CS 745 Project

Aratrik Chandra (23M0786)
Ankush Mitra (23M0755)
Pranab Paul (23M0800)

May 2024

Contents

Chapter 1

Task 1

Web applications are susceptible to a particular kind of security flaw called Cross-Site Scripting (XSS). It happens when a web programme incorporates user input without encoding or verifying it into the output it produces.XSS attacks are when an attacker sends malicious code, usually in the form of a browser-side script, to a different end user, injected into otherwise benign and trusted websites. The end user's browser executes the script, allowing the malicious script to access sensitive information like cookies and session tokens. XSS attacks can be categorized into reflected and stored types, with a third less well-known type called DOM-Based XSS. Reflected XSS attacks involve the injected script being reflected off the web server, while stored XSS attacks occur when the malicious script is stored on the server and sent to users without proper sanitization. To prevent XSS attacks, it is crucial to validate user input and encode or escape any output generated by a web application. The variety of XSS attacks is almost limitless, but they often involve transmitting private data, redirecting the victim to web content controlled by the attacker, or performing other malicious operations on the user's machine under the guise of a vulnerable site.

1.1 Proxy Setup

We use Nginx ,a popular web server software.A server listens on port 80 and proxies requests to another server at http://192.168.0.109:5000. The proxy_set_header directives ensure that the proxied server receives important information about the original request. This is the proxy configuration

```
include /etc/nginx/conf.d/*.conf;
include /etc/nginx/sites-enabled/*;

server {
  listen 80;
  server_name 192.168.0.113;

  location / {
    proxy_pass http://192.168.0.109:5000;
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Proto $scheme;
  }
}
```

1.2 Server Code

The server code in the app.py file:

```
from flask import Flask, render_template, request, jsonify
import json
import html  # Import html module for escaping HTML characters

app = Flask(__name__)

# File to store user messages
USER_DATA_FILE = 'user_data.json'
```

```

# Load existing user data from JSON file or initialize if file not found
try:
    with open(USER_DATA_FILE, 'r') as f:
        user_data = json.load(f)
except FileNotFoundError:
    user_data = []

@app.route('/')
def home():
    return render_template('index.html', messages=user_data)

@app.route('/submit', methods=['POST'])
def submit():
    if request.method == 'POST':
        message = request.form['message']

        # Store the message in user_data
        user_data.append(message)

        # Save updated user data to JSON file
        with open(USER_DATA_FILE, 'w') as f:
            json.dump(user_data, f, indent=4)

        # Return a response that includes the message for potential XSS execution
        return f"<script>alert('{html.escape(message)}');</script>"

    return home()

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

This is the html file:

```
    <!DOCTYPE html>
<html>
<head>
    <title>Flask XSS Demo</title>
</head>
<body>
    <h1>User Messages</h1>
    <ul>
        {% for message in messages %}
        <li>{{ message | safe}}</li>
        {% endfor %}
    </ul>
    <form method="POST" action="/submit">
        <input type="text" name="message" placeholder="Enter your message">
        <input type="submit" value="Submit">
    </form>

    <!-- Execute scripts embedded in user messages -->
    <script>
        var messages = {{ messages | tojson }};
        messages.forEach(function(message) {
            try {
                var script = document.createElement('script');
                script.textContent = message;
                document.body.appendChild(script);
            } catch (e) {
                console.error('Error executing script:', e);
            }
        });
    </script>
</body>
</html>
```

1.3 XSS Attack

In the app.py file, the line message = request.form['message'] retrieves the message from the form data in the POST request. This is user input and is a potential vector for XSS attacks.

1.4 XSS Mitigation

Before the input is used anywhere, it can be escaped using the html.escape() function. This function replaces any HTML characters in the user input with their corresponding HTML entities. For example,

```
"<" becomes \&lt; and ">" becomes \&gt; and so on.
```

1.5 Observation

In the file traffic-task-1.pcap, attacker submits the script :

```
<script> alert("XSS Attack") </script>
```

1.5.1 Before XSS Mitigation

In packet 47, first the attacker submits a script in the message box given in the server shown in figure 1.1. In packet 90, When a victim browser visits the server after the attack and his/her browser

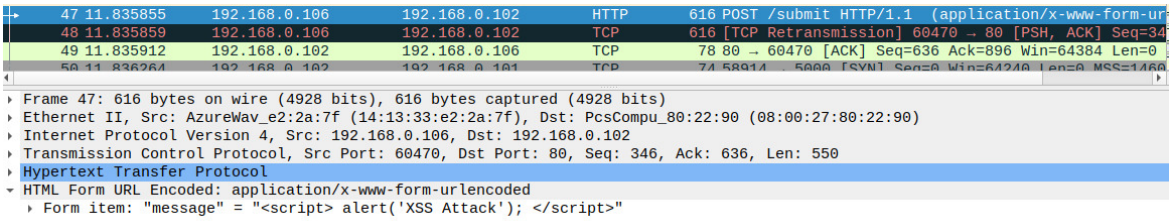

Figure 1.1: Attacker submits malicious input before mitigatino

executes the html of the server, victim is attacked successfully (shown a pop up by the server). We can see in figure 1.2, the html content which the victim browser gets from the server has JavaScript input in it which is executed in client browser and shows the pop up.

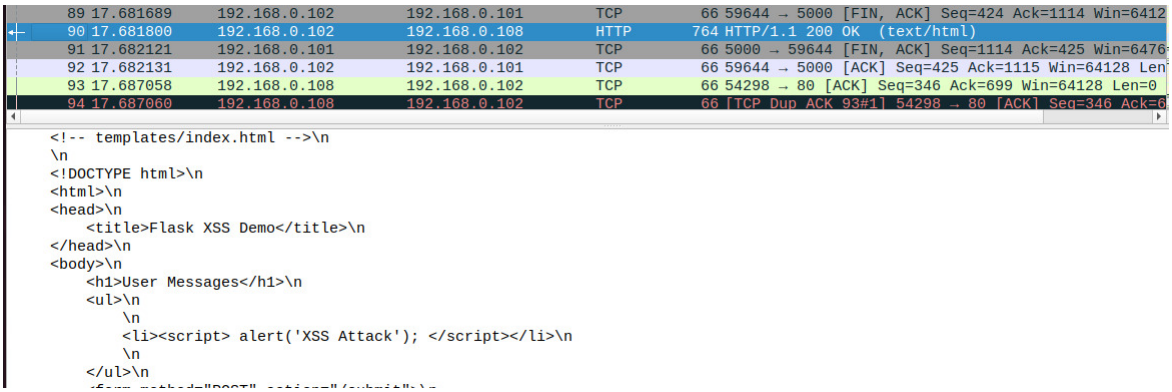

Figure 1.2: Victim browser is attacked

1.5.2 After XSS Mitigation

In packet 203, the attacker again submits a script in the message box given in the server shown in figure 1.3.

In packet 243, When a victim browser visits the server after the attacker submits a script.Then his/her browser executes the html of the server, but now there is no JavaScript input present in the html code because the input is sanitized now by the server. We can see in figure 1.4, the html content which the victim browser gets from the server has sanitized input in it which doesn't show any pop up.

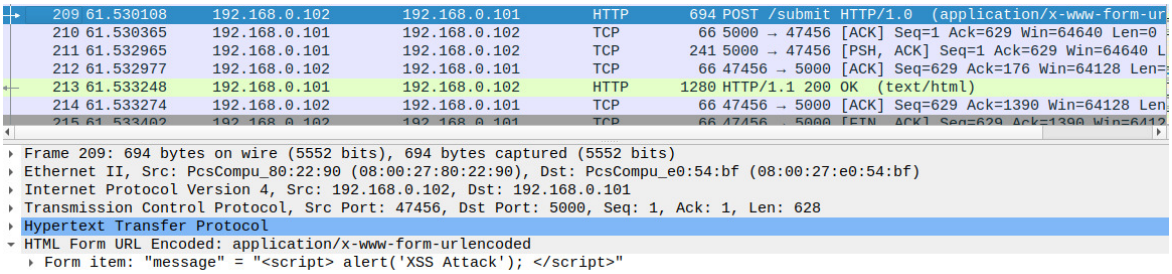

Figure 1.3: Attacker submits malicious input after mitigation

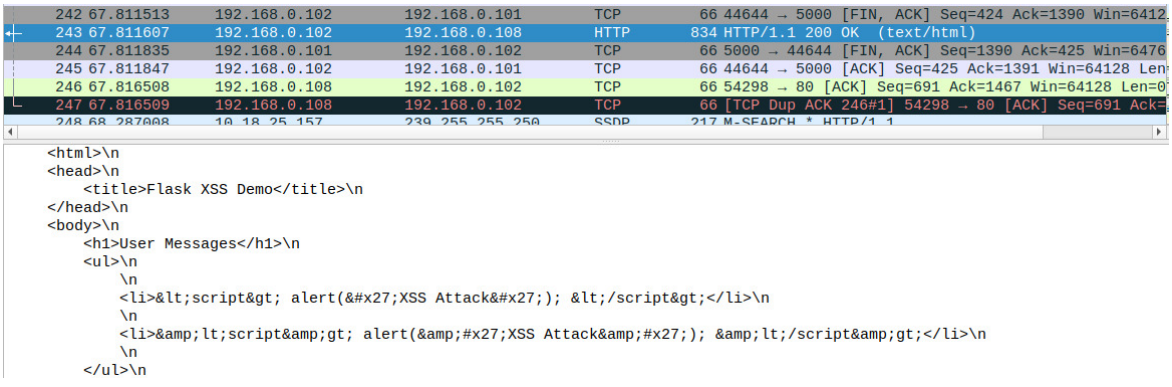

Figure 1.4: Victim browser is not attacked

Chapter 2

Task 2

Cross-Site Request Forgery (CSRF) is an attack that tricks an authenticated user into executing unwanted actions on a web application. It exploits the trust a web application has in its users. The impact can range from minor changes to the user's account to a complete compromise of the web application. Protection against CSRF involves using unique tokens in each request. CSRF attacks target state-changing requests, not data retrieval.

2.1 Proxy Setup

Same as previous task

2.2 Server setup

To experiment with the CSRF lab, first, ensure you have the necessary files set up: 'login.html', 'user.html', 'app.py', and 'attack.html' (or 'attack.py'). Start by running the Flask application using 'app.py', which will serve the web application.Here is the code of app.py:

