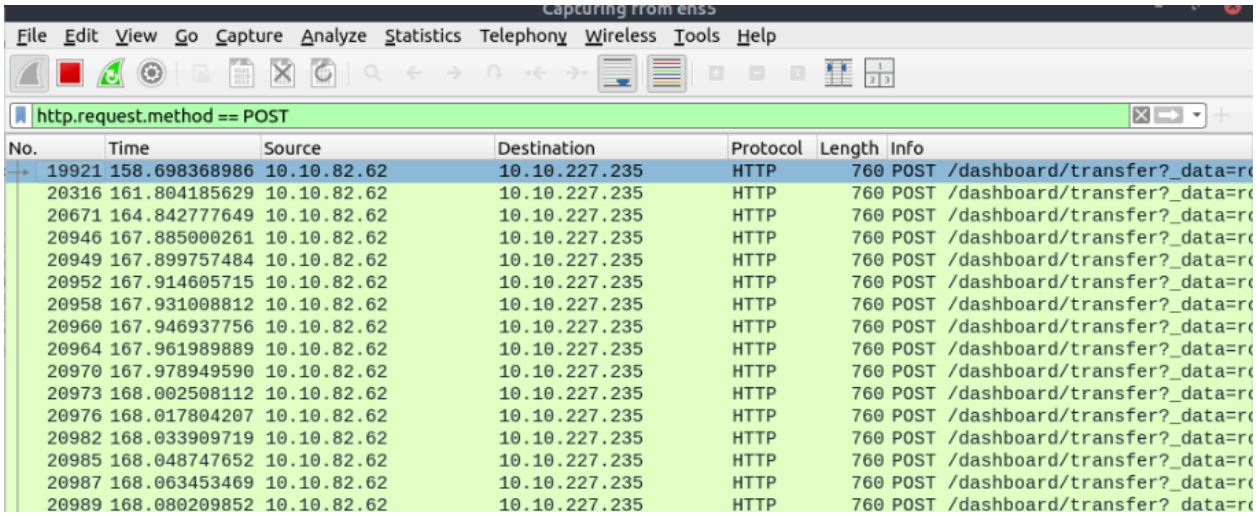
To start with, I am going to send the requests grouped over a single connection then I will click "send group (single connection)".

**Response**

Pretty Raw Hex Render

```
1 HTTP/1.1 204 No Content
2 x-remix-redirect: /dashboard/transaction-successful
3 x-remix-status: 302
4 Date: Tue, 22 Jul 2025 11:26:07 GMT
5 Connection: keep-alive
6 Keep-Alive: timeout=5
```

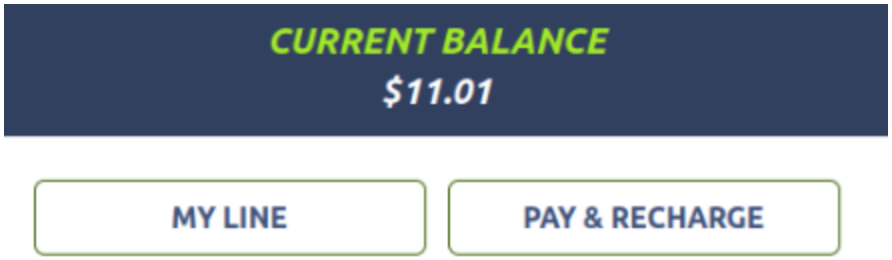
This response from the server tells us that the requests we send were successful, so now we can see on wireshark what happened.



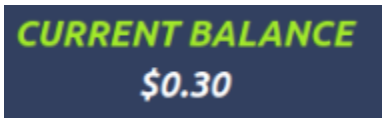

| No. | Time | Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|---|---|
| 19921 | 158.698368986 | 10.10.82.62 | 10.10.227.235 | HTTP | 760 | POST /dashboard/transfer?_data=r |
| 20316 | 161.804185629 | 10.10.82.62 | 10.10.227.235 | HTTP | 760 | POST /dashboard/transfer?_data=r |
| 20671 | 164.842777649 | 10.10.82.62 | 10.10.227.235 | HTTP | 760 | POST /dashboard/transfer?_data=r |
| 20946 | 167.885000261 | 10.10.82.62 | 10.10.227.235 | HTTP | 760 | POST /dashboard/transfer?_data=r |
| 20949 | 167.899757484 | 10.10.82.62 | 10.10.227.235 | HTTP | 760 | POST /dashboard/transfer?_data=r |
| 20952 | 167.914605715 | 10.10.82.62 | 10.10.227.235 | HTTP | 760 | POST /dashboard/transfer?_data=r |
| 20958 | 167.931008812 | 10.10.82.62 | 10.10.227.235 | HTTP | 760 | POST /dashboard/transfer?_data=r |
| 20960 | 167.946937756 | 10.10.82.62 | 10.10.227.235 | HTTP | 760 | POST /dashboard/transfer?_data=r |
| 20964 | 167.961989889 | 10.10.82.62 | 10.10.227.235 | HTTP | 760 | POST /dashboard/transfer?_data=r |
| 20970 | 167.978949590 | 10.10.82.62 | 10.10.227.235 | HTTP | 760 | POST /dashboard/transfer?_data=r |
| 20973 | 168.002508112 | 10.10.82.62 | 10.10.227.235 | HTTP | 760 | POST /dashboard/transfer?_data=r |
| 20976 | 168.017804207 | 10.10.82.62 | 10.10.227.235 | HTTP | 760 | POST /dashboard/transfer?_data=r |
| 20982 | 168.033909719 | 10.10.82.62 | 10.10.227.235 | HTTP | 760 | POST /dashboard/transfer?_data=r |
| 20985 | 168.048747652 | 10.10.82.62 | 10.10.227.235 | HTTP | 760 | POST /dashboard/transfer?_data=r |
| 20987 | 168.063453469 | 10.10.82.62 | 10.10.227.235 | HTTP | 760 | POST /dashboard/transfer?_data=r |
| 20989 | 168.080209852 | 10.10.82.62 | 10.10.227.235 | HTTP | 760 | POST /dashboard/transfer?_data=r |

