CS-745 Project Report

Build a Transparent SSL-Proxy Server

Team Members:

Akanksha Dadhich (23m0830)

Nikita Verma (23m0807)

Rashmi Kokare (23m0785)

Department of Computer Science and Engineering

INDIAN INSTITUTE OF TECHNOLOGY, BOMBAY

May, 2024

Acknowledgement

We would like to thank our guide, **Dr. Virender Singh**, for his constant support and guidance throughout this course. We extend our thanks to the other TAs of the Principles of Data and System Security course and the entire Computer Science Engineering department for providing interesting topics and enriched learning experience.

Abstract

This project delves into the critical role of Secure Sockets Layer (SSL) or Transport Layer Security (TLS) in safeguarding online communication. It explores how this application layer protocol establishes a secure tunnel between communicating parties, shielding data from unauthorized access by third parties. The report analyzes the use of symmetric encryption within this tunnel, negotiated using the SSL/TLS protocol itself.
Furthermore, the project examines the importance of digital certificates in verifying the authenticity of communicating entities. It discusses the typical involvement of a Public Key Infrastructure (PKI) for certificate distribution, with the option for direct exchange if parties possess pre-shared certificates.

Index

1 Setting Up of Servers

1. Begin by installing VirtualBox, a software that allows us to run multiple operating systems on a single machine.

2. Create a Linux virtual machine with a minimal set of packages. This VM, named *vm-nox-vanilla*, will lack a graphical interface but will include an OpenSSH server for remote access.

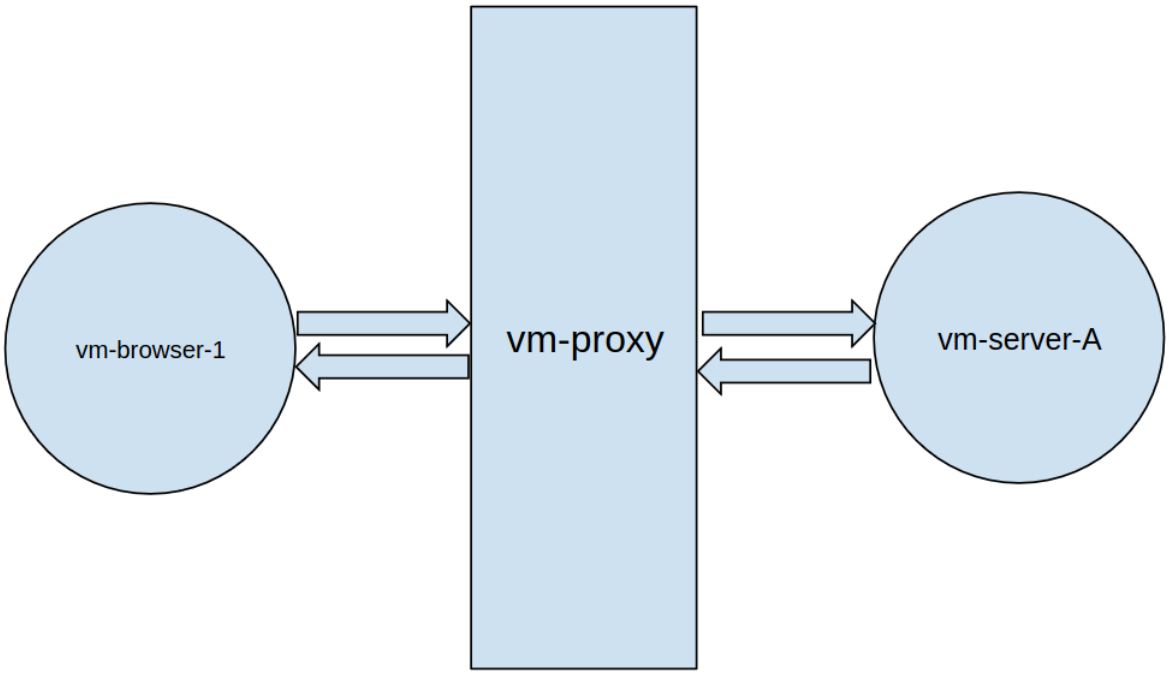
3. Next, create another Linux virtual machine, *vm-gui-vanilla*, equipped with a graphical desktop environment. This VM will only require a web browser, while other programs can be skipped during installation. However, an OpenSSH server is still recommended for remote access.