```
from flask import Flask, request, render_template, redirect, url_for, session,
    make_response
from flask_cors import CORS  # Import Flask-CORS

app = Flask(__name__)
CORS(app)  # Enable CORS for all origins

app.secret_key = 'your_secret_key'
app.config['SESSION_COOKIE_SECURE'] = True  # Use secure cookies (HTTPS only)
app.config['SESSION_COOKIE_HTTPONLY'] = True  # Prevent client-side JavaScript from
    accessing cookies

@app.route('/', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        session['username'] = request.form['username']
        return redirect(url_for('home'))
    return render_template('login.html')

@app.route('/home')
def home():
    if 'username' in session:
        return 'Logged in as ' + session['username']
    return redirect(url_for('login'))

@app.route('/change_username')
def change_username():
    print("Cookies: ",request.cookies)
    print("Session: ", session)
    if 'username' in session:
        new_username = request.args.get('new_username')
        if new_username:
            session['username'] = new_username
            response = make_response("Username changed successfully")
```

```
            response.headers['Access-Control-Allow-Origin'] = 'http
    ://192.168.0.113:80'  # Allow requests from attack server origin
            return response
        else:
            return "Error: 'new_username' parameter is missing or invalid"
    return redirect(url_for('login'))

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=5000)
```

Access the application in your browser and log in using the login form in 'login.html':

```
<!-- login.html -->
<!DOCTYPE html>
<html>
<body>
    <form method="POST">
        Username: <input type="text" name="username"><br>
        Password: <input type="password" name="password"><br>
        <input type="submit" value="Submit">
    </form>
</body>
</html>
```

2.3 Malicious Server Configuration

Then, open attack.html run by 'attack.py' in another tab or window. This page contains a button that, when clicked, sends a request to change the username to "attacker" on the '/change_username' endpoint of the Flask application.

Click the button in 'attack.html' to execute the attack.Here is code of attack.html:

```

 <!DOCTYPE html>
<html>
<head>
    <title>CSRF Attack</title>
</head>
<body>
    <button onclick="attack()">Click me</button>
    <script>
        function attack() {
            var xhr = new XMLHttpRequest();
            xhr.open("GET", "http://192.168.0.113/change_username?new_username=
    attacker", true);
      xhr.withCredentials = true;
            xhr.send();
        }
    </script>
</body>
</html>
```

The request is sent with the victim's session cookies, allowing the username to be changed without their consent. You can verify the attack's success by checking if the username was changed to "attacker" on the '/home' route or by observing the server logs.

This lab demonstrates a CSRF attack, where an attacker tricks a logged-in user into unknowingly executing malicious actions on a web application. CSRF attacks exploit the fact that the web application trusts the user's browser to include cookies in requests, even if they are initiated by malicious websites.

2.4 Observation

In the network capture, the client (192.168.0.108) is attempting to log in to the server (192.168.0.109) through the proxy (192.168.0.113) in packet 21. The proxy forwards the request to the server, which then processes the login attempt. The user gains authorization by successfully authenticating with the server's login system, which typically involves providing a valid username and password combination shown in 2.1 A malicious website could potentially modify the username by tricking the user into

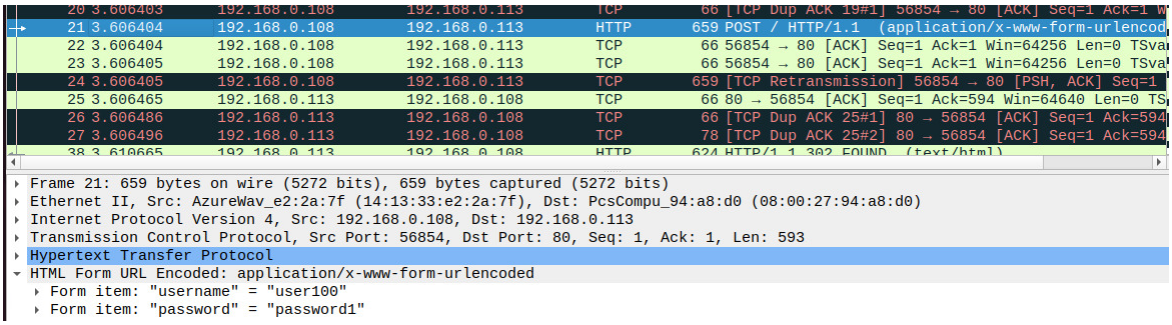

Figure 2.1: Client is authenticating itself to the server

clicking a link or button that sends a request to the proxy with the desired username change.This is happening in packet 77 shown in figure 2.2

Figure 2.2: malicious request from client browser to proxy

If the user is already logged in to the server and the proxy forwards the request with the user's session cookies, the server may mistakenly interpret the request as coming from the legitimate user and change the username to the attacker's specified value. This is happening in packet 87 as shown in figure 2.3

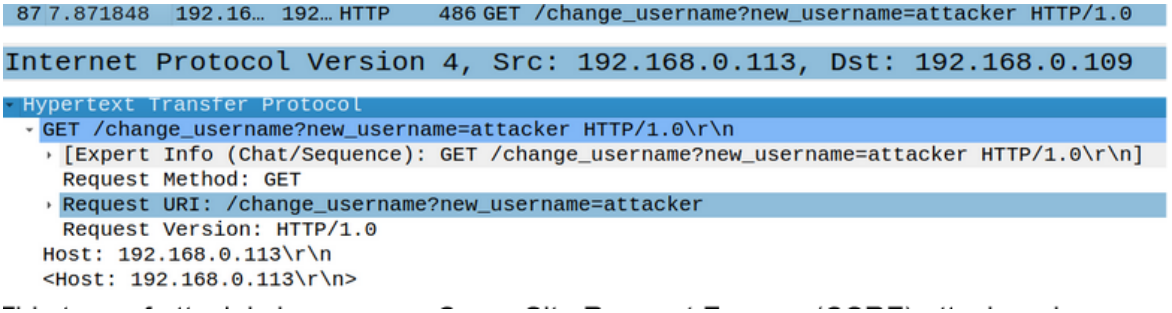

Figure 2.3: malicious request forwarded by proxy to main server

This type of attack is known as a Cross-Site Request Forgery (CSRF) attack and highlights the importance of implementing proper CSRF protection mechanisms in web applications.

2.5 CSRF Mitigation

To mitigate CSRF attacks, web applications can implement measures like CSRF tokens. These tokens are unique for each session and must be included in requests to perform actions, ensuring that requests originate from the legitimate application and not from an attacker's site.

Experimenting with this lab helps understand the importance of CSRF protection and the impact of failing to implement it. It underscores the need for developers to be aware of security best practices and to implement them to protect against such attacks.

2.6 Mitigation Techniques

To mitigate CSRF attacks, several changes were made to the Flask application. First, the Flask-WTF extension was installed to provide CSRF protection and form handling utilities. This extension allows for easy integration of CSRF tokens into forms to prevent CSRF attacks.

Next, the app.py file was modified to use CSRF protection. The CSRFProtect class from Flask-WTF was imported, and an instance of it was created and initialized with the Flask application. This enables CSRF protection for all routes in the application.

A LoginForm class was created using Flask-WTF's FlaskForm as the base class. This form includes fields for the username and password, along with a submit button. Validators were added to ensure that the username and password fields are not empty.

The / route, which handles the login form, was updated to use the LoginForm class to create an instance of the form. If the form is submitted and passes validation, the username is stored in the session and the user is redirected to the home page. Otherwise, the login form is rendered with the form instance.

The /home route, which displays the home page, was modified to check if the user is logged in by verifying if the username is in the session. If the user is logged in, their username is displayed. If not, the user is redirected to the login page.

The `/change_username` route, which allows changing the username, was modified to only accept POST requests and to use `request.form.get('new_username')` to retrieve the new username from the form data. The route is exempted from CSRF protection using `@csrf.exempt` since it is called by an external server for legitimate purposes.

```
1  from flask_wtf import FlaskForm
2  from wtforms import StringField, PasswordField, SubmitField
3  from wtforms.validators import DataRequired
4  from flask import Flask, request, render_template, redirect, url_for, session,
       make_response
5  from flask_wtf.csrf import CSRFProtect  # Import CSRFProtect from Flask-WTF
6
7  app = Flask(__name__)
8  app.secret_key = 'your_secret_key'
9  app.config['SESSION_COOKIE_SECURE'] = True
10 app.config['SESSION_COOKIE_HTTPONLY'] = True
11
12 # Enable CSRF protection for all routes
13 csrf = CSRFProtect(app)
14
15 class LoginForm(FlaskForm):
16     username = StringField('Username', validators=[DataRequired()])
17     password = PasswordField('Password', validators=[DataRequired()])
18     submit = SubmitField('Submit')
19
20 @app.route('/', methods=['GET', 'POST'])
21 def login():
22     form = LoginForm()
23     if form.validate_on_submit():
24         session['username'] = form.username.data
25         return redirect(url_for('home'))
26     return render_template('login.html', form=form)
27
28 @app.route('/home')
29 def home():
30     if 'username' in session:
31         return 'Logged in as ' + session['username']
32     return redirect(url_for('login'))
33
34 @app.route('/change_username', methods=['POST'])
35 @csrf.exempt  # Exempt this route from CSRF protection since it's called by an
       external server
36 def change_username():
37     if 'username' in session:
38         new_username = request.form.get('new_username')
39         if new_username:
40             session['username'] = new_username
41             return "Username changed successfully"
42         else:
43             return "Error: 'new_username' parameter is missing or invalid"
44     return redirect(url_for('login'))
```

```
45
46 if __name__ == "__main__":
47     app.run(host='0.0.0.0', port=5000)
```

Finally, the `login.html` template was modified to include the CSRF token in the form. This is done by including `csrf_token` in the form, which automatically generates and inserts the CSRF token into the form.