Navigating / filtering down to just the POST requests I can see 16 are displayed. To be honest! I couldn't tell you what this means yet, I have very limited experience with WireShark but we can see something did happen! Which is good. I will now check the accounts for any updates and see what the values are. Remember before the values were

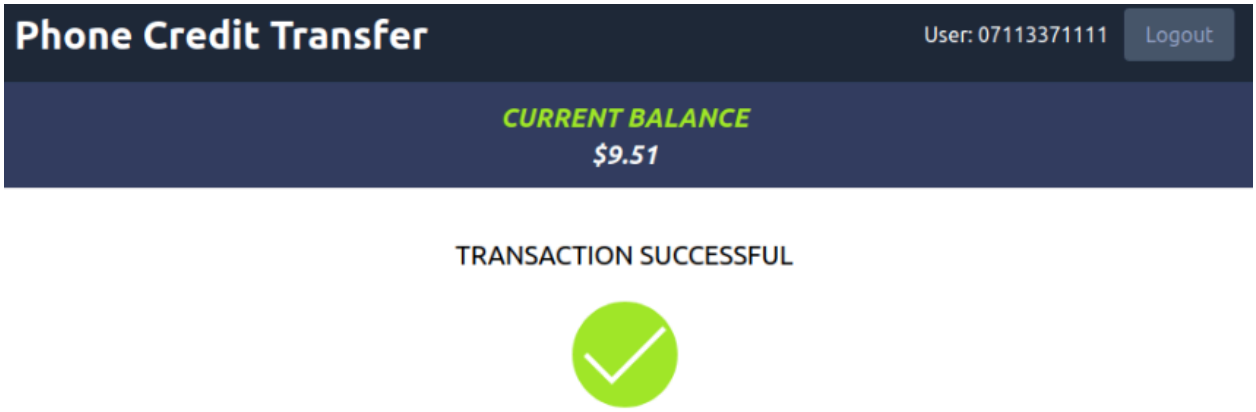
$4.80 for account 1 as money was send
$5.01 for account 2 as we never saw the updated amount. So I will log into account 2 and see what happened.



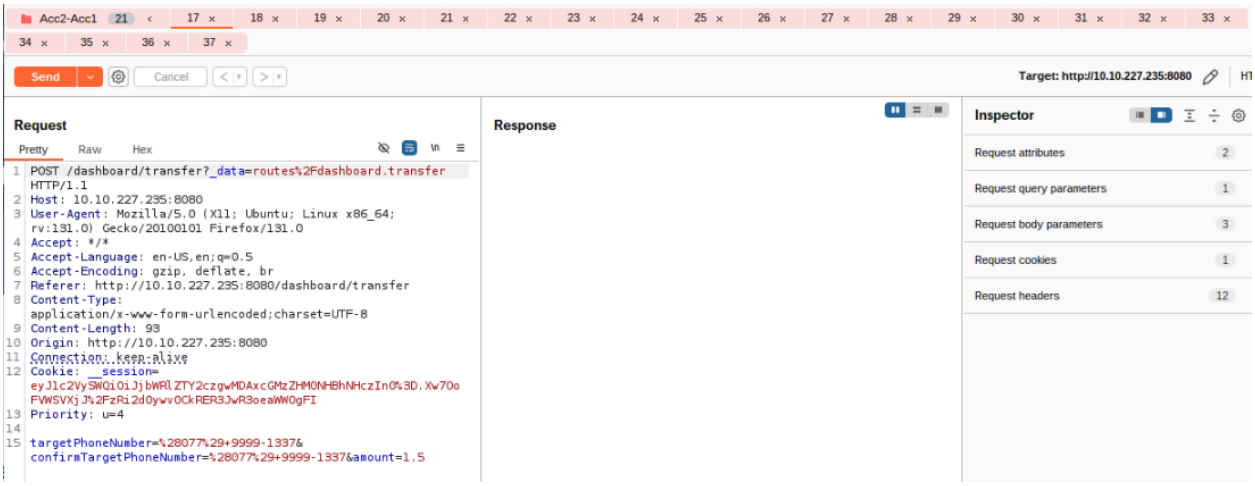

Success! Account 2 now has a total of $11.01. As we can see this was successful.
What I will do now is log into account 1 and see what happened in account 1.




So account 1 ended up broke! This is likely because I refreshed the web page, so if I go into account 2 again I will see we have a total of $11.01, which will now have to be the focus of the account because the other account doesn't have much money left. So this time I will do the same process but with account 2 to account 1 instead, this time a $1.50 over 20 sequences.



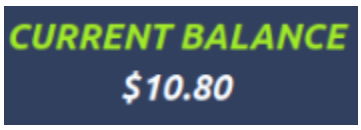

Starting to see a flaw with my idea, is that having 2 tabs open if I accidentally click the other account that is open the system will recognise it and only allow 1 account to be active at once, so I must be careful.



