



# UNIVERSITÀ DI TRENTO

Department of Information Engineering  
and Computer Science

Offensive Technologies 2021/2022

## GROUP 3 - FINAL REPORT

Lorenzo Cavada

`<lorenzo.cavada@studenti.unitn.it>`

Matteo Franzil

`<matteo.franzil@studenti.unitn.it>`

Dmytro Kashchuk

`<dmytro.kashchuk@studenti.unitn.it>`

Tommaso Sacchetti

`<tommaso.sacchetti@studenti.unitn.it>`

# Contents

<b>1</b>	<b>Executive summary</b>	<b>1</b>
<b>2</b>	<b>CCTF Resilient Server</b>	<b>2</b>
2.1	Scenario . . . . .	2
2.2	Environment setup . . . . .	2
2.3	Attack preparation . . . . .	3
2.3.1	Goals . . . . .	3
2.3.2	Tools used . . . . .	3
2.3.3	Strategy overview . . . . .	4
2.4	Defense preparation . . . . .	4
2.4.1	Goals . . . . .	4
2.4.2	Strategy overview . . . . .	4
2.5	Execution . . . . .	5
2.5.1	Attack . . . . .	5
2.5.2	Defense . . . . .	6
<b>3</b>	<b>CCTF Secure Server</b>	<b>7</b>
3.1	Scenario . . . . .	7
3.2	Environment setup . . . . .	7
3.3	Attack preparation . . . . .	7
3.3.1	Goals . . . . .	7
3.3.2	Tools used . . . . .	7
3.3.3	Strategy overview . . . . .	8
3.4	Defense preparation . . . . .	8
3.4.1	Goals . . . . .	8
3.4.2	Strategy overview . . . . .	9
3.5	Execution . . . . .	10
3.5.1	Attack . . . . .	10
3.5.2	Defense . . . . .	11
<b>4</b>	<b>Team roles</b>	<b>12</b>
	<b>References</b>	<b>13</b>
	<b>Attachments</b>	<b>14</b>
A	List of experiment networks . . . . .	14
B	Example of phishing email . . . . .	14

# 1 Executive summary

Modern network systems are nowadays increasingly subject to attacks. As their techniques evolve and become more complex, so must the defenses of the targeted systems. In the two CCTFs, we used our technical, coding, and team skills to simultaneously coordinate a network attack and protect from it in a realistic scenario.

As a red team, we leveraged our coding and research capabilities to build a vast arsenal of tools. These included both self-written and open-source software. With such weapons, we planned our attacks, dividing the work between members. We developed a strategy encompassing the first stages of penetration testing from pre-engagement to exploitation. In the live CCTFs, we were able to effectively stress the capabilities of the defending team, sometimes even succeeding with our efforts.

As a blue team, we focused on building a system that could withstand enemy attacks while providing a crystal-clear overview of its state, promptly informing us if an attack were to occur. We did extensive research on the features offered by the Linux environment, modifying the TCP/IP stack configuration and developing robust and specific firewall rules. We employed state-of-the-art tools such as Varnish and Nginx, and we wrote custom tools for displaying real-time information on traffic, requests, and system load.

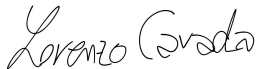
---

Work submitted in partial fulfillment for the course of *Offensive Technologies - University of Trento* - A.Y. 2021/2022.

This report is original work, has been done by the undersigned students and has not been copied or otherwise derived from the work of others not explicitly cited and quoted.

The undersigned students are aware that plagiarism is an academic offense whose consequences include failure of the exam.

Lorenzo Cavada  
220502



Matteo Franzil  
221214



Dmytro Kashchuk  
221228



Tommaso Sacchetti  
223028



## 2 CCTF Resilient Server

### 2.1 Scenario

The *red team* (or *attacking team*) had control over the three clients shown on the right of Figure 1. Instead, the *blue team* (or *defending team*) managed the *server* and the *gateway*.

The *server*'s only function was to provide ten static HTML pages while the *gateway* acted as a middleman between the server and the router (external network). On the other side of the *router*, the three *clients* were free to cause Denial of Service (*DoS*) to the server. The only requirement was that one of them had to behave as a legitimate user, performing not more than one request per second.

The *router* was not accessible by any team, and the Teaching Assistant (*TA*) controlled it. The operative system of all the machines was Ubuntu 18.04. Moreover, the testbed was isolated from the external network, with packages updates (*apt*) provided through a local repository.