```
1  <!-- login.html -->
2  <!DOCTYPE html>
3  <html>
4  <body>
5      <form method="POST">
6          {{ form.csrf_token }}
7          {{ form.username.label }} {{ form.username }}
8          {{ form.password.label }} {{ form.password }}
9          {{ form.submit }}
10     </form>
11 </body>
12 </html>
```

These changes ensure that the Flask application is protected against CSRF attacks by using CSRF tokens in forms and exempting routes that are called by external servers.

Chapter 3

Task3

Single Sign-On (SSO) is an authentication scheme that enables users to access multiple applications or websites with a single set of credentials. It operates on a trust relationship between the service provider (the application) and the identity provider. This method simplifies the login process and enhances security by eliminating the need for users to remember multiple passwords.This lab involves setting up a Single Sign-On (SSO) authentication module on two web applications, hosted on vm-server-A and vm-server-B. The goal is to enable users to log into both applications using SSO with credentials from Google, Facebook, or Twitter. This setup streamlines user access across multiple platforms, enhancing user experience and security.

3.1 Proxy Setup

Proxy setup is left same as the xss attack.

3.2 Server Setup

We implement a web application that uses the Authlib library to implement Google's OAuth2 authentication. It registers Google as an OAuth provider and sets up routes for the home page, login, logout, and authorization. When a user navigates to the login route, they are redirected to Google's OAuth page for authentication. After successful authentication, Google redirects back to the authorize route, where the application retrieves the user's information and stores the email in the session. The home page displays a personalized greeting if the user is authenticated, and the logout route clears the email from the session. The application is then run on a local server.

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import { Auth0Provider } from '@auth0/auth0-react';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <Auth0Provider
    domain="dev-gadx26t3gs4kpjyg.us.auth0.com"
    clientId="x75RkL0xahy18OEg5DVZf6PQnRa5Dc46"
    authorizationParams={{
      redirect_uri: window.location.origin
    }}
  >
    <App />
  </Auth0Provider>
  </React.StrictMode>
);

// If you want to start measuring performance in your app, pass a function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
```

```
reportWebVitals();
```

```
import { useAuth0 } from "@auth0/auth0-react";
import logo from './logo.svg';
import './App.css';

function App() {

  const { user, loginWithRedirect, isAuthenticated, logout } = useAuth0();

  console.log("Current user: ", user)
  return (
    <div className="App">
      {isAuthenticated && <h3> Hello {user.name} </h3>}
      <header className="App-header">
        {
          isAuthenticated ? (
            <button onClick={(e) => logout()}>Logout</button>
          ) : (
            <button onClick={(e) => loginWithRedirect()}>
            Login With Redirect
            </button>
          )
        }

      </header>
    </div>
  );
}

export default App;
```

3.3 Observation

After turning on the server it shows a login page. After clicking on the login page it redirects to the Auth0 sso authentication page, where it asks for your email address and password for secure authentication. If credentials matches then it will redirect to the successful login page where it will show the user's name and there is also a logout button upon clicking it will logout the user and the user need to login again then it will redirect to the sso again for authentication. There is also another option which is you can directly authenticate in sso with your email id by clicking on "continue with Google". Then the auth0 will authenticate and redirect user to the login page.

```
▸ Transmission Control Protocol, Src Port: 51468, Dst Port: 3000, Seq: 1, Ack: 1, Len: 993
▾ Hypertext Transfer Protocol
  ▸ GET /?code=1qmF44e2cx0EyfyGo3NFyUkWVmfDD8-DuG5u2rnE8VXdM&state=TXg0VDhrdFhuZGhXbmVLNGNmQzNwdWxmaTRpekZ2TUZffmxaYzR0
```

Figure 3.1: Requesting for SSO Login

```
  Transaction ID: 0x6df7
▸ Flags: 0x8180 Standard query response, No error
  Questions: 1
  Answer RRs: 0
  Authority RRs: 0
  Additional RRs: 0
▾ Queries
  ▾ play.google.com: type HTTPS, class IN
      Name: play.google.com
      [Name Length: 15]
      [Label Count: 3]
      Type: HTTPS (HTTPS Specific Service Endpoints) (65)
      Class: IN (0x0001)
  [Request In: 79]
  [Time: 0.001879000 seconds]
```