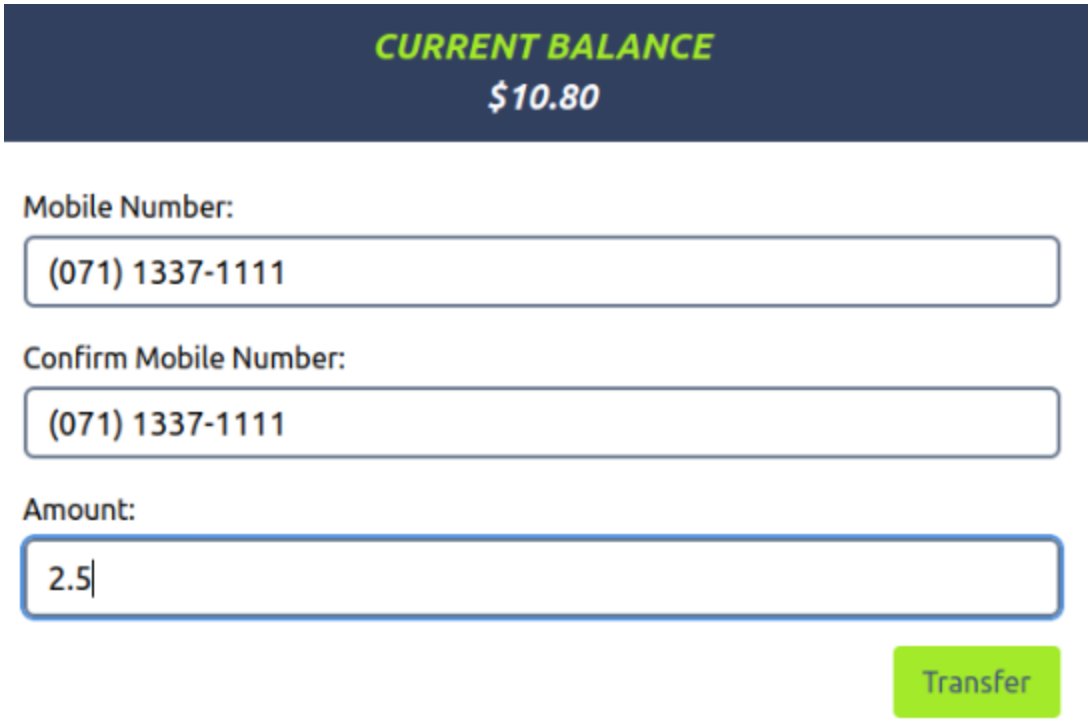

So here we are on the flip side, I will now select sending it over a single connection again and see what happens.

```
HTTP/1.1 204 No Content
x-remix-redirect: /dashboard/transaction-successful
x-remix-status: 302
Date: Tue, 22 Jul 2025 11:41:00 GMT
Connection: keep-alive
Keep-Alive: timeout=5
```

Let's now log into account 1 and see the updated balance. Remember I only send $1.50.

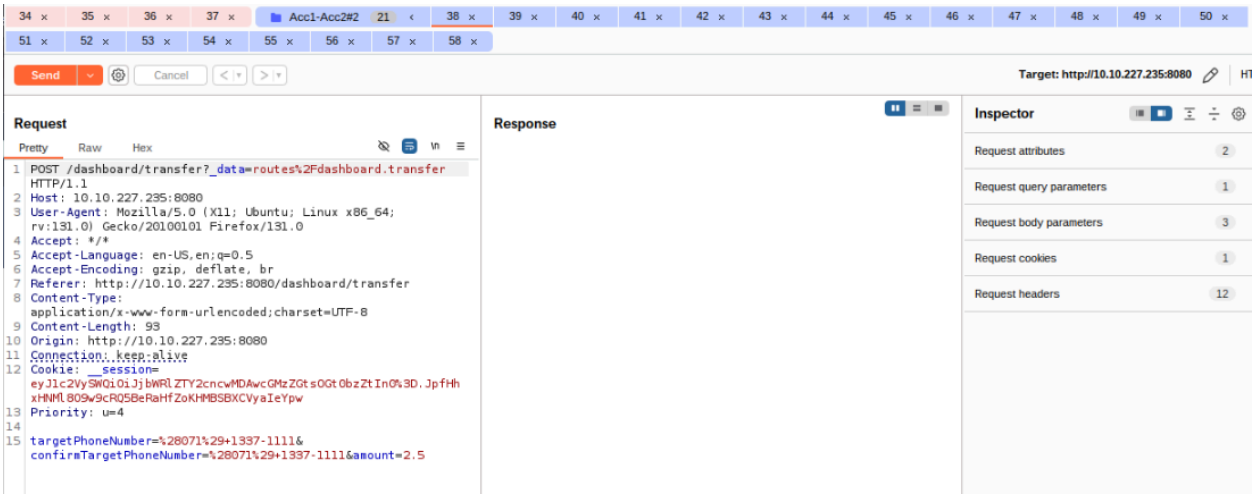



So we are making a difference. What I will not do is go back into account 2, because if I do, it may update and cause the whole thing to reverse. So what I will now do, is send a request of $2.50 this time back to account 2.





Now again, on let's find the recent HTTP POST request and try manipulating it again, 20 sequences in another group.