4. Duplicate the *vm-nox-vanilla* machine to create two new VMs: *vm-server-A* and *vm-proxy.* These will serve as our web server and network proxy, respectively.

5. Replicate the *vm-gui-vanilla* machine to create *vm-browser-1*. This VM will house the Firefox web browser used for our experiments.

6. With these VMs in place, you'll have a network with three active machines:
    a. A virtual machine dedicated to running the Firefox browser (*vm-browser-1*).
    b. A virtual machine acting as a proxy to route the browser's traffic to the application hosted on another VM (*vm-proxy*).
    c. A virtual machine serving as the web server hosting the target application (*vm-server-A*).

7. As illustrated in Figure, all communication between the browser in *vm-browser-1* and the web server in *vm-server-A* will be channeled through the proxy (*vm-proxy)*. Depending on your VirtualBox setup, you might need to adjust the network manager settings to ensure proper traffic routing.


vm-browser-1
vm-proxy
vm-server-A

2 Setting Up a Vulnerable Bulletin Board for XSS Attack Simulation

This chapter explores Cross-Site Scripting (XSS) vulnerabilities and their exploitation through a simulated attack. We will set up a vulnerable bulletin board application and demonstrate how an attacker can inject malicious scripts into the application to compromise the victim *proxy-server*.

Steps to follow the task:

1. We will install a web server (Apache2) and a database (postgresql) on a virtual machine (*vm-server-A*).

2. We will then deploy a bulletin board application, either using phpBB or a custom-developed script. The key is to ensure the application has exploitable vulnerabilities that allow XSS attacks.

3. Network traffic collection will be initiated on the proxy machine (vm-proxy) to capture all communication between the browser and the web server.
4. We are using Reflected XSS: The attacker injects malicious code into a form or search bar on the bulletin board. When the application processes the user input without proper validation, the malicious script gets reflected back to the victim's browser and executes.

5. Once a successful XSS attack is demonstrated, we implement mitigation techniques on the web server.

6. One approach to mitigate XSS vulnerabilities involves sanitizing all user input before processing it on the server-side. This can be achieved by converting the input to a plain text string, effectively removing any potential code embedded within.

7. Here's how this technique works:
    - ❖ When a user submits data through a form, search bar, or other input mechanism on the bulletin board application, the application captures this input.
    - ❖ Before using the user input in any database queries, logic processing, or output generation, the application explicitly converts the input to a string data type. This conversion process strips away any embedded HTML tags, scripts, or other code that might be malicious.
    - ❖ Once converted to a plain text string, the user input is considered safe and can be processed by the application without the risk of XSS attacks.