Figure 3.2: Google Authentication Response

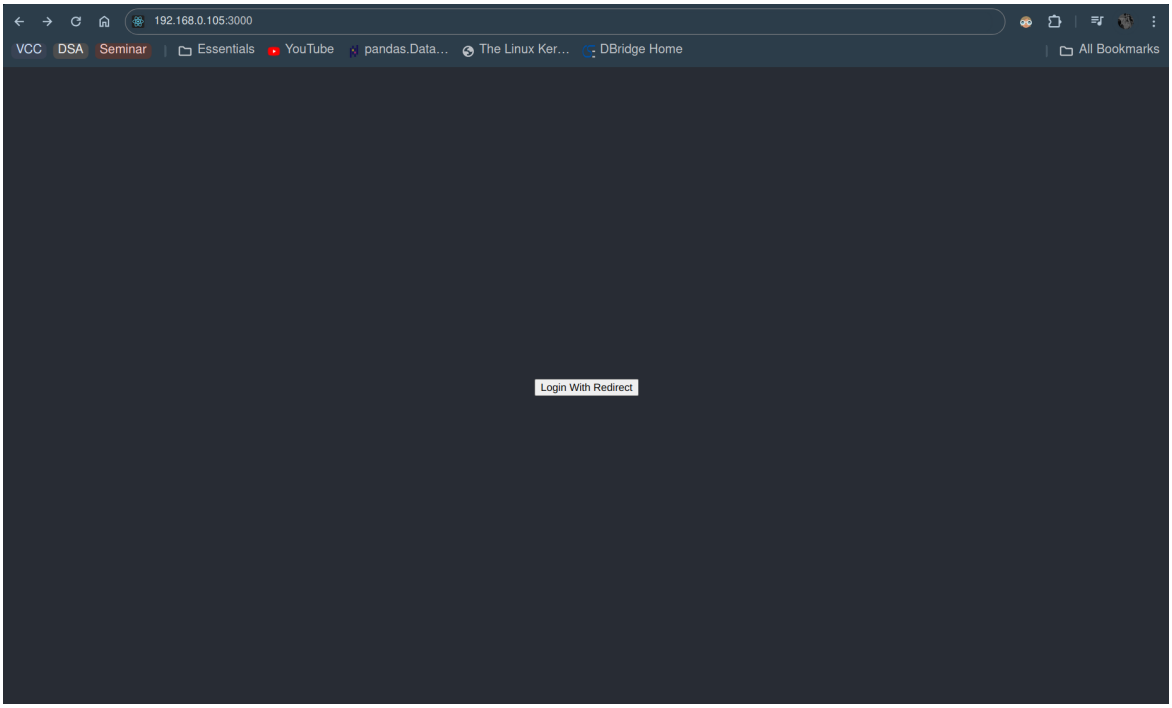

Figure 3.3: Login page for sso authentication

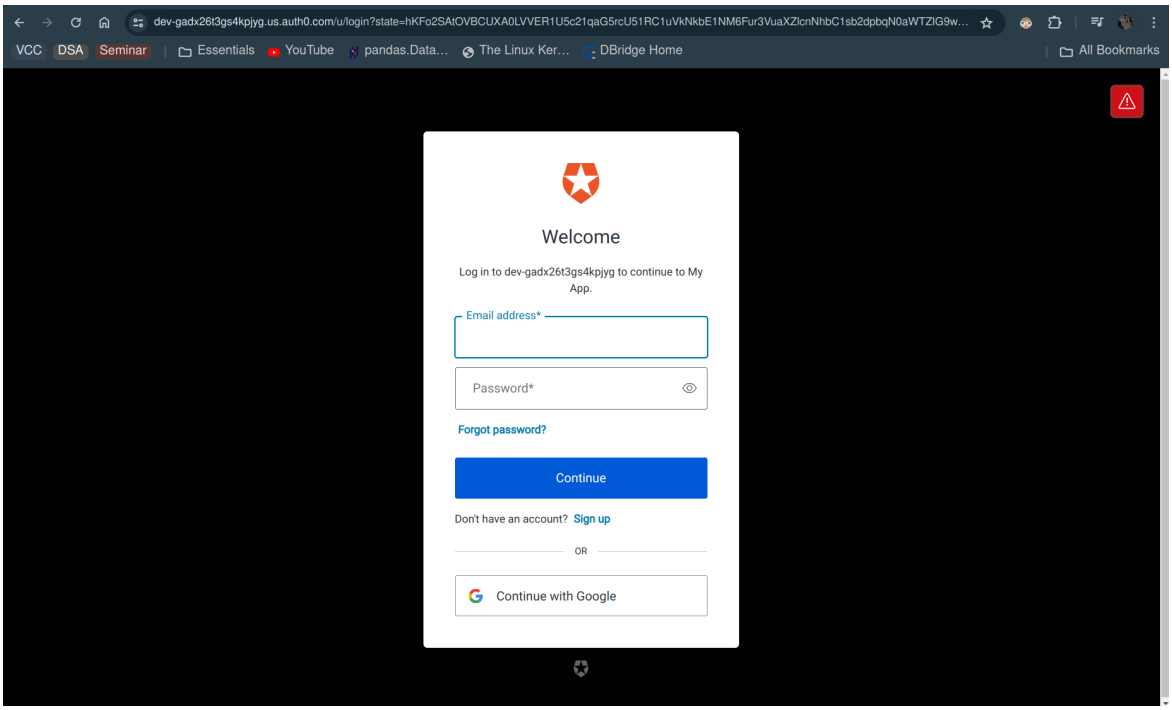

Figure 3.4: SSO Authentication interface

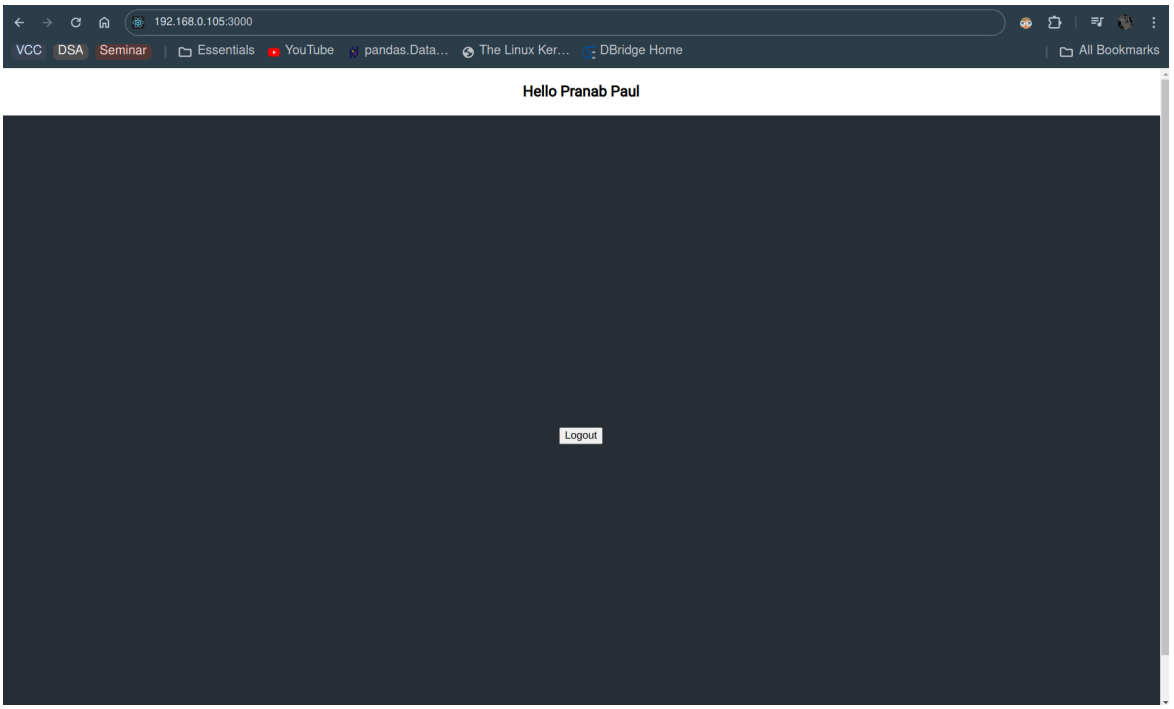

Figure 3.5: Logout page after sso authentication

Chapter 4

Task 4

Self-signed digital certificates are certificates signed by the same entity whose identity they certify, unlike certificates from a trusted certificate authority (CA). They are commonly used for testing and internal purposes, but are not trusted by default in web browsers or applications, making them unsuitable for public-facing websites or applications requiring trust from a wide range of users. Users will see security warnings when accessing sites secured by self-signed certificates.A self-signed digital certificate installed on a server and configured to use HTTPS encrypts all communication between the server and other entities, preventing data from being read by anyone intercepting the traffic, including a proxy. The proxy receives encrypted data instead of plaintext, preventing it from reading the contents of the communication. This encryption protects the confidentiality and integrity of the communication, ensuring the security of sensitive information.

4.1 Generating self signed certificates for Servers

To generate self-signed digital certificates for vm-server-A and vm-server-B, we can use the following commands.

```
    #  First, we generate a private key for each server
    openssl genrsa -out vm-server-a.key 2048
    openssl genrsa -out vm-server-b.key 2048

    # Next, we create a certificate signing request (CSR) for each server using the
    private keys:
openssl req -new -key vm-server-a.key -out vm-server-a.csr
openssl req -new -key vm-server-b.key -out vm-server-b.csr


# Then, generate self-signed certificates for each server using the CSR and private
    key:


openssl x509 -req -days 365 -in vm-server-a.csr -signkey vm-server-a.key -out vm-
    server-a.crt
openssl x509 -req -days 365 -in vm-server-b.csr -signkey vm-server-b.key -out vm-
    server-b.crt
```

We can then use the generated ‘vm-server-a.crt‘ and ‘vm-server-a.key‘ files for vm-server-A, and ‘vm-server-b.crt‘ and ‘vm-server-b.key‘ files for vm-server-B, to configure Apache or other services to use these self-signed certificates for secure communication.

4.2 Proxy Setup

: In this network setup, there are two servers, ‘server1.in‘ and ‘server2.in‘, configured to listen on HTTPS port 443. Both servers use the same SSL certificate (‘certP.pem‘) and private key (‘keyP.pem‘). The first server (‘server1.in‘) proxies requests to ‘https://192.168.0.109:5000‘, while the second server (‘server2.in‘) proxies requests to ‘https://192.168.0.106:5000‘. The proxy configuration includes headers to pass the original host, client IP address, and protocol, ensuring that the proxied requests appear as if they were directly sent to the backend servers. The ‘proxy_ssl_session_reuse on;‘

directive enables session reuse for better performance. This setup allows both servers to act as reverse proxies, forwarding HTTPS requests to the specified backend servers.

```
...

    server {
        listen 443 ssl;
        server_name server1.in;

        ssl_certificate certP.pem;
        ssl_certificate_key keyP.pem;

        location / {
            proxy_pass https://192.168.0.109:5000;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Proto $scheme;

            # Configure proxy to pass HTTPS traffic correctly
            proxy_redirect off;
            proxy_ssl_session_reuse on;
            proxy_set_header Connection '';
            proxy_http_version 1.1;
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Connection "upgrade";
        }
    }

    server {
        listen 443 ssl;
        server_name server2.in;

        ssl_certificate certP.pem;
        ssl_certificate_key keyP.pem;

        location / {
            proxy_pass https://192.168.0.106:5000;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Proto $scheme;

            # Configure proxy to pass HTTPS traffic correctly
            proxy_redirect off;
            proxy_ssl_session_reuse on;
            proxy_set_header Connection '';
            proxy_http_version 1.1;
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Connection "upgrade";
        }
    }
}
```

4.3 Server Setup

The ‘app.py‘ file is crucial to this setup as it defines the behavior of the Flask application that serves as the backend for the web server. It contains routes that handle incoming requests, such as displaying the home page (‘/‘ route) and submitting messages (‘/submit‘ route). The file also includes logic to interact with the ‘user_data.json‘ file, storing and retrieving messages submitted by users. Additionally, it defines the SSL context for running the Flask server in HTTPS mode, ensuring secure communication between the client and the server. Overall, ‘app.py‘ controls the functionality and behavior of the web application, making it an essential component of the setup. In this way we set up vm-server-a.It’s similar set up for vm-server-b.

```
from flask import Flask, render_template, request, jsonify
import json
import html  # Import html module for escaping HTML characters

app = Flask(__name__)
```

```
6
7  # File to store user messages
8  USER_DATA_FILE = 'user_data.json'
9
10 # Load existing user data from JSON file or initialize if file not found
11 try:
12     with open(USER_DATA_FILE, 'r') as f:
13         user_data = json.load(f)
14 except FileNotFoundError:
15     user_data = []
16
17 @app.route('/')
18 def home():
19     return render_template('index.html', messages=user_data)
20
21 @app.route('/submit', methods=['POST'])
22 def submit():
23     if request.method == 'POST':
24         message = request.form['message']
25         print(message)
26
27         # Store the message in user_data
28         user_data.append(message)
29
30         # Save updated user data to JSON file
31         with open(USER_DATA_FILE, 'w') as f:
32             json.dump(user_data, f, indent=4)
33
34         # Return a response that includes the message for potential XSS execution
35 #        return f"<script>alert('{html.escape(message)}');</script>"
36
37     return home()
38
39 if __name__ == '__main__':
40     app.run(host="0.0.0.0", port=5000, debug=True, ssl_context=('certA.pem', 'keyA.
       pem'))
```

4.4 When client makes a request to the server

When the client requests vm-server-A (similar for vm-server-B), first the request is passed to the proxy and then the server's self signed certificate is provided to the client. In order to authenticate the server's identity the client needs to have the server's self signed digital certificate beforehand.

When the server's certificate is provided to the client(shown in figure 4.1) at the time of requesting, the client verifies the certificate with the help of the following code

```
1      from OpenSSL import SSL
2
3  # Load the server's certificate
4  server_cert = SSL.load_certificate(SSL.FILETYPE_PEM, open('server_cert.pem').read())
5
6  # Load the presented certificate (received during handshake)
7  presented_cert = SSL.load_certificate(SSL.FILETYPE_PEM, open('presented_cert.pem').
       read())
8
9  # Compare the certificates
10 if server_cert.digest('sha256') == presented_cert.digest('sha256'):
11     print("The server's certificate is verified.")
12 else:
13     print("The server's certificate could not be verified.")
```

Now the server is authenticated to the client. Now they start transferring application data.