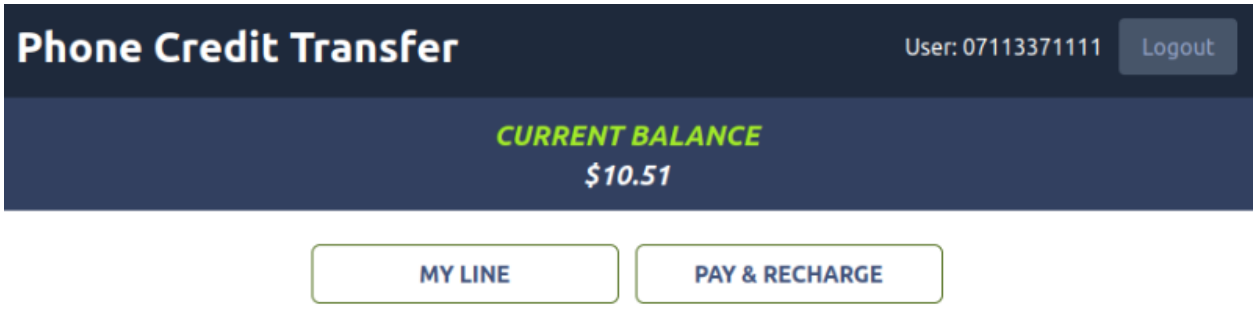



Here we go, this time I called this group Acc1-Acc2#2 so I know it is the second time I have done this, I am not closing the other tabs yet, just because I can keep track of what I have done as a way of documenting my progress and seeing if the next time I do this I could do it any differently, such as more quickly/efficiently.

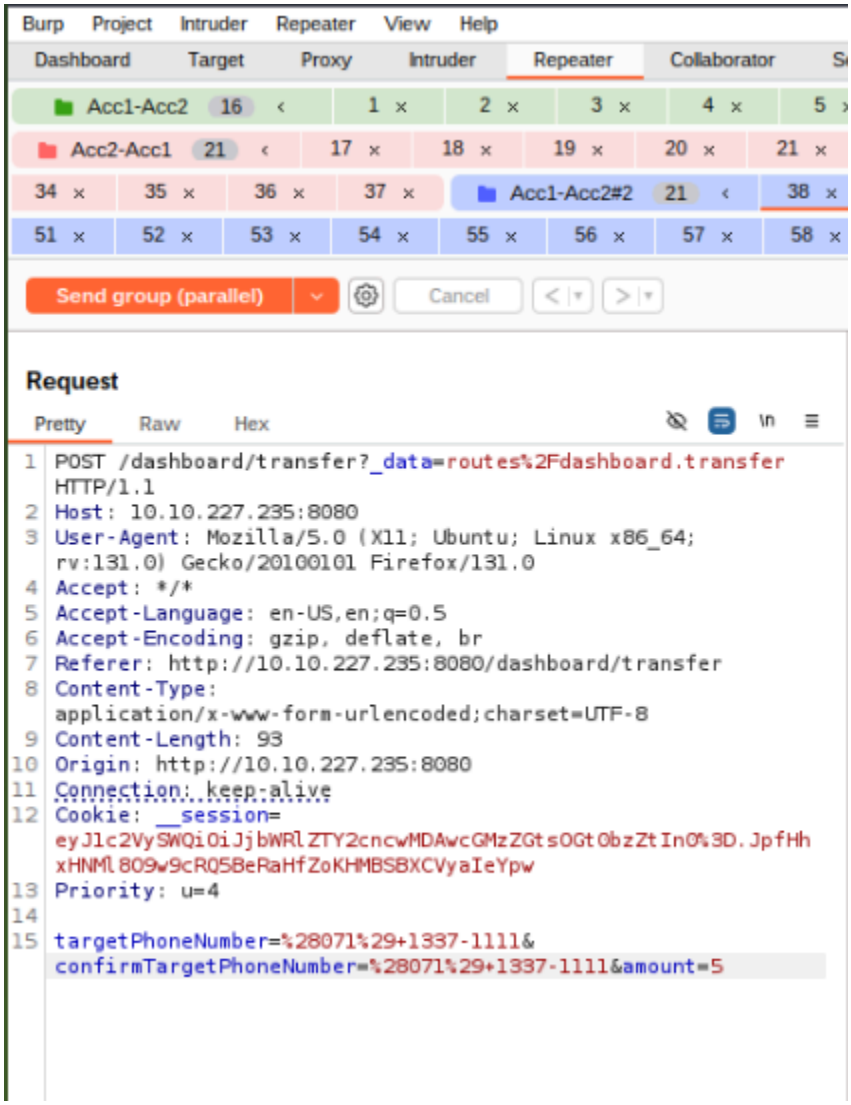
I send over a single connection again!

```
HTTP/1.1 204 No Content
x-remix-redirect: /dashboard/transaction-successful
x-remix-status: 302
Date: Tue, 22 Jul 2025 11:45:21 GMT
Connection: keep-alive
Keep-Alive: timeout=5


```

Let’s now log into account 2 and see what happened.

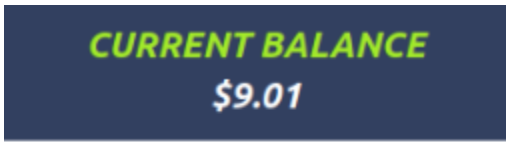


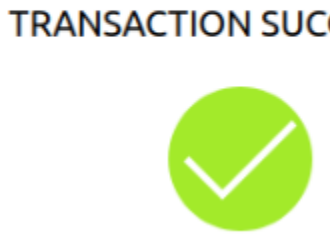

Right! So now much changed. This is very strange. But also exciting. It seems to have problems when it comes to more money it seems.
What I am going to do though is a bit of a risk.. I am going to reopen the same request I just made, but this time I am going to manually change the amount to 5 and send the group parallel too and see what happens.