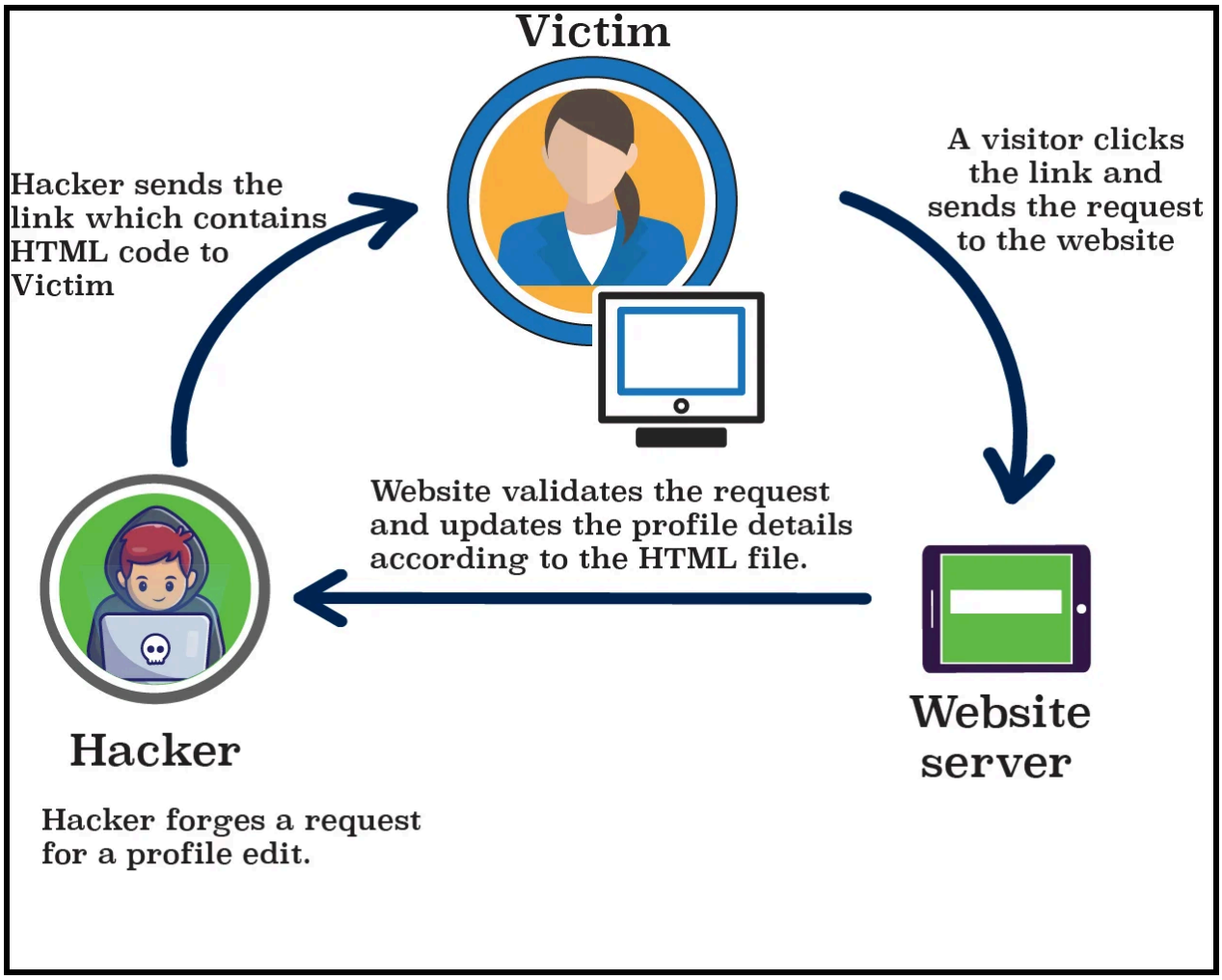
8. Throughout the experiment, network traffic will be captured in "***traffic-task-1.pcap***".

<u>Some other experiments done for this task:</u>

3 Simulating a CSRF Attack and Mitigating XSS on an Extended Network

This task builds upon the initial setup and demonstrates a Cross-Site Request Forgery (CSRF) attack. In Cross-Site Request Forgery (CSRF) attacks, the attacker sets up a malicious website or web page under their control. This platform will host the code designed to exploit the victim's browser. The attacker then entices the victim to visit the attacker-controlled platform. This could involve sending a seemingly legitimate link through email, social media messages, or other deceptive tactics.

By clicking the link and visiting the attacker's platform, the victim's browser unknowingly executes the malicious code, potentially performing unauthorized actions on the victim's accounts on other websites.



Following steps for this task:

1. Deploy a New Web Server *vm-server-B* (clone it from *vm-nox-vanilla* machine).

2. Start the newly created *vm-server-B* virtual machine.

3. Configure *vm-server-B* to host a web application vulnerable to CSRF attacks. This could be a custom script or a pre-existing application with a known CSRF

vulnerability. This can be done by constructing a web page that has following HTML:

```
<img
src="https://t4.ftcdn.net/jpg/00/97/58/97/360_F_97589769_t45CqXyzjz0KXwoBZT9P
RaWGHRk5hQqQ.jpg" style="display: block; margin-left: auto; margin-right: auto;
width: 70%;">
                    <form hidden target="transFrame"
action="YOUR_WEBSITE_URL_VULNERABLE_TO_CSRF_ATTACK"
method="POST">
        <input id="username" name="username" value="attacker" >
        <input id="email" name="email" value="attacker">
  <button id="submit" type="submit"></button>

 </form>
 <body onload="document.forms[0].submit.click()">

 <iframe hidden style="" name="transFrame" id="transFrame"></iframe>
```

Malicious website view

Actual website which is vulnerable to CSRF attack

- This HTML page(attacker’s) contains a form that will automatically submit a POST request to change the name and email on the target website when loaded.

