

Exploring and Exploiting a Race Condition Vulnerability

Step 1: Launch the Environment

1. **Start the Machine:** Click the **Start Machine** button on the right to launch the attached VM.
2. **Start the AttackBox:** Click the **Start AttackBox** button at the top.
3. **Access the Target Application:** Using the AttackBox, navigate to `http://MACHINE_IP:8080`.

Step 2: Credentials for Login The following credentials are available for testing:

- **User 1:**
 - Mobile: 077999991337
 - Password: `pass1234`
 - **User 2:**
 - Mobile: 07113371111
 - Password: `pass1234`
-

Understanding the Target Application

The application is a credit transfer portal belonging to a mobile operator. The objective is to check for a race condition vulnerability by attempting to transfer more credit than the account holds.

1. **Log In:** Use either set of credentials to log in.
 2. **Initiate Credit Transfer:**
 - Navigate to the **Pay & Recharge** section and click **Transfer**.
 - Input the recipient's phone number and the transfer amount.
 - Experiment with both valid amounts (within the account balance) and invalid amounts (exceeding the balance).
-

Analyzing HTTP Requests Using Burp Suite

1. Open Burp Suite and set up a proxy:
 - Go to the **Intercept** tab and click **Open browser**.
 - If you encounter a sandbox error, navigate to **Settings > Burp's Browser** and enable the option to run without a sandbox.
2. Study HTTP requests:
 - Use Burp's browser to interact with the application.
 - Observe **POST** requests under the **HTTP history** tab.

Exploiting the Race Condition

Step 1: Send Request to Repeater

1. Right-click on a **POST** request in the **HTTP history** and select **Send to Repeater**.

Step 2: Duplicate Requests

1. Navigate to the **Repeater** tab.
 2. Create a **tab group**:
 - Click the **+** icon near the request tab and select **Create tab group**.
 - Assign a group name and include the current request.
 3. Duplicate the request:
 - Right-click the tab and choose **Duplicate tab** (or press **CTRL+R** multiple times).
 - Duplicate it 20 times for testing purposes.
-

Step 3: Send Requests

Burp Suite Repeater provides several methods to send requests:

Option 1: Send Group in Sequence

- **Single Connection**: Sends all requests in sequence over one connection, useful for detecting client-side desync issues.
- **Separate Connections**: Establishes and closes a new TCP connection for each request.

Results:

- Out of 21 requests, **five succeeded** and **sixteen were denied**. Successful requests took ~3 seconds each, while denied requests took ~4 ms.

Option 2: Send Group in Parallel

- Sends all requests simultaneously.

Results:

- All 21 requests succeeded, transferring credit.
 - Each request completed in ~3.2 seconds.
-

Wireshark Observations

1. **Sequence Sending:** Each request is sent as a single packet.
2. **Parallel Sending:**
 - Requests are synchronized to arrive simultaneously.
 - With HTTP/1, synchronization uses **last-byte withholding**, delaying the final byte of each request until all are ready.

By carefully analyzing and exploiting the system's handling of parallel requests, we successfully transferred credit multiple times, demonstrating the presence of a race condition vulnerability.