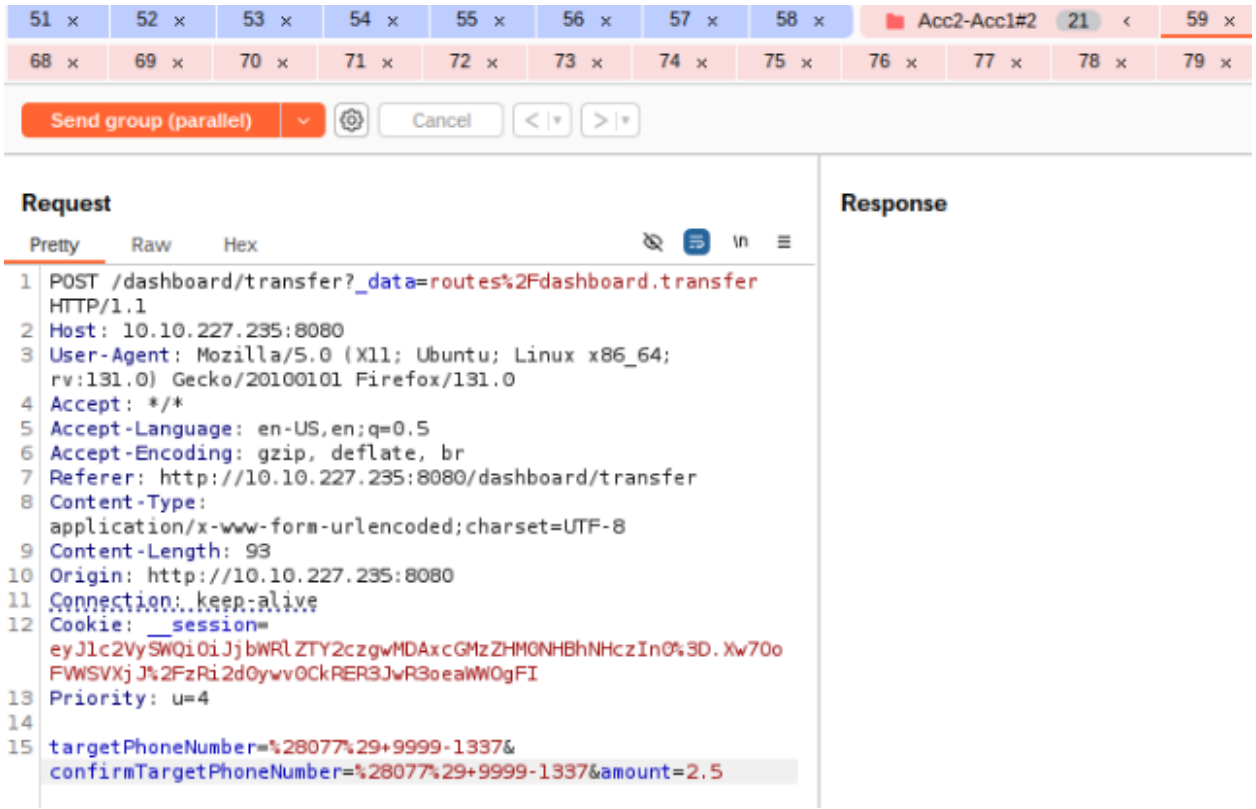


Declined due to insufficient balance, what if I changed it back to $2.5 hm… Insufficient balance again. Let's go back into account 2 and see what happened. Still $10.51, which means account 1 has no money left.

This time I will send account 1 money from account 2 but tweak it in the way I just tried. I will send a request of $1.50 and then manually change it and see what happens.