- The `action` attribute of the form specifies the URL of the target website's name and email change endpoint.

- The form includes a hidden input field named name and email with the value set to the attacker's desired values.

- JavaScript is used to automatically submit the form when the page loads, simulating the victim's browser executing the malicious request without their knowledge.

4. Hosting this crafted HTML page on a web server(*vm-server-B*) that the attacker has control on.

5. Trick the victim(*vm-browser-1*) into visiting the malicious HTML page.

6. When the victim(*vm-browser-1*) visits the malicious HTML page, their browser will automatically submit the form, triggering the CSRF attack. This will result in the victim's name and email being changed on the target website(actual server *vm-server-A*) without their consent.
7. Capture the traffic in "***traffic-task-2.pcap***".

<u>XSS Mitigation techniques:</u>

1.CSRF Tokens: Introduce a CSRF token (also known as a synchronizer token) as an additional security layer. This token is a unique, unpredictable value associated with a user's session.

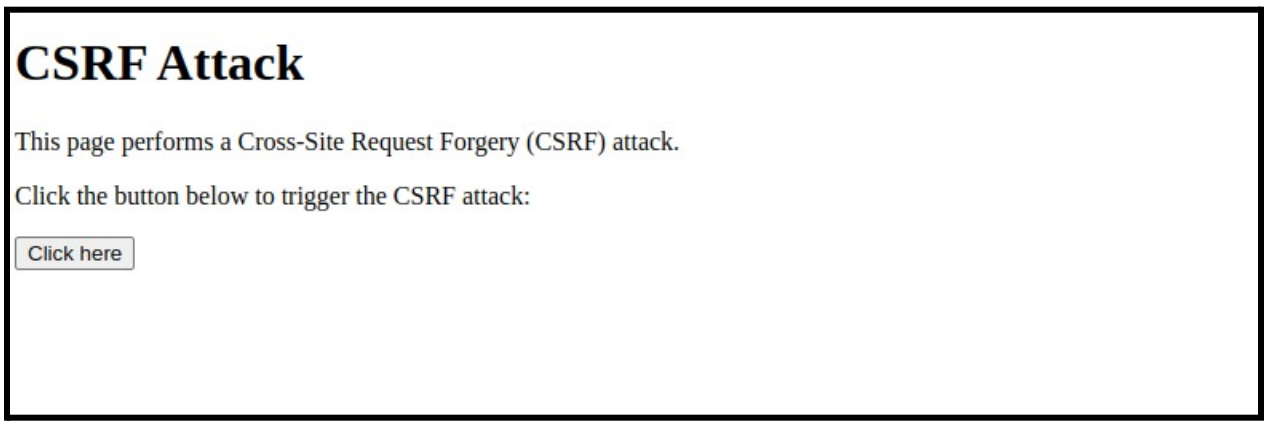
Embed in Forms: Embed the CSRF token as a hidden field in all HTML forms within your application.

Validate Token: Upon form submission, validate the submitted CSRF token on the server-side. Ensure it matches the token associated with the user's current session. If the tokens don't match, reject the request as a potential CSRF attempt.

2. Input Validation and Sanitization: Implement robust input validation and sanitization techniques on the server-side.

3. Content Security Policy (CSP): Implement a Content Security Policy (CSP) on your server. This policy defines which resources (scripts, stylesheets, images) can be loaded by your web pages, restricting the execution of unauthorized code.

Some other experiments done for this task:



Values stored in database like this:

```
Password:
==== AUTHENTICATION COMPLETE ====
server@nikitaserver:/var/www/html/phpbb$ sudo mysql
[sudo] password for server:
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 127
Server version: 10.11.7-MariaDB-2ubuntu2 Ubuntu 24.04

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> use phpbb
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
MariaDB [phpbb]> select * from user_inputs;
+--------------+----------------------+
| name         | email                |
+--------------+----------------------+
| akanksha     | akanks@gmail.com     |
| akanksha     | akanks@gmail.com     |
| Chaitra      | chaitra@gmail.com    |
| Chaitra      | chaitra@gmail.com    |
| Chaitra123   | chaitra@gmail.com    |
| nikita       | nikita@1234          |
| hello        | 112435               |
| hello        | 112435               |
| hello        | 112435               |
| hello        | 112435jhk            |
| hello tttt   | 112435jhk            |
| hefcfllo tttt | 112435jhk           |
| hefcfllo tttt | 112435jhk           |
| dadhich      | dadhich3001@gmail    |
| dadhich      | dadhich3001@gmail    |
| dadhich12344 | dadhich3001@gmail    |
| love         | dadhich3001@gmail    |
| love         | dadhich3001@gmail    |
| love         | dadhich3001@gmail    |
| love         | dadhich3001@gmail    |
| Akanksha     | nikita@gmail.com     |
| Nikita       | rita@google.com      |
| Nikita       | rita@google.com      |
| Nikita       | rita@google.com      |
| Nikita       | rita@google.com      |
| Nikita       | rita@google.com      |
```

4 Implementing Single Sign-On for Secure Multi-Application Access

We are using Google Sign-In API for this task to install Google SSO using the python-social-auth library on a web server VM running a Python/Django web application.

It includes following steps:

1. Prerequisites for this are:
   - ❖ A running web server VM with Python and pip installed.
   - ❖ A Django web application on the server VM.
   - ❖ A Google Cloud Platform (GCP) project with the Google Sign-In API enabled.

2. Install required packages using:
   sudo pip install python-social-auth
   pip install requests social-core

3. Update Django settings.py to add **YOUR_CLIENT_ID** and **YOUR_CLIENT_SECRET**

4. Restart the development server using python manage.py runserver.

5. Access your Django application's login page on the browser VM.You should see a button or option to sign in with Google. Click on the button and follow the Google login flow.

6. Capture the network traffic during the login process. Save the captured traffic as "***traffic-task-3.pcap***"

5 Securing Communication with Self-Signed Certificates and Encrypted Traffic

This task focused on securing communication between web browsers and web servers by enabling HTTPS with self-signed certificates.

1. We created self-signed certificates on the web servers (*vm-server-A* and *vm-server-B*) using OpenSSL.

2. These certificates act as digital credentials for the servers, but unlike trusted certificates issued by a Certificate Authority (CA), they are self-issued.

3. We have created self signed certificates using openssl command with a validity period of 365 days (adjustable with -days). It uses a 2048-bit RSA key for encryption (considered secure).

4. After that, modifying the apache configuration file for SSL (updating the certificate and key file paths). This has been done for both the servers (*vm-server-A* and *vm-server-B*).

5. Followed by enabling the SSL module and SSL site:

sudo a2enmod ssl

sudo a2ensite default-ssl.conf

6. Restart apache2 to apply the changes.

7. Use *tcpdump* command to capture the encrypted traffic on the proxy machine's interface connected to the web servers. This captured traffic was saved to a file named "***traffic-task-4.pcap****".*