4.5 Observation

After secure connection is established the data transferred between the client and the server is fully encrypted. In packet 27 in traffic-task-4.pcap, the data transferred from client(192.168.0.114) to server(192.168.0.113) through proxy is completely encrypted as we can see in figure 4.2

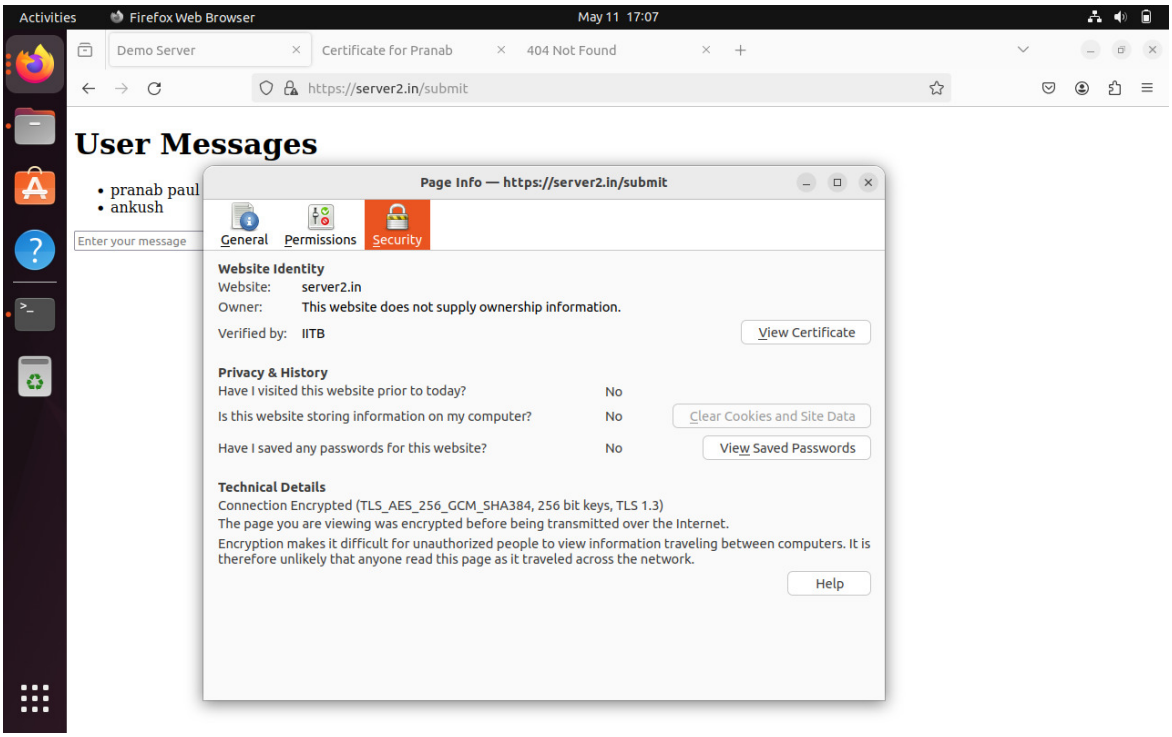

Figure 4.1: Server certificate provided to client

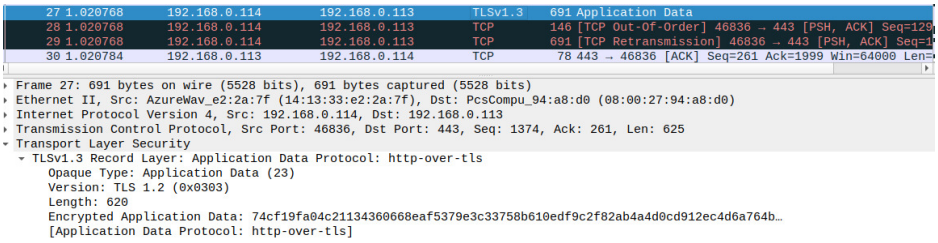

Figure 4.2: Encrypted data transfer from client to server

In packet 32 in traffic-task-4.pcap, the data transferred from server(192.168.0.113) to client(192.168.0.114) through proxy is completely encrypted as we can see in figure 4.3

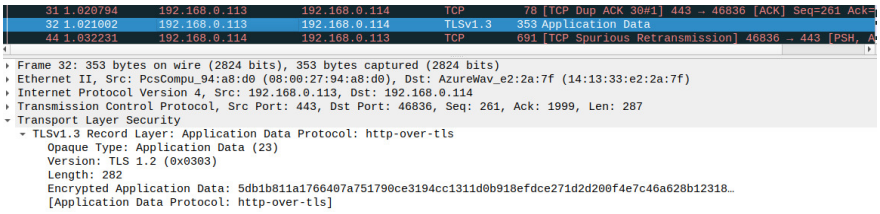

Figure 4.3: Encrypted data transfer from server to client

Chapter 5

Task 5

In this task we first disable the Single-Sign-On authentication module. We will use client-side digital certificates to authenticate the client to the proxy server.

5.1 Server Configuration

Servers are running flask applications, below is the code for app.py in vm-server-A application and the other server also have similar code :

```
1
2  from flask import Flask, render_template, request, jsonify
3  import json
4  import html  # Import html module for escaping HTML characters
5
6  app = Flask(__name__)
7
8  # File to store user messages
9  USER_DATA_FILE = 'user_data.json'
10
11 # Load existing user data from JSON file or initialize if file not found
12 try:
13     with open(USER_DATA_FILE, 'r') as f:
14         user_data = json.load(f)
15 except FileNotFoundError:
16     user_data = []
17
18 @app.route('/')
19 def home():
20     return render_template('index.html', messages=user_data)
21
22 @app.route('/submit', methods=['POST'])
23 def submit():
24     if request.method == 'POST':
25         message = request.form['message']
26         print(message)
27
28         # Store the message in user_data
29         user_data.append(message)
30
31         # Save updated user data to JSON file
32         with open(USER_DATA_FILE, 'w') as f:
33             json.dump(user_data, f, indent=4)
34     return home()
35
36 if __name__ == '__main__':
37     app.run(host="0.0.0.0", port=5000, debug=True,
38     ssl_context=('serverA.crt', 'serverA.key'))
```

5.2 Generating Client Certificate

In the client side we will first generate client certificate by following commands:

```
openssl genrsa -out client.key 2048
openssl req -new -key client.key -out client.csr

# Sign the client certificate with CA
openssl x509 -req -in client.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out
client.crt -days 365 -sha256
openssl pkcs12 -export -out client.p12 -inkey client.key -in client.crt
```

5.3 Proxy Configuration

The proxy will authenticate the client using its digital certificate signed by the CA. For that we need to make changes in the nginx.conf file present in the proxy server.The changes are as follows:

```
...

    server {
        listen 443 ssl;
        server_name server1.in;

        ssl_certificate certP.pem;
        ssl_certificate_key keyP.pem;

  ssl_client_certificate ca.crt;
  ssl_verify_client on;

        location / {
            proxy_pass https://192.168.0.109:5000;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Proto $scheme;

            # Configure proxy to pass HTTPS traffic correctly
            proxy_redirect off;
            proxy_ssl_session_reuse on;
            proxy_set_header Connection '';
            proxy_http_version 1.1;
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Connection "upgrade";
        }
    }

    server {
        listen 443 ssl;
        server_name server2.in;

        ssl_certificate certP.pem;
        ssl_certificate_key keyP.pem;

  ssl_client_certificate ca.crt;
        ssl_verify_client on;

        location / {
            proxy_pass https://192.168.0.106:5000;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Proto $scheme;

            # Configure proxy to pass HTTPS traffic correctly
            proxy_redirect off;
            proxy_ssl_session_reuse on;
            proxy_set_header Connection '';
            proxy_http_version 1.1;
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Connection "upgrade";
        }
    }
}
```

5.4 Client sending request

:

After using the client certificate

Before requesting to two servers the client will add its certificate(Certificate Name: Pranab) and CA's certificate in his/her browser:

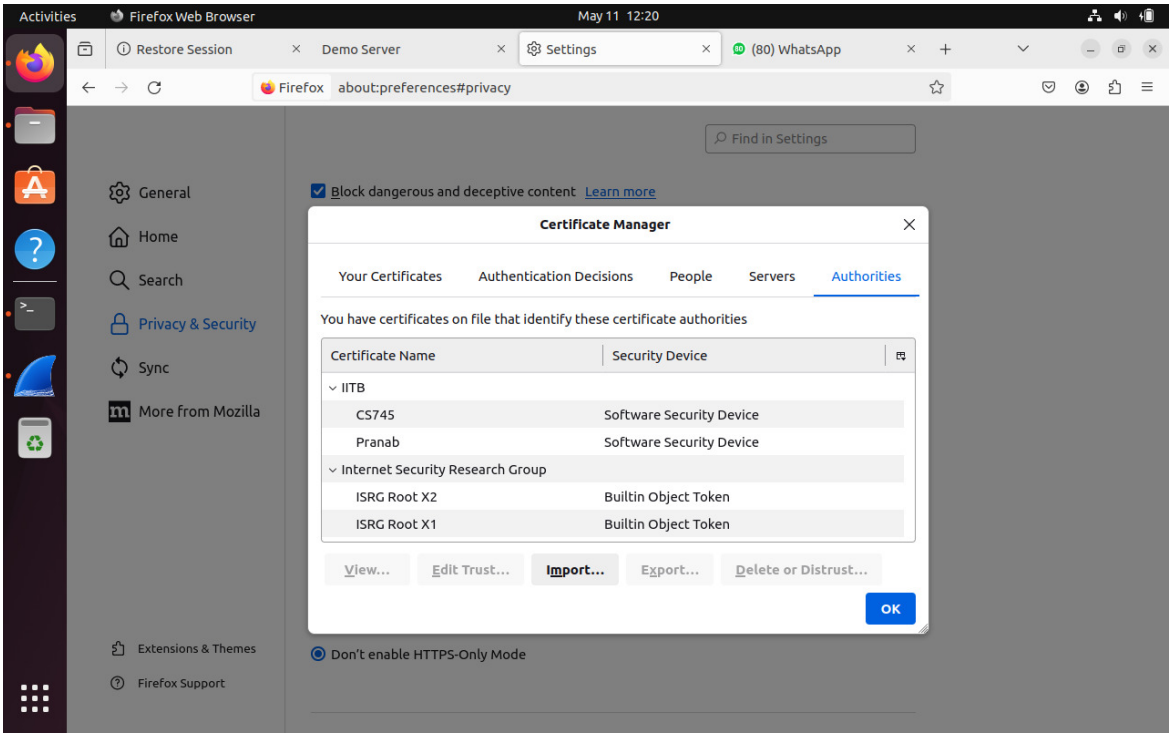

Figure 5.1: CA entry in client browser

Before requesting to server, client can check the server's certificate and verify it shown in figure 5.3