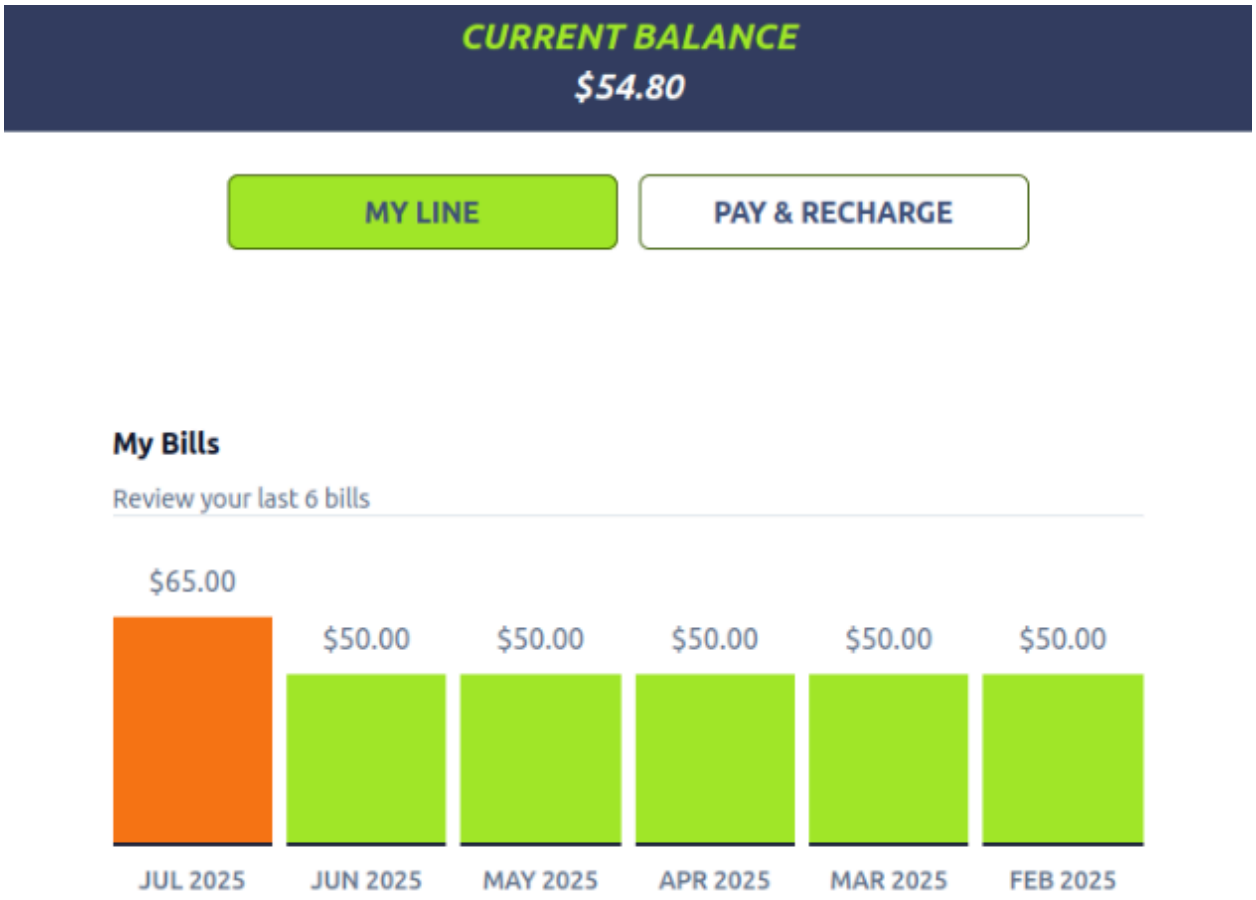



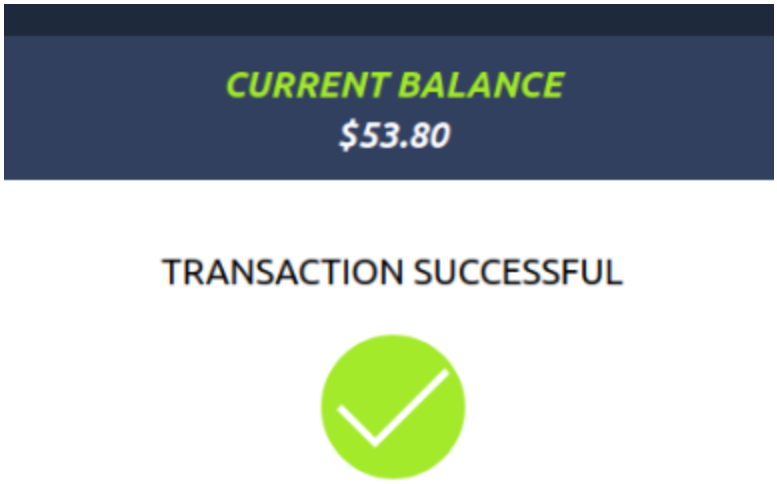
Let’s try it again and see what happens.

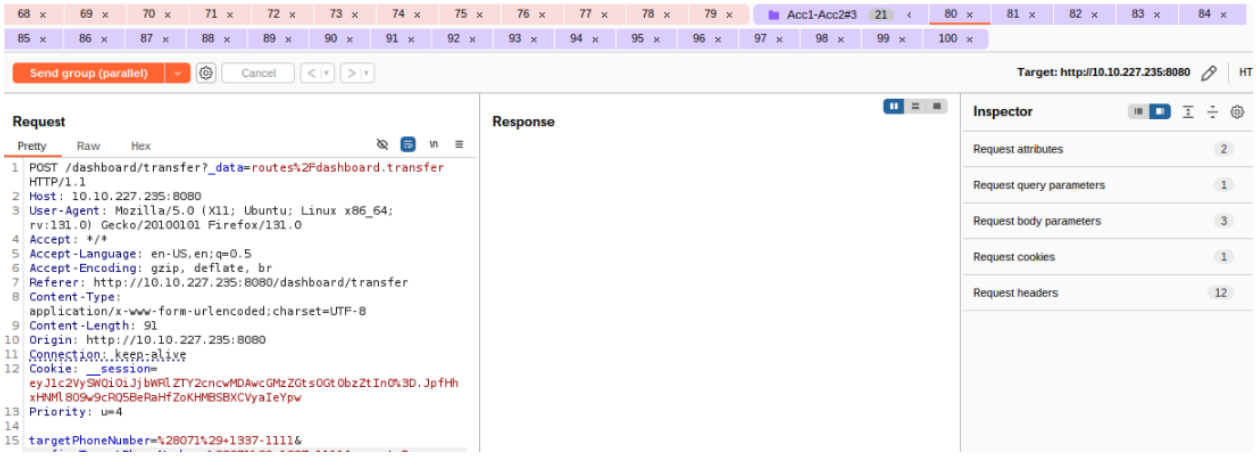
```
HTTP/1.1 204 No Content
x-remix-redirect: /dashboard/transaction-successful
x-remix-status: 302
Date: Tue, 22 Jul 2025 12:00:39 GMT
Connection: keep-alive
Keep-Alive: timeout=5

```

Let’s log into account 1 and see what happened.

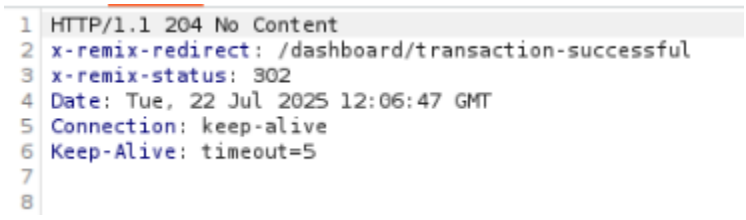



Now we are getting somewhere! So THIS is how we do it. This time I will send money back.. Again $1, but then manually change the value but still send 20 sequences.