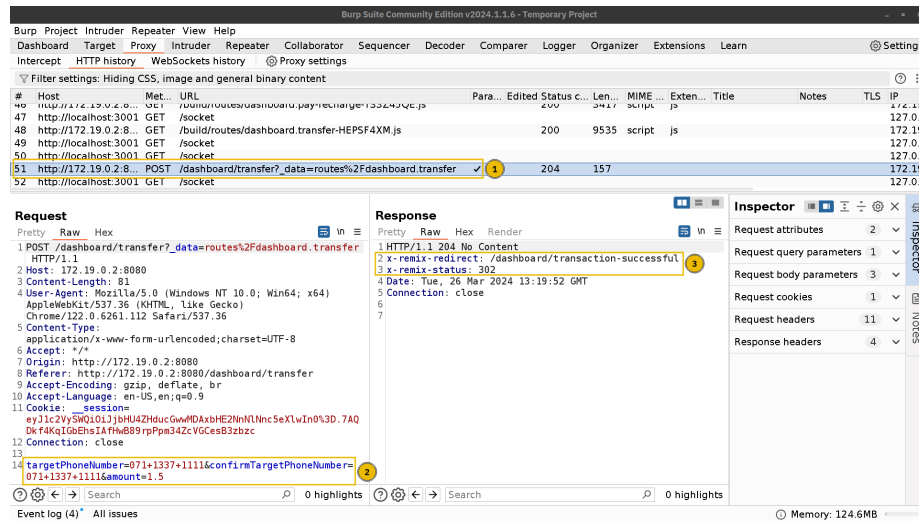


Figure 1: Burp Suite Proxy HTTP history showing an HTTP POST request and response.

Now that we have seen how the system reacts to valid and invalid requests, let's see if we can exploit a race condition. Right-click on the POST request you want to duplicate and choose **Send to Repeater**.

In the **Repeater** tab, as shown in the numbered screenshots below:

1. Click on the + icon next to the received request tab and select **Create tab group**
2. Assign a group name, and include the tab of the request you just sent to the importer before clicking **Create**
3. Right-click on the request tab and choose **Duplicate tab** (If this option is not available in your version, you can press **CTRL+R** multiple times instead)
4. As a starting point, we will duplicate it 20 times

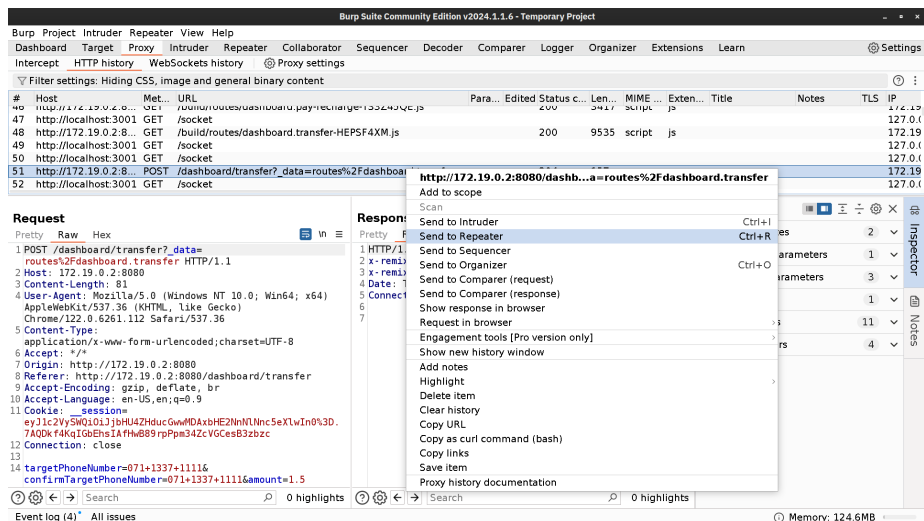


Figure 2: Burp Suite Proxy HTTP history showing an HTTP POST request being sent to Repeater.

- Next to the Send button, the arrow pointed downwards will bring a menu to decide how you want to send the duplicated requests

Next, we will exploit the target application by sending the duplicated request. Using the built-in options in Burp Suite Repeater, the drop-down arrow offers the following choices:

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

Sending Request Group in Sequence

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

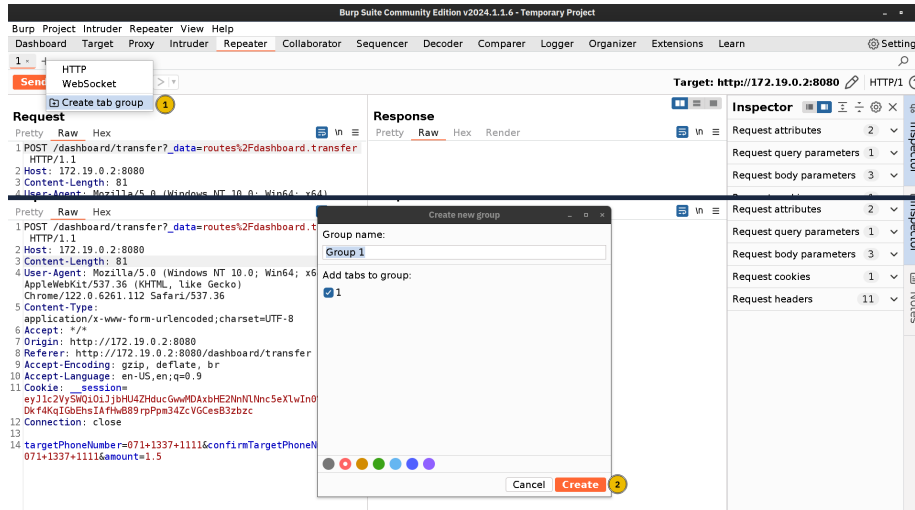


Figure 3: Creating a tab group in Burp Suite Repeater.

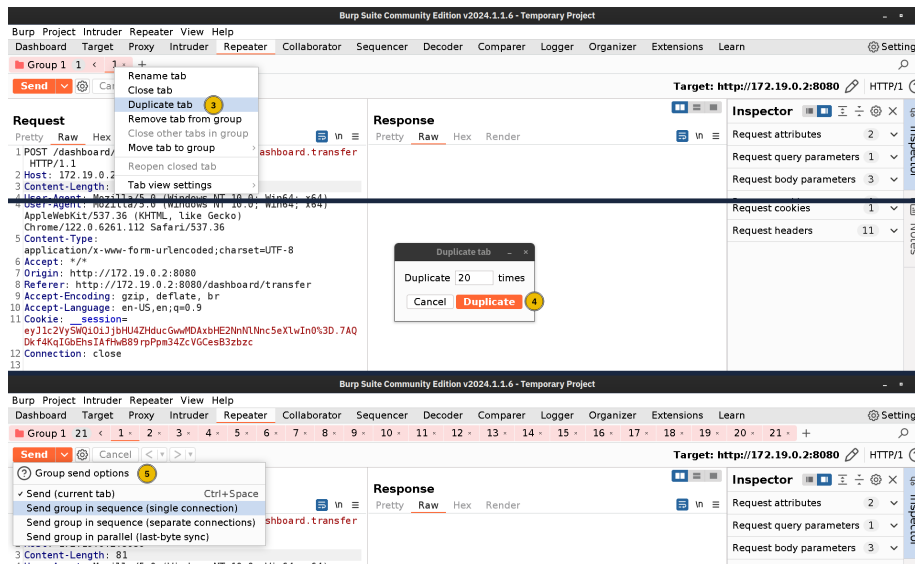


Figure 4: Duplicating a tab 20 times within a tab group in Burp Suite Repeater.

the subsequent request.

We tested this option to attack the web application. The screenshot below shows 21 TCP connections for the different POST requests in the group we sent.

- The first group (labelled 1) comprises five successful requests. We could confirm that they were successful by checking the respective responses. Furthermore, we noticed that each took around 3 seconds, as indicated by the duration (labelled 3).
- The second group (labelled 2) shows sixteen denied requests. The duration was around four milliseconds. It is interesting to check the Relative Start time as well.