Figure 5.4 is the server response after authentication:

5.5 Without using the client certificate

If the client does not provide the client certificate or provides wrong certificate (e.g CS745) to the proxy at the time of requesting the server then the client will not be allowed to reach the server at all shown in figure 5.5 and 5.6

In that case the server response would be like figure 5.7

5.6 Conclusion

In this way the proxy can authenticate the client using its digital certificate. We also save the traffic from client to two servers vm-server-A and vm-server-B in vm-proxy as traffic-task-5.pcap

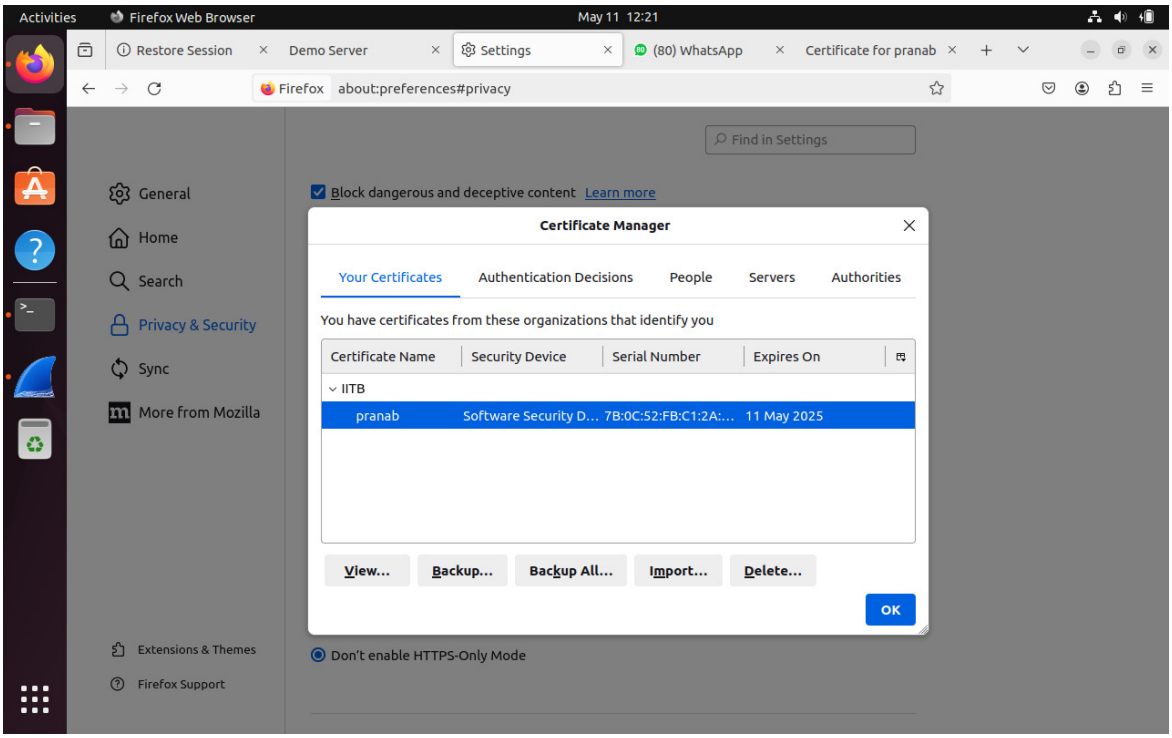

Figure 5.2: Client certificate in client browser

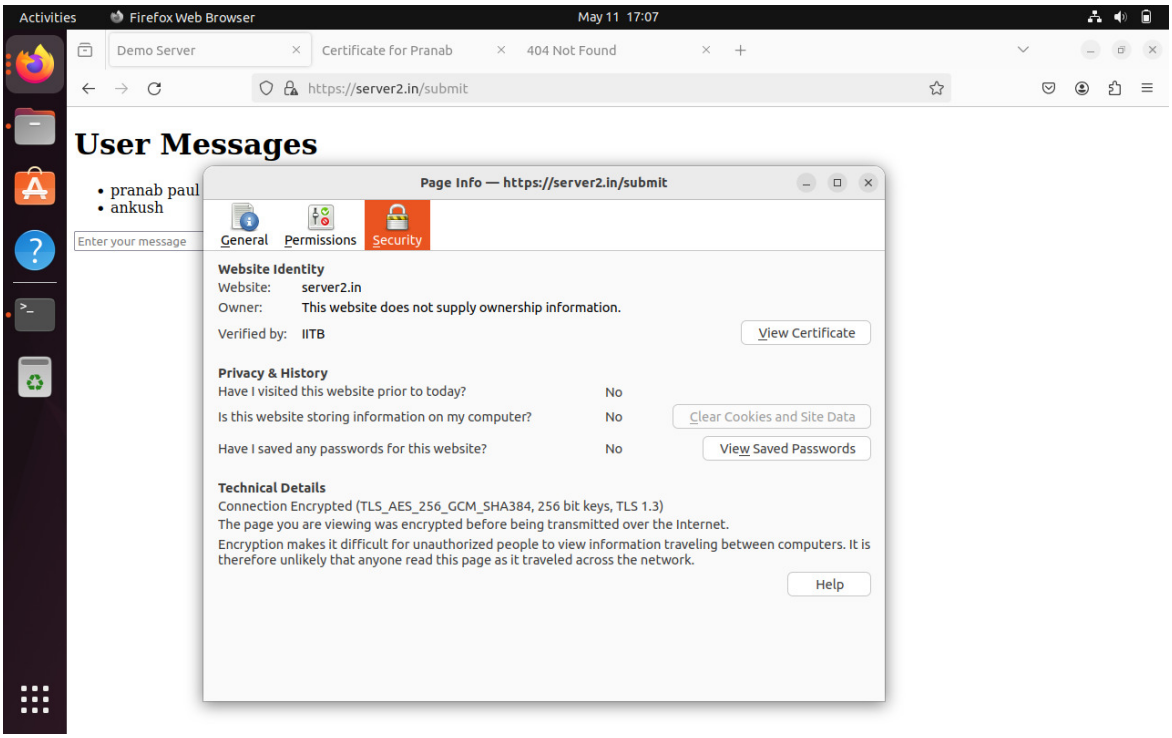

Figure 5.3: Server certificate in client browser

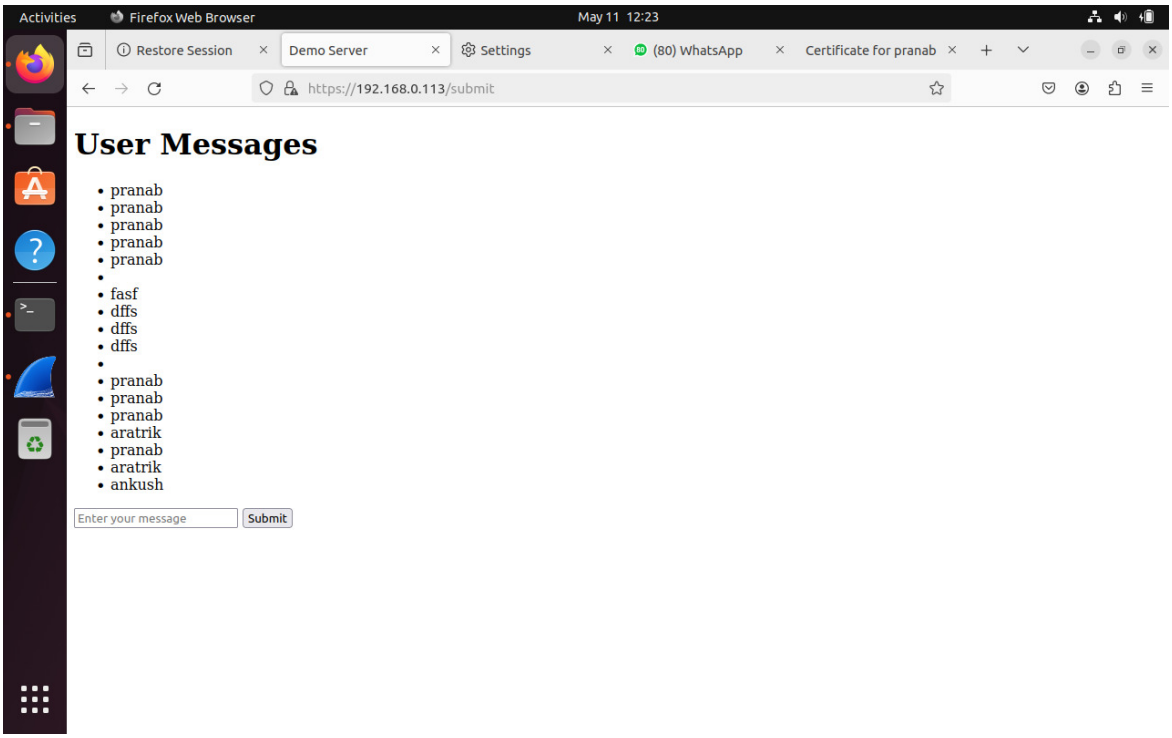

Figure 5.4: Server certificate in client browser

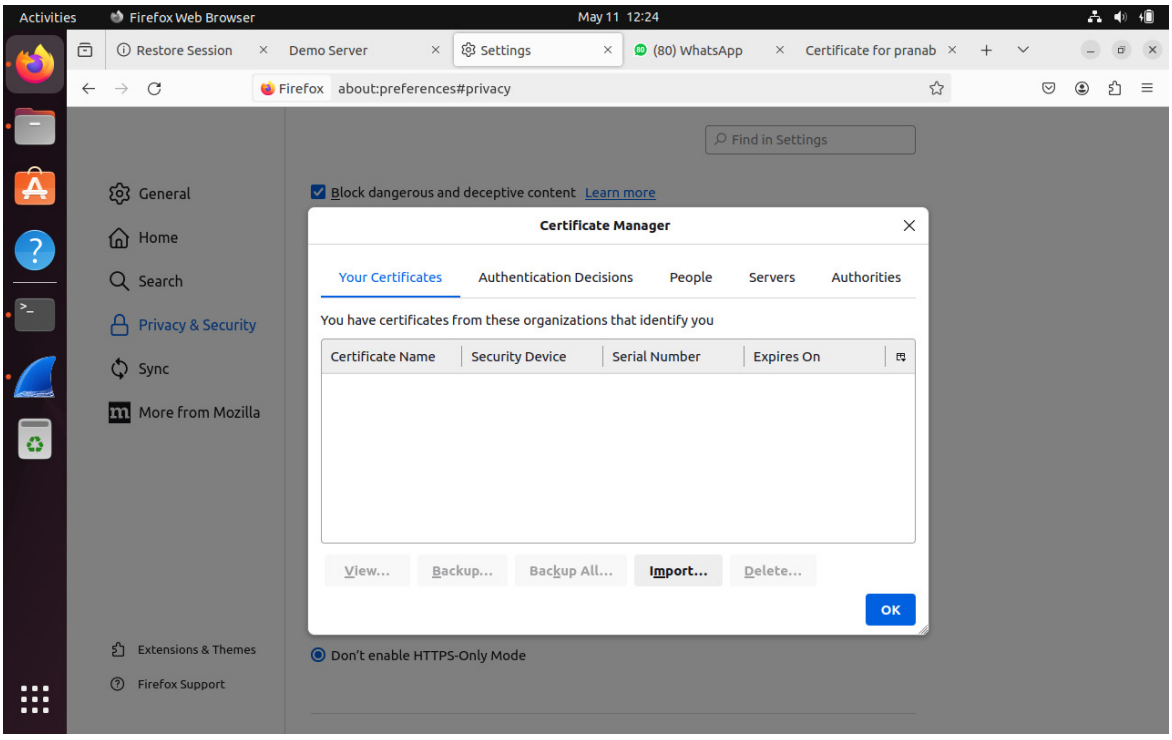

Figure 5.5: Client certificate not present in client browser

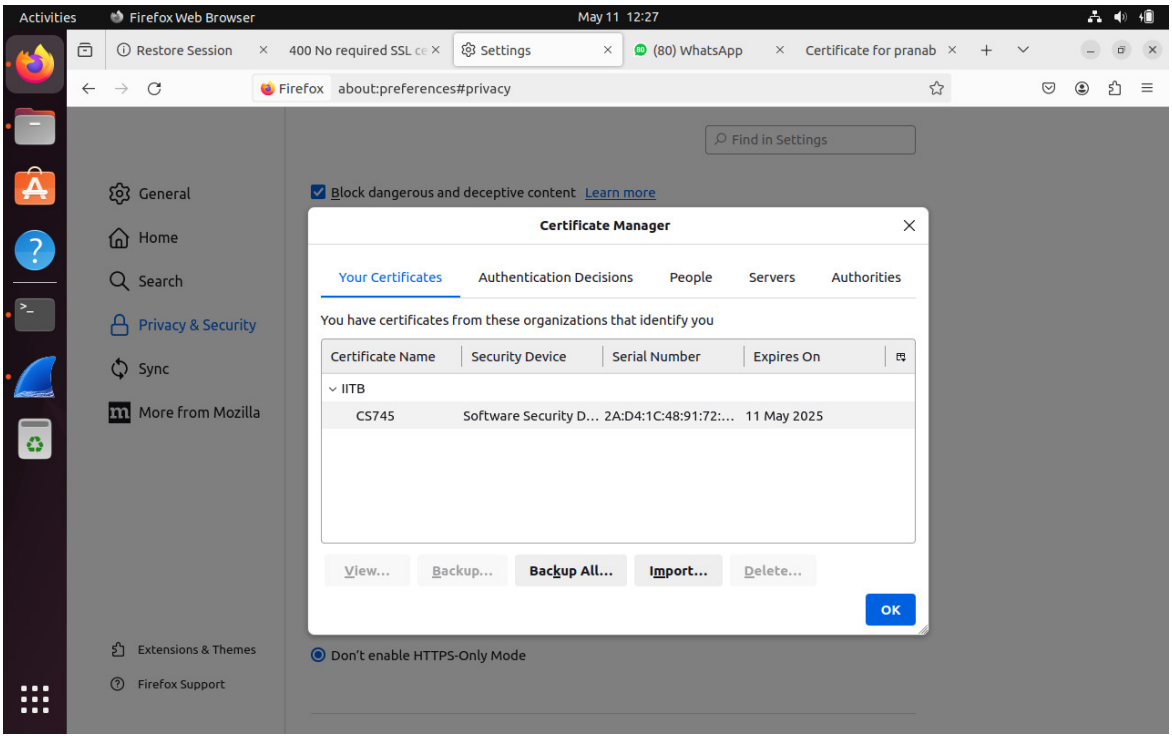

Figure 5.6: Wrong Client certificate(CS745) in client browser

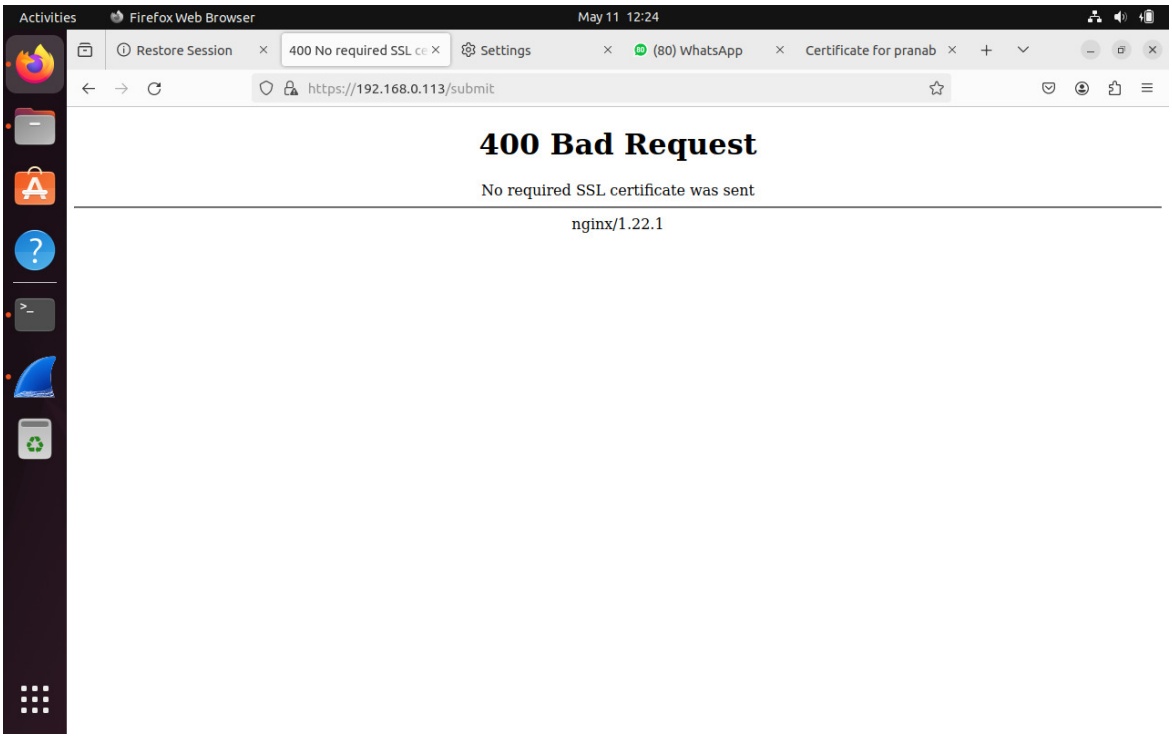

Figure 5.7: server response when client certificate not present or correct

Chapter 6

Task6

In this task we want to modify the proxy to act as a transparent SSL-proxy. In other words, when the proxy becomes a SSL-proxy, it establishes two SSL tunnels on either side of it. That is: Browser wants to establish a secure (SSL/HTTPS) connection with the web server but the proxy intercepts this request between Browser and Web server and establishes one SSL connection with the Browser pretending to be the intended web-server and another SSL connection with the web server to pretend as Browser.

6.1 Workflow

The client has self signed digital certificate which it will send to the proxy at the time of establishing the connection between client and proxy. The proxy first authenticates the client certificate.For authentication proxy needs to have the client certificate beforehand.

Next the proxy will send its self signed certificate to the server for authenticating itself to the server.For that the server needs to have the proxy's certificate beforehand. When the connection is established between client and proxy and between proxy and server data can be transferred between client and server via the proxy will act as a Man-In-The-Middle.