The rules prohibited us from using the interfaces or the IPs of the DETERLab network. Both defense and attack were allowed only using the testbed interfaces and IPs shown in Figure 1, which were the same for both the red and the blue team [14]. Attachment A shows a complete list of the networks available in the experiment.

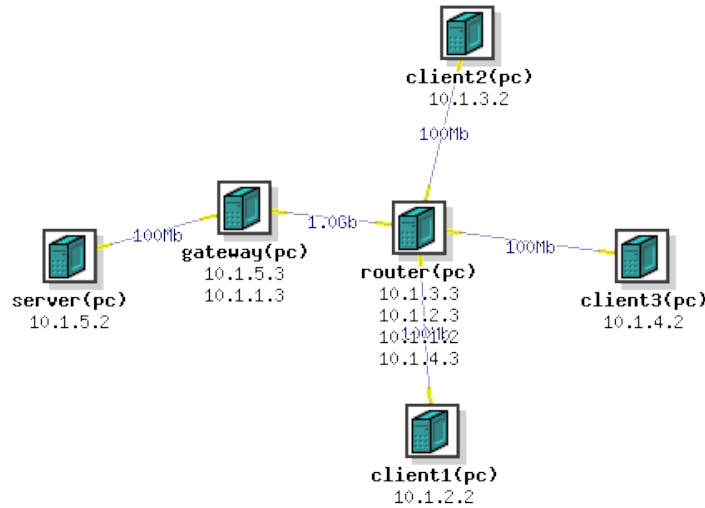


Figure 1: The full network diagram with interface IPs.  
DETERLab management interfaces are not shown.

### 2.2 Environment setup

We started the preparation phase by setting up a private GitHub repository containing all the tools, scripts, and materials needed for the CCTF.

Each member of the group cloned the repository on their DETERLab home directory, which is mounted as a Network File System (*NFS*) drive on the `users.deterlab.net` machine along with all the experiment machines.

To automate the setup process, we wrote bootstrap scripts that pulled and installed dependencies automatically and configured the machines accordingly. For the *gateway* and the *server* they mainly set up the firewall rules and configured the web server. For the *clients*, the script installed the few tools that were not already available on the repository.

## 2.3 Attack preparation

### 2.3.1 Goals

The red team had the following tasks:

- At the beginning, choose a client and declare it as “legitimate” to the Teaching Assistant, then set it up to act legitimately<sup>1</sup>.
- Use the remaining two clients to disrupt the web server’s Service Level Agreement by making it exhaust its resources, flood incoming links, and ultimately cause it to crash and be unable to serve any request.

### 2.3.2 Tools used

To organize the attack, we spent a lot of time looking for open source tools that could allow us to cause DoS to the blue team machines. One of the problems in this phase was the lack of a WAN connection in the testbed. While we were able to retrieve and update some packages using the local `apt` repository, we could not use tools like `pip` nor download anything directly to the machines. We partially solved this issue by fetching the source code of some tools along with their dependencies and compiling everything locally.

After discarding various ineffective or useless tools, we narrowed down our list to the following ones:

- **GoldenEye** [9]: GoldenEye is a now-deprecated, yet-useful open-source tool that employs maximum parallelism and sets up a customizable amount of threads and outbound sockets on the machine. These sockets repeatedly start pinging the target while abusing the HTTP Keep-Alive options to keep connections open and waste the target’s memory. Its downside is that it is nearly impossible to stop.
- **Hulk** [8]: Hulk is a Go/Python tool that spawns several threads for each connection and stresses the target server by repeatedly making concurrent HTTP requests and leaving the connections open, exhausting its resource pool.
- **SlowLoris** [7]: SlowLoris is a DoS tool that starts making a slew of parallel requests to the target server. Once the connections are open, it sends headers periodically to keep them alive, thus exhausting the servers’ thread pool and memory and bringing it to a halt.
- **Sockstress** [4]: a slightly different implementation of the SlowLoris tool. It not only keeps an empty connection open, but It also allows requesting arbitrary pages from a web server.
- **Targa3** [6]: a simple C program that generates packets with weird combinations of TCP and IP flags (e.g., invalid fragmentation, packet size, offsets, TCP segments) and starts flooding the target server with them.
- **hping3** [12]: a versatile Unix tool that allows elaborate packet crafting. It handles fragmentation, arbitrary packet body, and size. It also supports spoofing and can be used as a DoS tool, albeit with limitations.