Address A	Port A	Address B	Port B	Packets	Bytes	Stream ID	Total Packets	Percent Filtered	Packets A -> B	Bytes A -> B	Packets B -> A	Bytes B -> A	Rel Start	Duration	Bits/s A -> B
192.168.124.8	34318	172.18.0.2	8080	10	248	2	10	100.00%	5	148	5	495 bytes	12.048695	3.0378	2.738 bits/s
192.168.124.8	34332	172.18.0.2	8080	10	248	3	10	100.00%	5	148	5	495 bytes	15.070648	3.0280	2.729 bits/s
192.168.124.8	49942	172.18.0.2	8080	10	248	4	10	100.00%	5	148	5	495 bytes	18.302499	3.0292	2.728 bits/s
192.168.124.8	49978	172.18.0.2	8080	10	248	5	10	100.00%	5	148	5	495 bytes	21.135373	3.0273	2.729 bits/s
192.168.124.8	49980	172.18.0.2	8080	10	248	6	10	100.00%	5	148	5	495 bytes	24.165785	3.0305	2.726 bits/s
192.168.124.8	49984	172.18.0.2	8080	10	248	7	10	100.00%	5	148	5	522 bytes	27.250325	0.0053	
192.168.124.8	49990	172.18.0.2	8080	10	248	8	10	100.00%	5	148	5	522 bytes	27.210097	0.0047	
192.168.124.8	50002	172.18.0.2	8080	10	248	9	10	100.00%	5	148	5	522 bytes	27.217852	0.0044	
192.168.124.8	50016	172.18.0.2	8080	10	248	10	10	100.00%	5	148	5	522 bytes	27.225338	0.0039	
192.168.124.8	50032	172.18.0.2	8080	10	248	11	10	100.00%	5	148	5	522 bytes	27.232535	0.0049	
192.168.124.8	50046	172.18.0.2	8080	10	248	12	10	100.00%	5	148	5	522 bytes	27.240391	0.0043	
192.168.124.8	50062	172.18.0.2	8080	10	248	13	10	100.00%	5	148	5	522 bytes	27.247532	0.0040	
192.168.124.8	50064	172.18.0.2	8080	10	248	14	10	100.00%	5	148	5	522 bytes	27.254428	0.0046	
192.168.124.8	50074	172.18.0.2	8080	10	248	15	10	100.00%	5	148	5	522 bytes	27.261638	0.0039	
192.168.124.8	50086	172.18.0.2	8080	10	248	16	10	100.00%	5	148	5	522 bytes	27.268885	0.0042	
192.168.124.8	50098	172.18.0.2	8080	10	248	17	10	100.00%	5	148	5	522 bytes	27.276484	0.0045	
192.168.124.8	50090	172.18.0.2	8080	10	248	18	10	100.00%	5	148	5	522 bytes	27.353520	0.0035	
192.168.124.8	50096	172.18.0.2	8080	10	248	19	10	100.00%	5	148	5	522 bytes	27.363537	0.0042	
192.168.124.8	50100	172.18.0.2	8080	10	248	20	10	100.00%	5	148	5	522 bytes	27.369552	0.0042	
192.168.124.8	50104	172.18.0.2	8080	10	248	21	10	100.00%	5	148	5	522 bytes	27.378191	0.0039	
192.168.124.8	50112	172.18.0.2	8080	10	248	22	10	100.00%	5	148	5	522 bytes	27.391244	0.0042	

Figure 5: Wireshark showing 21 different POST requests; five are successful and the remaining ones are not successful.

The screenshot below shows the whole TCP connection for a request. We can confirm that the POST request was sent in a single packet.

No.	Time	Source	Destination	Protocol	Length	Info
181	15.32085	192.168.172.18.0	192.168.172.18.0	TCP	74	66 50100 -> 8080 [SYN] Seq=0 Win=32120 Len=0 MSS=1460 SACK_PERM TSval=1692094924 TSecr=0 Wd=128
182	15.32098	192.168.172.18.0	192.168.172.18.0	TCP	74	8080 -> 50100 [SYN, ACK] Seq=0 Ack=1 Win=31856 Len=0 MSS=1460 SACK_PERM TSval=144016941 TSecr=1692094924 Wd=128
183	15.32108	192.168.172.18.0	192.168.172.18.0	TCP	66	66 50100 -> 8080 [ACK] Seq=1 Ack=1 Win=32120 Len=0 TSval=1692094924 TSecr=144016941
184	15.32110	192.168.172.18.0	192.168.172.18.0	TCP	66	66 8080 -> 50100 [ACK] Seq=1 Ack=696 Win=31872 Len=0 TSval=144016941 TSecr=1692094924
185	15.32134	192.168.172.18.0	192.168.172.18.0	TCP	66	66 50100 -> 8080 [ACK] Seq=696 Ack=185 Win=32080 Len=0 TSval=1692094927 TSecr=144016944
186	15.32174	192.168.172.18.0	192.168.172.18.0	HTTP	256	HTTP/1.1 204 No Content
187	15.32181	192.168.172.18.0	192.168.172.18.0	TCP	66	66 8080 -> 50100 [FIN, ACK] Seq=185 Ack=696 Win=31872 Len=0 TSval=144016944 TSecr=1692094927
188	15.32186	192.168.172.18.0	192.168.172.18.0	TCP	66	66 50100 -> 8080 [FIN, ACK] Seq=696 Ack=186 Win=32080 Len=0 TSval=1692094928 TSecr=144016944
189	15.32510	192.168.172.18.0	192.168.172.18.0	TCP	66	66 8080 -> 50100 [ACK] Seq=186 Ack=697 Win=31872 Len=0 TSval=144016945 TSecr=1692094928