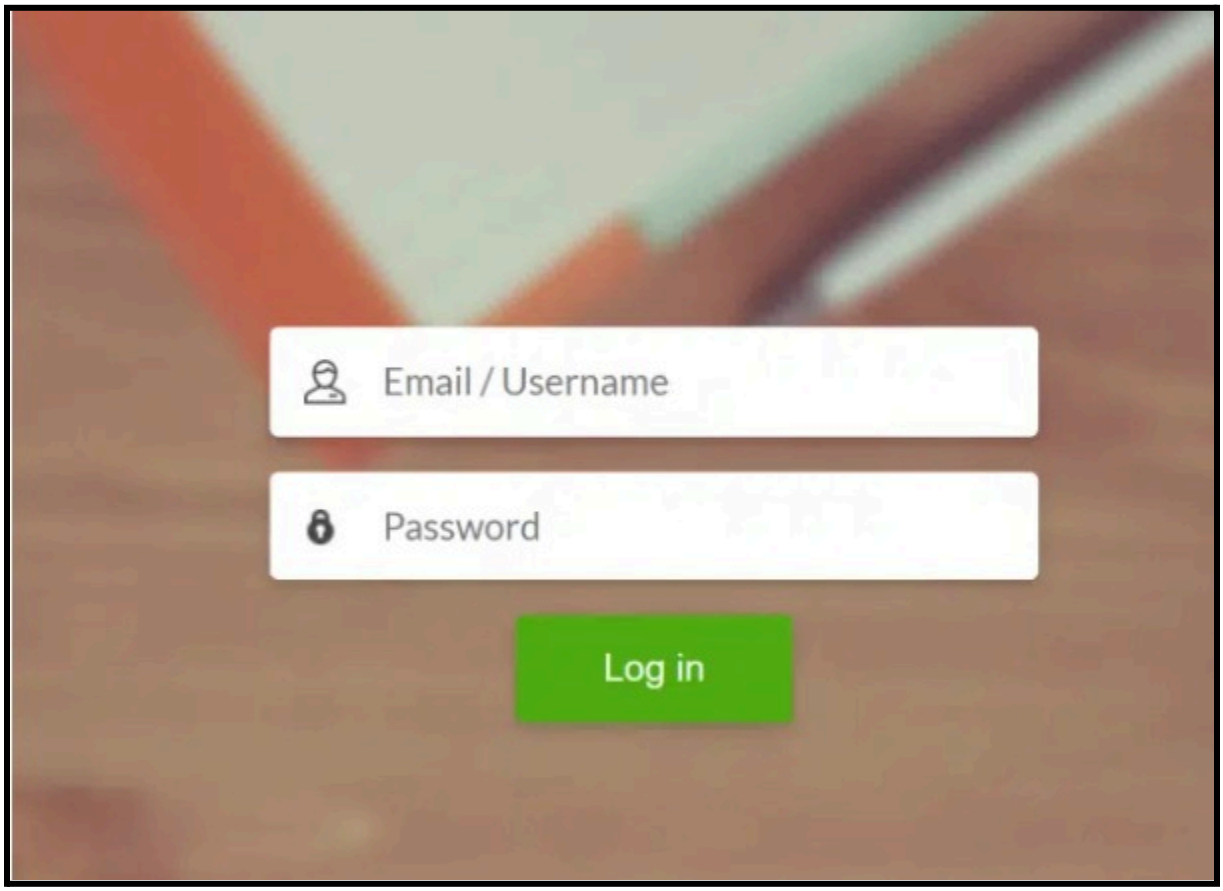
6 Transitioning to Client-Side Certificates for Secure User Authentication

Here we need to disable Single Sign-On (SSO) and enable user authentication using client-side digital certificates for our web applications.

Disabling SSO:

For this we have to modify Django *settings.py.* Search for the section where social authentication providers (including Google) are configured. This may involve looking for lines related to *django-allauth*, *django-social-auth,* or *python-social-auth*, depending on the authentication library used. Locate the configuration for Google authentication. It typically includes settings like,
SOCIAL_AUTH_GOOGLE_OAUTH2_KEY and ***SOCIAL_AUTH_GOOGLE_OAUTH2_SECRET***
Comment out or remove the configuration related to Google authentication.
After that remove or comment out the URLs related to Google authentication in *urls.py*.

After making the changes, restart your Django server to apply the modifications:
sudo systemctl restart gunicorn



Enable user authentication from browser:
We can create self-signed certificates on each client machine (*vm-browser-1* and *vm-browser-2*) using a tool like OpenSSL.

Configuring Django for Client-Side Certificate Authentication:

Install *django-client-certificate.*
Add Middleware to *settings.py.*

Add *'client_certificate.middleware.ClientCertificateMiddleware'* to the *MIDDLEWARE* list.
Configure setting related to *MIDDLEWARE CLIENT_CERT_REQUIRED: True,* etc..
Restart Development Server or Web Server using *python manage.py runserver.*

Now open a terminal window on your client machines (*vm-browser-1* and *vm-browser-2*). Navigate to the directory where you generated the certificates (in out directory).
Use the following command to export the user certificate in PKCS#12 format:
openssl pkcs12 -export -out user-vm_browser_1.pfx -inkey out/user-vm_browser_1.key -in out/user-vm_browser_1.crt

This will create a new file named *user-vm_browser_1.pfx* in the same directory. This file contains both the user's private key and certificate in a format compatible with most browsers.