An honorable mention goes to **loacker**. While testing **hping3**’s capabilities, we noticed that it was not good enough. Since our goal was to obfuscate the SYN flood, we wanted to craft packets and make them look like SYN packets of `curl` requests. Even if very flexible, **hping3** did not support setting TCP options such as `sack-permitted` or `NOP`. To overcome this limit, we built **loacker**: a small Go program, which employs maximum parallelism and uses raw sockets to flood the target with credible, `curl`-like SYN packets, and that also allows for source IP spoofing.

---

<sup>1</sup>“Legitimately” means a minimum of one request every ten seconds and a maximum of one every second.

We developed our tool to generate legitimate requests, merely called `legitimate`. It is a small `bash` script that continuously sends `curl` requests to the server while logging parameters such as response times and error codes. It also has an option to modify the random delay between requests.

### 2.3.3 Strategy overview

Before starting the CCTF, we outlined a four-part strategy, hoping to have a flexible and structured template that would allow us to focus our efforts on each phase, rather than just sending random attacks hoping for something to work.

- **Reconnaissance:** enumerate the vulnerabilities of the adversary team by checking their open ports, firewall rules, and the blueprint of the authorized traffic.
- **Focus:** after discovering the enemy team's configuration and key weaknesses (e.g., some bogus TCP combinations were allowed), choose the best attack plan.
- **Slow start:** to avoid wasting all of our options immediately, start attacking the enemy team with our weaker tools – in the hope that some misconfiguration could give us an early advantage.
- **Wafer attack:** launch our most powerful tools (`loacker` and `hping3`), hoping to catch the defending team unprepared and unable to respond to the attack promptly.

## 2.4 Defense preparation

### 2.4.1 Goals

The blue team only goal was to keep the server response time below 500 milliseconds (ms). As already mentioned in subsection 2.1, the web server had to provide ten static HTML pages, numbered from 1 to 10 and customized at the team's will.

### 2.4.2 Strategy overview

The defense strategy involved TCP/IP stack hardening on both server and gateway, installing Varnish [17], and configuring a two-level firewall. Moreover, we built ad hoc monitoring tools to examine the incoming network traffic.

#### Gateway

As the gateway was our first level of defense, we wanted to set up the firewall to block most of the unwanted traffic efficiently.

We used the `PREROUTING` chain of the `raw` and `mangle` tables to do this, both configured with a default `ACCEPT` policy. Instead, we set up a default `DROP` policy for the `INPUT`, `FORWARD`, and `OUTPUT` chains. The `raw` table filtering happened before connection tracking (`CONNTRACK`), making it less CPU intensive. We used it to filter out fragmented packets, bogus TCP flags, unusual Maximum Segment Size (MSS)<sup>2</sup> values in `SYN` packets, UDP and ICMP protocols. Moreover, we added an IP whitelist with the known IPs of the whole network.

The `mangle` table filtering happened after the connection tracking, allowing us to implement a stateful firewall. We used it to block `INVALID` packets, and new connections started with packets without the `SYN` flag.

In the `INPUT` and `OUTPUT` chains, we blocked almost everything that was coming from and going to the router – connections with the server or DETERLab were left untouched – while on the `FORWARD` chain, we allowed only TCP traffic directed to the server on port 80.

---

<sup>2</sup>The typical MSS of an IPv4 packet is usually 1460 bytes [11]

Finally, we tweaked the TCP/IP stack. These settings are directly tied to the Linux kernel but can be changed at runtime. In particular, we increased memory allocation and the size of the SYN backlog, we reduced the number of SYN-ACK retries and the FIN timeout, and we switched the congestion control algorithm to BBR [2]. The SYN cookies were already enabled by default.

## Server

On the server, we configured the firewall using only the INPUT and OUTPUT chains of `iptables`, with a default DROP policy. Additionally, we allowed bidirectional TCP traffic on port 80 only for ESTABLISHED and RELATED connections.

We hardened the TCP/IP stack in the server the same way we did in the gateway.