Figure 6: Wireshark showing the TCP connection related to a POST 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:

- In the Relative Start column, we notice that all 21 packets were sent within a window of 0.5 milliseconds (labelled 1).

- All 21 requests were successful; they resulted in a successful credit transfer. Each request took around 3.2 seconds to complete (labelled 2).

Address A	Port A	Address B	Port B	Packets	Bytes	Stream ID	Total Packets	Percent Filtered	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A	Rel. Start	Duration	Bits/s A → B
192.168.124.8	43900	172.18.0.2	8080	12	24B	2	12	100.00%	6	14B	6	561 bytes	3.234826	1.2033	2.747 bits/s
192.168.124.8	43901	172.18.0.2	8080	12	24B	3	12	100.00%	6	14B	6	561 bytes	3.234839	1.1894	2.759 bits/s
192.168.124.8	43916	172.18.0.2	8080	12	24B	4	12	100.00%	6	14B	6	561 bytes	3.234865	1.2178	2.734 bits/s
192.168.124.8	43920	172.18.0.2	8080	12	24B	5	12	100.00%	6	14B	6	561 bytes	3.234899	1.2532	2.905 bits/s
192.168.124.8	43934	172.18.0.2	8080	12	24B	6	12	100.00%	6	14B	6	561 bytes	3.234905	1.2318	2.722 bits/s
192.168.124.8	43940	172.18.0.2	8080	12	24B	7	12	100.00%	6	14B	6	561 bytes	3.234938	1.2287	2.717 bits/s
192.168.124.8	43948	172.18.0.2	8080	12	24B	8	12	100.00%	6	14B	6	561 bytes	3.234947	1.1267	2.514 bits/s
192.168.124.8	43958	172.18.0.2	8080	12	24B	9	12	100.00%	6	14B	6	561 bytes	3.234969	1.3331	2.808 bits/s
192.168.124.8	43970	172.18.0.2	8080	12	24B	10	12	100.00%	6	14B	6	561 bytes	3.234992	1.3751	2.771 bits/s
192.168.124.8	43986	172.18.0.2	8080	12	24B	11	12	100.00%	6	14B	6	561 bytes	3.234999	1.3477	2.795 bits/s
192.168.124.8	44000	172.18.0.2	8080	12	24B	12	12	100.00%	6	14B	6	561 bytes	3.237021	1.3678	2.777 bits/s
192.168.124.8	44006	172.18.0.2	8080	12	24B	13	12	100.00%	6	14B	6	561 bytes	3.237045	1.3208	2.740 bits/s
192.168.124.8	44018	172.18.0.2	8080	12	24B	14	12	100.00%	6	14B	6	561 bytes	3.237049	1.1396	2.602 bits/s
192.168.124.8	44020	172.18.0.2	8080	12	24B	15	12	100.00%	6	14B	6	561 bytes	3.237092	1.2245	2.729 bits/s
192.168.124.8	44040	172.18.0.2	8080	12	24B	16	12	100.00%	6	14B	6	561 bytes	3.237104	1.1543	2.789 bits/s
192.168.124.8	44044	172.18.0.2	8080	12	24B	17	12	100.00%	6	14B	6	561 bytes	3.237136	1.2460	2.713 bits/s
192.168.124.8	44044	172.18.0.2	8080	12	24B	18	12	100.00%	6	14B	6	561 bytes	3.237140	1.2592	2.700 bits/s
192.168.124.8	44072	172.18.0.2	8080	12	24B	19	12	100.00%	6	14B	6	561 bytes	3.237190	1.3407	2.784 bits/s
192.168.124.8	44078	172.18.0.2	8080	12	24B	20	12	100.00%	6	14B	6	561 bytes	3.237195	1.2659	2.694 bits/s
192.168.124.8	44080	172.18.0.2	8080	12	24B	21	12	100.00%	6	14B	6	561 bytes	3.237208	1.3956	2.753 bits/s
192.168.124.8	44090	172.18.0.2	8080	12	24B	22	12	100.00%	6	14B	6	561 bytes	3.238278	1.3402	2.765 bits/s