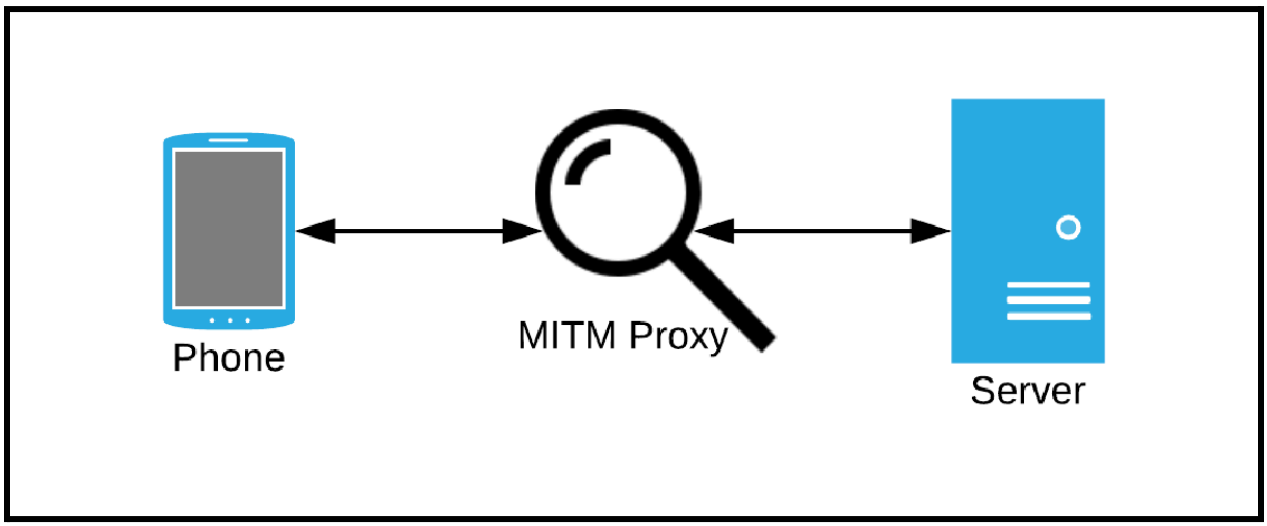
Importing the Certificate into Your Browser:

*1. Open Chrome and navigate to **Settings** -> **Privacy and security** -> **Security**.*
*2. Click on **Manage certificates**.*
*3. In the "Import" tab, click **Import**.*
*4. Select the `user-vm_browser_1.pfx` file you exported earlier and click **Open**.*
5. Enter a password (choose a strong password to protect the certificate) when prompted. This password will be required each time you use the certificate for authentication.
*6. Click **Import**.*

After that access the application's login page. Select the appropriate certificate associated with the user trying to access the application. If the certificate is valid and matches the configured criteria (e.g., subject field), the user should be granted access.
Network traffic capturing continues on the proxy machine (*vm-proxy)* throughout the certificate-based login processes and is saved as "*traffic-task-5.pcap*".

7 Implementing a Transparent SSL Proxy for Encrypted Traffic Inspection

Setting up a transparent SSL proxy on *vm-proxy* using *mitmproxy* to intercept and analyze HTTPS traffic between *vm-browser* and *vm-server-A.*



This task is done using following steps:

1. Install *mitmproxy* using:

   sudo apt install mitmproxy

2. Start *mitmproxy* in Transparent Mode:
   mitmproxy --host -T -p 8080

3. In *vm-browser* change network settings as:
   Access your Network settings and navigate to edit your Network Proxy configuration. Choose the Manual option, then specify localhost and port 8080 as the settings for both your HTTP and HTTPS proxies.

4. Open a web browser on *vm-browser* and attempt to access a website hosted by *vm-server-A.*
5. *Save the captured traffic as "**traffic-task-6.pcap**".*

8 References

This chapter provides a list of resources consulted during the completion of this project. In addition to the resources listed below, several other websites and online tutorials were consulted during the project making.

1	https://blog.stapps.io/mitmproxy-on-ubuntu-19-04/
2	https://portswigger.net/web-security/csrf
3	https://youtu.be/8qGL58yGWD0?si=pSsP_NcX7uDDxdX5
4	https://youtu.be/TKyHxceaEVg?si=8hcIrwkpVZhjT4YN
5	https://youtu.be/_TjRgW6ViYo?si=TsGi8qn6Ae-IqrJS
6	https://www.w3schools.com/django/django_create_project.php
7	https://phoenixnap.com/kb/install-virtualbox-on-ubuntu