Then we installed Varnish in front of Apache web server. Varnish is a web application accelerator, or HTTP reverse proxy, that caches the contents and serves them at a much higher rate [17]. As the web server was serving only static (empty) HTML pages, it was more efficient than *Apache*, easier to set up than *Nginx*, and it could handle slow HTTP attacks better.

We also configured Varnish as an application-level firewall, filtering out unwanted HTTP requests and allowing only GET requests.

## Monitoring

We spent a lot of time developing monitoring tools. We wanted something capable of providing useful insight into the actions taken by the *red team* and an easy way to check the server status constantly. After analyzing different scenarios, we depicted three tools, and we wrote them using Python and relying on the Pyshark [10] library.

- **traffic\_logger.py**: constantly checked the incoming traffic, displaying and updating every 500 ms a table with type, source IP, and size of each packet received. It was running on both the server and the gateway, helping us clearly understand what was happening on the network and understand what the red team was doing. Table 1 schematizes the script’s output.

SRC_IP	TCP	B_TCP	SYN	B_SYN	UDP	B_UDP	ICMP	B_ICMP	OTHER	B.OTHER
CLIENT_1	0	0	0	0	0	0	0	0	0	0
CLIENT_2	90	6497	1	74	0	0	0	0	0	0
CLIENT_3	0	0	0	0	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...
DELTA	9	0	1	0	0	0	0	0	0	0

Table 1: Traffic logger console output example

- **server\_logger.py**: checked the incoming HTTP packets to understand which page was requested by who. This tool allowed us to identify legitimate requests and discover incoming HTTP floods. It was running only on the server.
- **apache\_logger.py**: checked the Varnish log file, which we configured to include the response time of each HTTP request. It reported every HTTP request with a response time above a given threshold. It helped us understand when the server was overloaded and no longer able to serve requests within the time limit.

## 2.5 Execution

### 2.5.1 Attack

The CCTF mostly went according to the plan on the red team’s side. We stuck to our four-point attack strategy, and the events unfolded as follows:

- **Reconnaissance:** using different network scans, we discovered that the adversary team had replaced Apache with a NodeJS/Express web server. Moreover, we found that their firewall rules were a bottleneck for regular traffic, as there was a strict limit on concurrent requests per second per IP. Moreover, most bogus TCP flags and combinations were not allowed. During this phase, the rival team identified the legitimate client IP and blocklisted the others, forcing us to spoof any request as the legitimate client.
- **Focus:** we decided to start with SlowLoris and similar slow attacks, as the switch to NodeJS probably left them unpatched, then we switched to regular bulk SYN flooding.
- **Slow start:** We first used a client to send SlowLoris and Sockstress attacks to the server. It ended up being mostly a minor annoyance for the defense team. Then, we proceeded with carefully measured amounts of GoldenEye and hulk to see if a smaller amount of traffic would start to destabilize the server.
- **Wafer attack:** at approximately T+2200 seconds, we decided to pull off loacker first, then four instances of hping3 (as it is single-threaded). The opponent's server was quickly overloaded, and it started dropping packets and timing out entirely. We sustained the attacks until T+6000, often switching target between the gateway and the server.

Our strategy saw an abrupt end when the enemy team deployed a very suspicious countermeasure, leaving us crawling in the dust for half an hour. Realizing that curl uses incremental ports to open connections<sup>3</sup>, the adversary team blocked every type of traffic but the one coming from the legitimate client with last observed curl source port  $\pm 20$ . One of the defending team members was changing this sliding window manually.

In this period, the server and gateway machines were still recovering from our attacks, and there were some brief timeframes in which the server still did not output timely responses or timed out. Still, we kept blasting them with loacker and hping3, hoping to continue overloading their machines.

However, we failed to realize their sketchy approach in time for a complete change of strategy, and the CCTF ended shortly afterward.

## 2.5.2 Defense

The defense setup worked surprisingly well. During the CCTF, we managed to block most of the attacks. Still, some of them harmed us. In particular, GoldenEye partially slowed down our system. Luckily, it also caused a complete clogging of the adversary system, cutting them off from their machines.

The main two problems that we faced were auto-inflicted. First, our iptables rule for filtering based on MSS ended up doing more harm than good. That rule worked flawlessly against floods originated by hping3 and similar tools despite correctly allowing curl requests, with a typical MSS value of 1460. However, it also filtered out SYN packets of SSH connections, whose MSS value is 1330, thus cutting us out of the gateway for 30-40 minutes and effectively reducing our monitoring capacities. Fortunately, routing and filtering were not affected, and we were able to monitor the situation from the server. Ultimately, we rebooted the machine, inevitably losing some requests in that small timeframe.