Figure 7: Wireshark showing 21 requests sent in parallel and at the same time.

By paying close attention to the screenshot above, we notice that each request led to 12 packets; however, in the previous attempt (send in sequence), we see that each request required only 10 packets. Why did this happen?

According to Sending Grouped HTTP Requests documentation, when sending in parallel, Repeater implements different techniques to synchronize the requests' arrival at the target, i.e., they arrive within a short time frame. The synchronization technique depends on the HTTP protocol being used:

- In the case of HTTP/2+, the Repeater tries to send the whole group in a single packet. In other words, a single TCP packet would carry multiple requests.
- In the case of HTTP/1, the Repeater resorts to last-byte synchronization. This trick is achieved by withholding the last byte from each request. Only once all packets are sent without the last-byte are the last-byte of all the requests sent. The screenshot below shows our POST request sent over two packets.

No.	Time	Source	Destination	Protocol	Length	Info
19	0.080369	192.168.0.1	172.18.0.2	TCP	74	44076 → 8080 [SYN] Seq=0 Win=32768 Len=0 MSS=1460 SACK_PERM TSval=1692642613 TSecr=0 WS=128
87	0.092169	172.18.0.2	192.168.0.1	TCP	74	8080 → 44076 [ACK] Seq=1 Ack=1 Win=32768 Len=0 MSS=1460 SACK_PERM TSval=144564827 TSecr=1692642613 WS=128
74	0.091177	192.168.0.1	172.18.0.2	TCP	66	44076 → 8080 [ACK] Seq=1 Ack=1 Win=32768 Len=0 TSval=1692642614 TSecr=144564827
104	0.091699	192.168.0.1	172.18.0.2	TCP	718	44076 → 8080 [POST] Seq=1 Ack=1 Win=32768 Len=0 TSval=1692642614 TSecr=144564827 (TCP segment of a reassembled PDU)
159	0.091544	172.18.0.2	192.168.0.1	TCP	66	8080 → 44076 [ACK] Seq=1 Ack=696 Win=32768 Len=0 TSval=144564827 TSecr=1692642614
145	0.103117	192.168.0.1	172.18.0.2	HTTP	67	POST /dashboard/transfer?data=routes&id=dashboard/transfer HTTP/1.1 (application/x-www-form-urlencoded)
147	0.103541	172.18.0.2	192.168.0.1	TCP	66	8080 → 44076 [ACK] Seq=1 Ack=697 Win=32768 Len=0 TSval=144564829 TSecr=1692642616
240	3.053787	172.18.0.2	192.168.0.1	HTTP	223	HTTP/1.1 204 No Content
249	3.265835	192.168.0.1	172.18.0.2	TCP	66	44076 → 8080 [ACK] Seq=697 Ack=158 Win=32000 Len=0 TSval=1692646078 TSecr=144568091
250	3.266099	172.18.0.2	192.168.0.1	TCP	66	8080 → 44076 [FIN, ACK] Seq=158 Ack=697 Win=32768 Len=0 TSval=144568092 TSecr=1692646078
251	3.266214	192.168.0.1	172.18.0.2	TCP	66	44076 → 8080 [FIN, ACK] Seq=697 Ack=159 Win=32000 Len=0 TSval=1692646079 TSecr=144568092
252	3.266287	172.18.0.2	192.168.0.1	TCP	66	8080 → 44076 [ACK] Seq=159 Ack=698 Win=31872 Len=0 TSval=144568092 TSecr=1692646079

Figure 8: Wireshark showing the TCP connection related to a POST request when using last-byte synchronization.