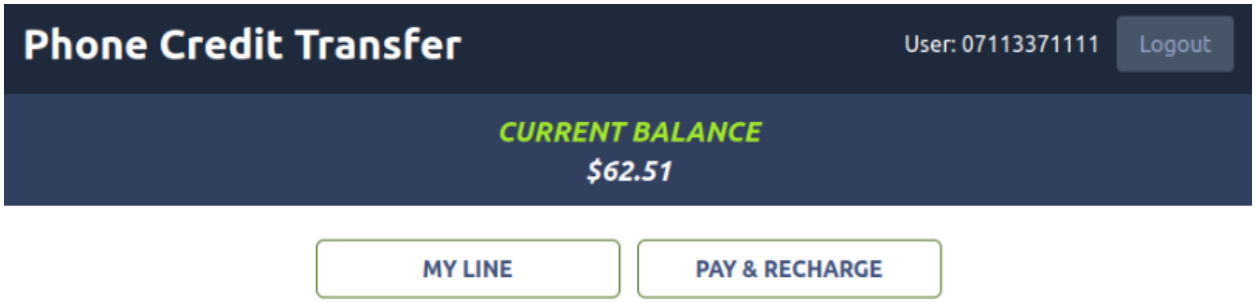


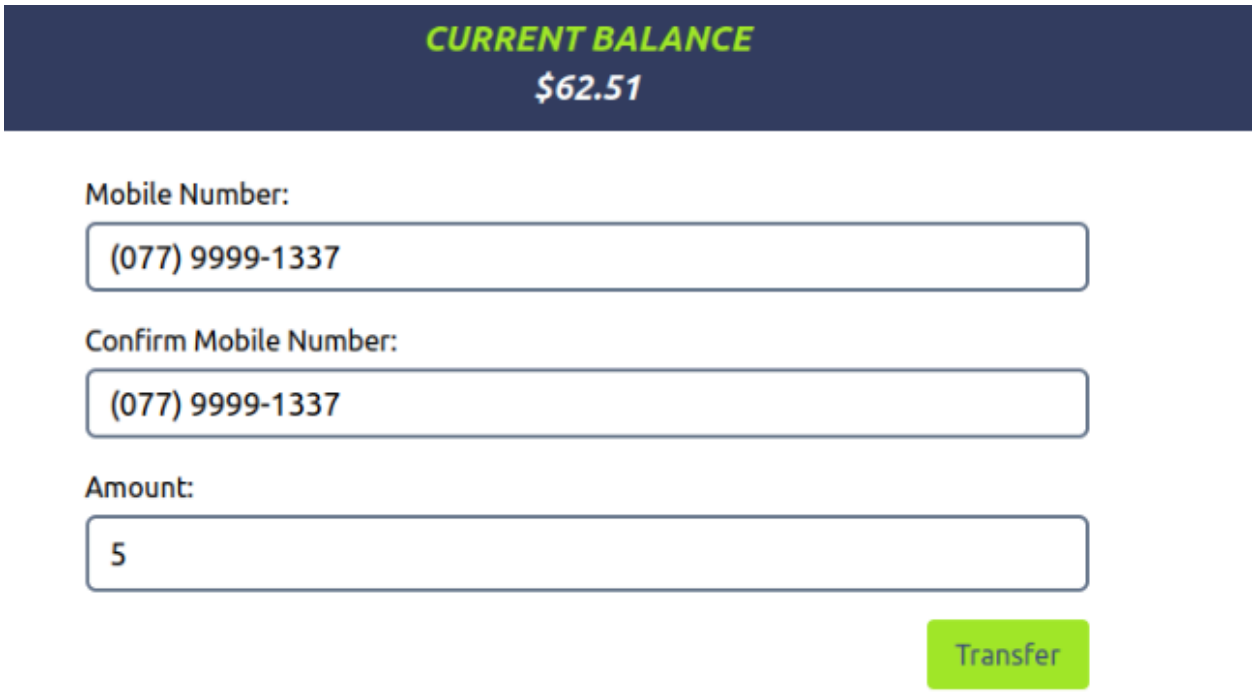

This time I have changed the amount to 5 and have still kept the 20 tabs again as per usual. This time I am sending the request to account 2. Let’s see what happens now.