Second, our traffic monitoring scripts were very CPU-intensive, causing some unexpected overloading issues at the gateway and some requests to slow down for a few seconds. The overload happened because we were adding another layer of packet processing (at some point two), written in Python – not the most efficient programming language – to an already stressed gateway.

In the last part of the CCTF, having assessed that our firewall was performing flawlessly, we stopped our monitoring tools and used tcpdump to have a general idea of the situation without causing unnecessary overloads.

---

<sup>3</sup><https://serverfault.com/a/875453>



## 3 CCTF Secure Server

### 3.1 Scenario

The network structure was identical to the Resilient Server CCTF (subsection 2.1). The only difference was that the router was performing Network Address Translation (NAT), and we noticed that only during the competition. For the blue team, this implied that every incoming packet had the source IP of the router (10.1.1.2).

The *server* had a LAMP stack configured with Apache, MySQL, and PHP. There were exposed APIs for a bank system accepting HTTP GET requests to create new users, fetch the balance, withdraw or deposit money. The *gateway* was left free to be customized by the blue team.

During the CCTF, some bank customers actively made requests and interacted with the bank APIs. Only the blue team knew the customers' names, not the red one. The blue team's goal was to operate the bank system correctly and keep the database consistent without being fined or robbed by the red team.

On the attack side, the three clients at the red team's disposal had only one limitation about traffic floods of one request per second. However, short bursts were allowed as long as the red team respected the limit over time (e.g., ten requests in a second then silent for nine seconds).

The red team had infinite money and could freely create new users and interact with them. Their ultimate goal was to rob the bank with compromising requests, asking for ransoms, or forcing the bank to pay fines [15].

### 3.2 Environment setup

Our previous setup, used in the Resilient CCTF and described in subsection 2.2, was again employed for this CCTF.

### 3.3 Attack preparation

#### 3.3.1 Goals

The red team's goal was to steal as much money as possible from the bank. Several approaches were possible to achieve this goal:

- Deleting or corrupting users' accounts via malicious requests to the APIs.
- Compromise the database and lead it into unexpected behavior (e.g., reducing a user balance below zero, creating money out of nowhere).
- Bring the server down to a halt.

Except for flooding techniques, it was possible to use any means to achieve such goals. For example, brute forcing any blue team member's password was allowed.

#### 3.3.2 Tools used

Since flooding was not allowed in this CCTF, we actively searched for exploits in the unpatched PHP and MySQL code provided to the blue team.

To help in our analysis, we used two main tools:

- **Nmap** [13]: a utility for network discovery and security auditing. It is used by white and black hat hackers alike in settings that range from reconnaissance to vulnerability discovery;

- **SQLMap** [3]: a penetration testing tool that automates the process of detecting and exploiting SQL injection flaws. When provided with an URL that supports query strings, **SQLMap** sequentially checks relevant injections and provides results to the user.

Slow DoS attacks were permitted, enabling us to use **Sockstress** [4] and **SlowLoris** [7].

To automate the requests to the web server, we developed **Attila**: a bash function with log functionalities that made possible to send requests to the server in a one-liner. This tool allowed us to create powerful scripts with minimal boilerplate code. An example of usage is the following which sends a request to the server and logs the response time:

```
attila "random_user" "user_s_password" "withdraw" "100"
```

A few hours before the CCTF, we tried a social engineering attack (described in subsection 3.3.3) using **Emkei's Fake Mailer** [5], a website that allows anyone to send forged emails with customized fields (e.g., sender, attachments, signature).

### 3.3.3 Strategy overview

Unlike the Resilient Server CCTF, we had almost no idea where to start, so we decided on a more generic and planned strategy, outlining it step-by-step. Then we divided the work between the team members, looking for different vulnerabilities.

1. At the beginning of the CCTF, use tools such as **Nmap** and **SQLMap** to test the capabilities of the adversarial system, looking for open ports and vulnerabilities.
2. After narrowing down the attack surface, choose the priority of the following attacks:
  - a. **Dictionary attacks**: brute force any of the opposing team's passwords, such as their deterlab SSH passwords or the legitimate bank users' passwords.
  - b. **Database flood**: flood the opposing system with a large number of user registration requests to enlarge the database tables enough to slow down the queries and the system.
  - c. **Discovery**: verify the presence of additional pages on the system, possibly forgotten, and try to compromise them.
  - d. **Corner case checking**: send crafted requests to the web server to crash it or trigger odd behavior. Test for SQL injection and XSS strings, then use **Attila** scripts to check edge cases such as empty requests, empty username requests, strange Unicode characters, and more.
3. Use the aforementioned **Emkei's Fake Mailer** to send a phishing email, pretending to be the Teaching Assistant. The email would contain instructions to set a specific root password for the database and create "default" bank users.<sup>4</sup> The email text is reported in Attachment B.

## 3.4 Defense preparation

### 3.4.1 Goals

The blue team's goals were to ensure 100% server availability over time and to preserve the database integrity. The server had to serve all the requests, and any timed-out request was marked as disputed and verified later. The database had to be as resilient as possible, with the underlying PHP code blocking any incorrect database query.

Moreover, the bank had to comply with different policies such as *KYC* requirements (unique customer identification), transactions logging, and data privacy. Violating any of these policies would result in a fine, ultimately decreasing the team's total score.

---

<sup>4</sup>We had the phishing email idea only a few hours before the CCTF. We report it here for completeness.

### 3.4.2 Strategy overview

We changed the server configuration to an LNMP stack with Nginx, MySQL, and PHP, thus removing the Apache web server. The provided PHP code was poorly written, with no input validation, plaintext passwords, and any other problem we could think of. The first part of the work was to fix the source code while preserving the APIs functionalities and the log structure.

Then, due to the packets rate restriction, our strategy focused on hardening the LNMP stack on the server rather than handling floods on the gateway like in the Resilient Server CCTF.

#### Gateway

We set up the gateway as the first line of defense, configuring the firewall with a default **DROP** policy and allowing everything on the DETERLab interface and only TCP traffic coming from the router to the server on port 80. The **iptables** configuration was similar to the one mentioned in paragraph 2.4.2 but without the **raw** and the **mangle** tables rules. We also limited at a maximum of ten parallel connections per IP.

#### Server

We decided to use Nginx to have a lighter web server that could handle loads and multiple connections better than Apache without requiring additional modifications. Also, using Nginx instead of Apache should have theoretically reduced the complexity of hardening our configuration, but in reality, we had a tough afternoon making the LNMP stack work flawlessly.

We hardened the database following the security guidelines provided by MySQL [16].

We refactored the PHP code entirely, adding a multi-layer input validation and filtering. We filtered all the input parameters using **htmlentities**, then we used the **filter\_var** function to check for odd strings or integers. Finally, we changed all the queries to prepared statements.

On top of this filtering, we added checks for negative amounts, integer overflow, user existence (to avoid duplicate users), and password hashing using the **ARGON2I** algorithm [1].

#### Database

We kept MySQL, disabling local file access and introducing an incremental delay after each failed login attempt to the database. Remote access was already disabled.

We rebuilt the database as shown in Figure 2. A user was identified through a unique username and could have zero or more transfers. Randomly-generated 32-bit unique identifiers distinguished transfers, and each transfer could be associated with one and only one user.



Figure 2: Database ER diagram.

We set up three different users: a *root user* with complete access to the database, an *editor user* with **INSERT**, **SELECT** and **UPDATE** privileges on our two tables, and a *monitor user* to check the database consistency and perform backups.

#### Monitoring

Again, as in the Resilient Server CCTF, we developed monitoring tools in Python. Since DoS attacks were not allowed, we focused on the integrity of the database.

- **database\_checker.py**: a script that checked the consistency of the database every 60 seconds. It connected to the database with the *monitor user* and performed four checks:
  - *Negative amount*: the sum of the transfers values of any user could have not been negative. If a user had a negative balance, it displayed the correspondent username on the screen.
  - *Number of transfers and users*: the number of users and transfers should have never decreased since we did not allow the delete command.
  - *Matching balance*: each user’s balance should have matched the sum of their transfers. In case of incongruences, the corresponding username was displayed on the screen.

The script also monitored the MySQL log file, displaying an alert in case of any failed attempt to connect to the database. Moreover, we added a backup function to save the entire database and the **request.log** file if the former was intact. During the CCTF, we kept the three most recent backups.