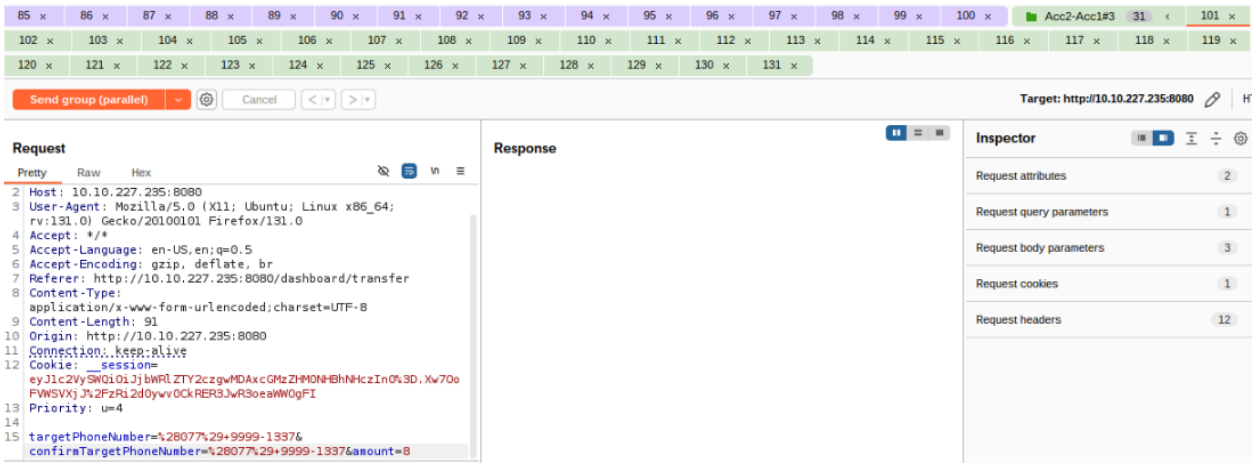



Let’s log into account 2 and see what happened.




So this is telling us it is certainly working whatever we are doing. I will up the stakes do the same process from account 2 back to account 1 but this time change it to $8 but 30 duplicates tabs and see what happens!

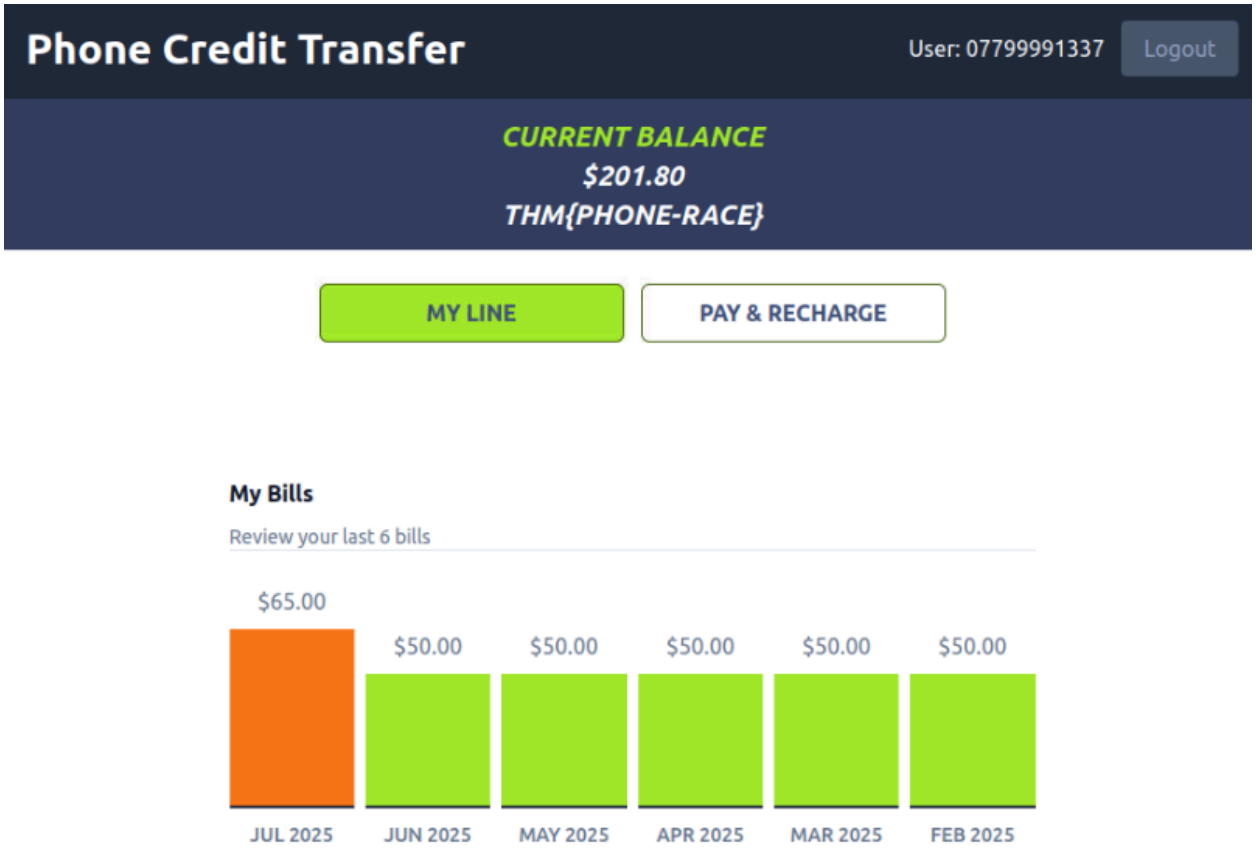


If this all goes well, it should mean that when I log into account 1 again I should have the $100 needed to pass the challenge. So let's see. Sending the parallel request again

```
HTTP/1.1 204 No Content
x-remix-redirect: /dashboard/transaction-successful
x-remix-status: 302
Date: Tue, 22 Jul 2025 12:12:53 GMT
Connection: keep-alive
Keep-Alive: timeout=5
```

Let's see.




WE DID IT!!!!
My sweet, sweet flag…
THM{PHONE-RACE}.

Conclusion:

This task was actually quite challenging, I did this task without really referring to the guide THM provided, and had very limited external support because I was so eager to crush this task on my own using my initiative to overcome obstacles. The truth is, I am quite proud, because I am freestyling to the point I know what I am doing to a degree, and experimenting and taking risks to achieve a better understanding of how processes are working, rather than just head hunting flags. These skills I have demonstrated, although I spend a substantial amount of time on this task, I am far better equipped to deal with it in future, in a much faster and efficient manner as well as demonstrate to others how to achieve the same results too.