This tool allowed us to control the database consistency and offered a quick way to restore everything in case of a successful attack.

- **traffic\_logger.py**: we reused the **traffic\_logger** developed for the Resilient Server CCTF without modifications. It helped us understand that the router was doing NAT.

## 3.5 Execution

### 3.5.1 Attack

In this CCTF, the attack part was a failure. Our strategy mostly followed the plan, but unfortunately, we could not compromise anything on the blue team side, and thus we did not steal any money.

As a last-minute idea, we tried our luck with a phishing email (Attachment B). To make it appear as realistic as possible, we added the Teaching Assistant’s signature, removed any attachments triggering Gmail’s spam filters, and sent one copy to the other two team leaders. Our plan failed because Team 1 asked the Teaching Assistant for clarification and thus spoiled the phishing attempt to Team 2, whose leader did not even check his incoming email.

In the initial part of the CCTF, we deployed **Nmap** and **SQLMap** scans as planned. We additionally used **Nmap**’s vulnerability scanners, hoping to find some exploitable vulnerabilities. Unfortunately, we had no luck. Not only were all SQL injection and XSS attempts blocked, but the scan yielded no vulnerability exploitable in a reasonable amount of time.

We split the job and started working on the side channels with no other options left. Using **Attila**, we flooded the database with a list of a million users to inject them into the opponent’s database. However, the slow one-request-per-second rate meant that only a small fraction of the users made their way into the system. At a certain point, we spotted an already registered user, thus triggering the second step of our plan.

We equipped our **BruteForcer** script with a list of common passwords and ran it against the web server. In the end, maybe also because of the limited rate, we could not guess the user’s password.

Meanwhile, another member of the team tried different side-channel tactics. After failing to discover secondary pages on the web server, he checked for corner cases such as integer parsing, empty usernames or passwords, and other odd combinations. He tried using Unicode sequences and international characters as the last stand. Unfortunately, nothing of the above worked.

After the end of the competition, the professor revealed that our blue team missed a severe vulnerability and that our red team almost managed to exploit it unknowingly during the database flooding. The size of the log file had a quadratic growth. So, by increasing the dimension of the tables in the database, the size of the log file would increase, eventually causing the server to halt during the logging procedure. Unfortunately, the strict request rate limit and the fact that we started late to flood the database resulted in us not exploiting this vulnerability.

### 3.5.2 Defense

Our defenses defied the opposing red team attacks. The monitoring tools reported no compromise, and the Teaching Assistant later confirmed this. Unfortunately, we misconfigured something initially, ending up missing some requests. Some were flagged as disputed and later resolved, but others went lost.

First, we forgot to allow DNS traffic on the gateway when configuring the firewall. Without DNS traffic, the gateway was cut out entirely from the DETERLab network for a few minutes before we could realize what was happening. All the incoming requests in that timeframe went inevitably lost. Second, the concurrent connections limit did more harm than good. We set up the firewall to allow a maximum of ten parallel connections from the same IP. However, we did not notice that the router was performing NAT, so when the red team started attacking us with **Sockstress**, thus opening several parallel connections, our firewall blocked almost all the legitimate ones due to the abovementioned rule.

We took a while to realize what was happening because everything was running fine from our standpoint. The only weird thing is that we were receiving only traffic from the router, but we thought the Teaching Assistant was sending it, so we were not alarmed.

## 4 Team roles

**Lorenzo Cavada:** blue team member, author of the monitoring and logging scripts used in both CCTFs. He additionally wrote a brute-forcing script for the Secure Server CTF.

**Matteo Franzil:** read team member, he researched the attack tools, testing everything and ensuring they were working correctly and effectively. He focused on writing automated tools, such as `legitimate` and `Attila`. He also wrote most of the setup scripts for both teams' automated setup.

**Dmytro Kashchuk:** red team member, author of the helper scripts to wrap all the attack tools and make them easier to execute during the CCTFs. He had the idea of the phishing email sent before the Secure Server CTF and was also the one to send it.

**Tommaso Sacchetti:** blue team member, he managed the firewall and web server configurations in both CCTFs. He had the idea to configure Varnish in the Resilient Server and switch to NGINX in the Secure Server. He also wrote `loacker` for the red team.

## References

- [1] A. Biryukov, D. Dinu, and D. Khovratovich, “Argon2: New generation of memory-hard functions for password hashing and other applications,” in *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, 2015, pp. 292–302.
- [2] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, “Bbr: Congestion-based congestion control,” *ACM Queue*, vol. 14, September-October, pp. 20 – 53, 2016. [Online]. Available: <http://queue.acm.org/detail.cfm?id=3022184>
- [3] B. Damele and M. Stampar. SQLMap. SQLMap. [Online]. Available: <https://sqlmap.org/> Accessed 2021-12-07.
- [4] Defuse. Sockstress. GitHub. [Online]. Available: <https://github.com/defuse/sockstress> Commit d609ad41e71b7ffe504947e69599a65a64fae529. Accessed 2021-12-06.
- [5] Emkei. Emkei’s Fake Mailer. [Online]. Available: <https://emkei.cz/> Accessed 2021-12-07.
- [6] FFFaraz. Etcetera. GitHub. [Online]. Available: <https://github.com/fffaraz/Etcetera> Commit b0acbf49ca950807b7ee4bc79d14d53785a88b9b. Accessed 2021-12-06.
- [7] Gkbrk. SlowLoris. GitHub. [Online]. Available: <https://github.com/gkbrk/slowloris> Commit 5e2bb196cf362d95478dda9e4a6a75fd3f4af0fc. Accessed 2021-12-06.
- [8] Grafov. Hulk. GitHub. [Online]. Available: <https://github.com/grafov/hulk> Commit ed2b11c1530e03fb872c008fa669a56d767be474. Accessed 2021-12-06.
- [9] Jseidl. Goldeneye. GitHub. [Online]. Available: <https://github.com/jseidl/GoldenEye> Commit 792862f5c8cb98f9ffcb9fab245e2c663e3a1026. Accessed 2021-12-06.
- [10] KimiNewt. PyShark. GitHub. [Online]. Available: <https://github.com/KimiNewt/pyshark> Accessed 2022-01-03.
- [11] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach*, 7th ed. Boston, MA: Pearson, 2016, pp. 258–260.
- [12] Linux DIE manpage. HPing3. [Online]. Available: <https://linux.die.net/man/8/hping3> Accessed 2021-12-06.
- [13] G. Lyon. Nmap. Nmap. [Online]. Available: <https://nmap.org/> Accessed 2021-12-07.
- [14] J. Mirkovic. CCTF: Resilient Server. DETERLab. [Online]. Available: <https://www.isi.deterlab.net/file.php?file=/share/shared/ResilientServerCCTFPre-Set> Accessed 2021-12-07. Used as a blueprint; the CCTF had slightly different rules.
- [15] ——. CCTF: Secure Server. DETERLab. [Online]. Available: <https://steel.isi.edu/Projects/Intel/CTF/CTF2/> Accessed 2021-12-07. Used as a blueprint; the CCTF had slightly different rules.
- [16] Oracle. Security guidelines. Oracle. [Online]. Available: <https://dev.mysql.com/doc/refman/8.0/en/security-guidelines.html> Accessed 2022-01-04.
- [17] Varnish Software. Varnish. Varnish Software. [Online]. Available: <https://varnish-cache.org/> Accessed 2022-01-12.

# Attachments

## A List of experiment networks

Subnet	Point 1	Point 2
10.1.1.0/29	router (10.1.1.2)	gateway (10.1.1.3)
10.1.2.0/29	router (10.1.2.3)	client1 (10.1.2.2)
10.1.3.0/29	router (10.1.3.3)	client2 (10.1.3.2)
10.1.4.0/29	router (10.1.4.3)	client3 (10.1.4.2)
10.1.5.0/29	server (10.1.5.2)	gateway (10.1.5.3)

Table 2: All point-to-point experiment networks.

## B Example of phishing email

The following is a sample email that was sent before the start of the Secure CCTF by the red team. Names and personal details have been edited out for privacy.

Dear students, I kindly ask you to set your DB password as below:

user: root

password: 6Wt8\$Y9Cms\*Dz22u

also I ask you to create the following users in the database:

giorgio;V2XdZ5^\*Wub6eZZb

fabio;H\$736bzE6J%eGzd%

babbonatale;jmQ2^EtQ4!SkmUY6

--

Teaching Assistant - Ph.D. student

DISI - Department of Information Engineering and Computer Science

University of Trento, IT

Povo 2, Office ---

Web site: ---