6.2 Proxy Setup

```
1     ##
2 ## tinyproxy.conf -- tinyproxy daemon configuration file
3 ##
4 ## This example tinyproxy.conf file contains example settings
5 ## with explanations in comments. For decriptions of all
6 ## parameters, see the tinproxy.conf(5) manual page.
7
8 User tinyproxy
9 Group tinyproxy
10
11 # ...
12
13 Port 8888
14
15 Listen 192.168.0.103
16
17 Timeout 600
18
19 DefaultErrorFile "/usr/share/tinyproxy/default.html"
20
21 StatFile "/usr/share/tinyproxy/stats.html"
22
23 Syslog On
24
25 MaxClients 100
26
27 Allow 127.0.0.1
28 Allow ::1
29 Allow 192.168.0.103/24
```

```
#Allow 192.168.0.0/16
#Allow 172.16.0.0/12
#Allow 10.0.0.0/8

ViaProxyName "tinyproxy"

FilterURLs Off

FilterDefaultDeny Yes

# Anonymous:
#Anonymous "Host"
#Anonymous "Authorization"
#Anonymous "Cookie"

# ConnectPort:

#ReversePath "/google/" "http://www.google.com/"
#ReversePath "/wired/"  "http://www.wired.com/"

#ReverseOnly Yes

#ReverseMagic Yes

#ReverseBaseURL "http://localhost:8888/"
```

6.3 Server Setup

```
    from flask import Flask, render_template, request, jsonify
import json
import html  # Import html module for escaping HTML characters

app = Flask(__name__)

# File to store user messages
USER_DATA_FILE = 'user_data.json'

# Load existing user data from JSON file or initialize if file not found
try:
    with open(USER_DATA_FILE, 'r') as f:
        user_data = json.load(f)
except FileNotFoundError:
    user_data = []

@app.route('/')
def home():
    return render_template('index.html', messages=user_data)

@app.route('/submit', methods=['POST'])
def submit():
    if request.method == 'POST':
        message = request.form['message']
        print(message)

        # Store the message in user_data
        user_data.append(message)

        # Save updated user data to JSON file
        with open(USER_DATA_FILE, 'w') as f:
            json.dump(user_data, f, indent=4)

    return home()

if __name__ == '__main__':
    app.run(host="0.0.0.0", port=5000, debug=True, ssl_context=('certA.pem', 'keyA.
    pem'))
```

6.4 Observation

When the client browser wants to establish a secure (SSL/HTTPS) connection with the web server then our proxy intercepts this request between Browser and Web server and establishes one SSL

connection with the Browser pretending to be the intended web-server and another SSL connection with the web server to pretend as Browser.

We can observe this by analyzing packets in traffic-task-6.pcap file.

- For the connection between client and proxy a **Client Hello** is sent from client to proxy as we can see in packet 229 in figure 6.1

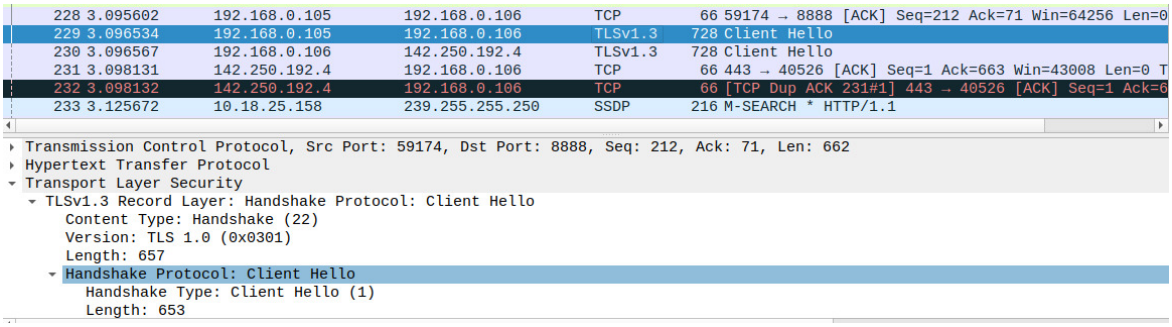

Figure 6.1: Client Hello from client to proxy

- For the connection between proxy and server a **Client Hello** is sent from proxy to server as we can see in packet 230 in figure 6.2

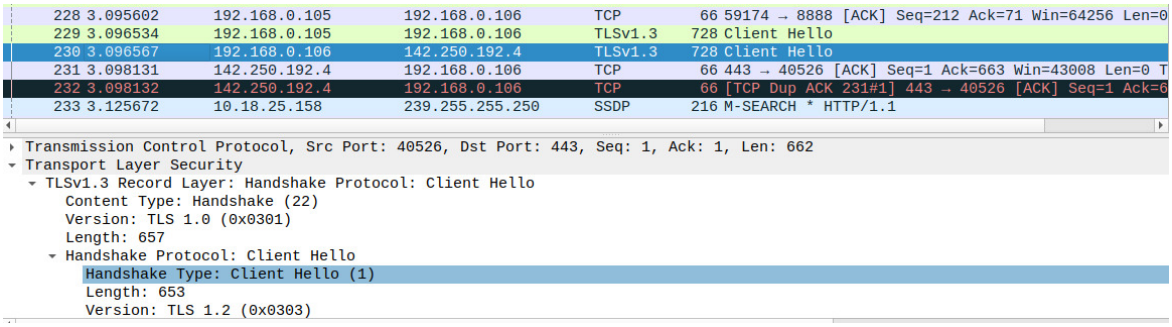

Figure 6.2: Client Hello from proxy to server

- For the connection between proxy and server a **Server Hello** is sent from server to proxy as we can see in packet 235 in figure 6.3

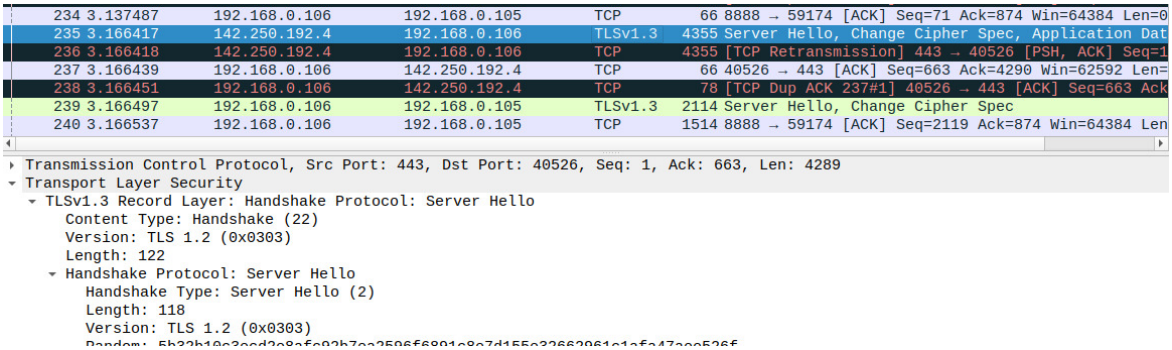

Figure 6.3: Server Hello from server to proxy

- For the connection between client and proxy a **Server Hello** is sent from proxy to client as we can see in packet 239 in figure 6.4

  After two secure connections are established : one is between client and proxy and one is between proxy and server data transfer can be started.

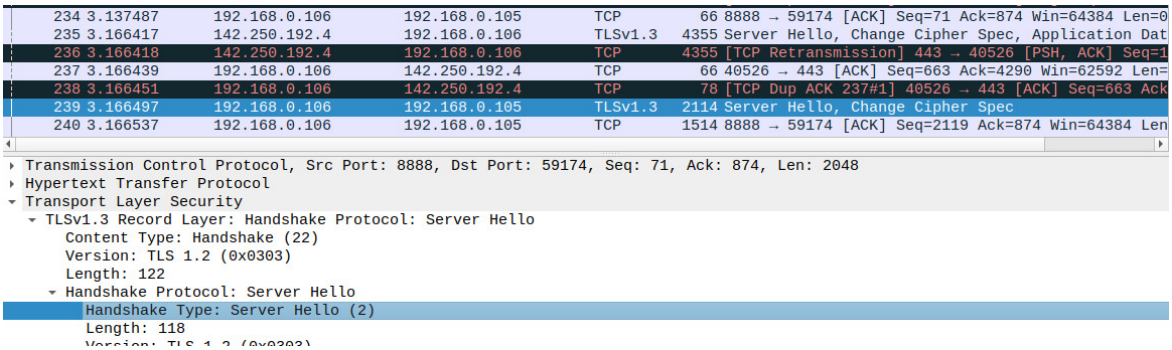

Figure 6.4: Server Hello from proxy to client

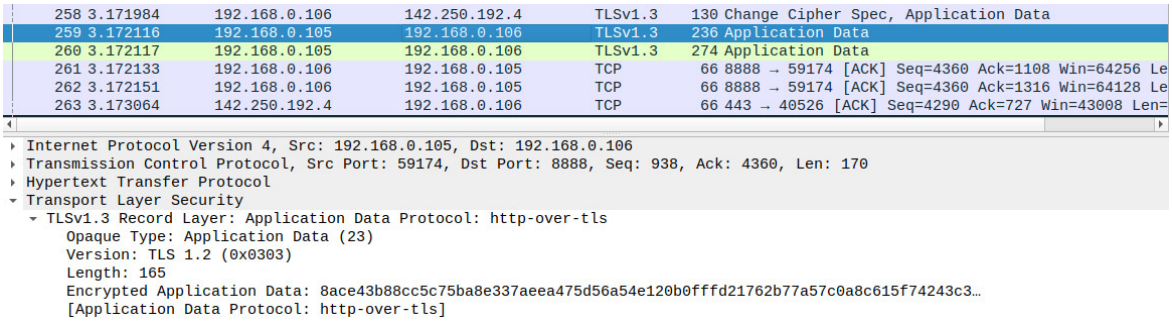

Figure 6.5: Data transfer (request) from client to proxy

- Client makes a request to the proxy in packet 259 shown in figure 6.5
- Proxy makes a request acting as client to the server in packet 265 shown in figure 6.6

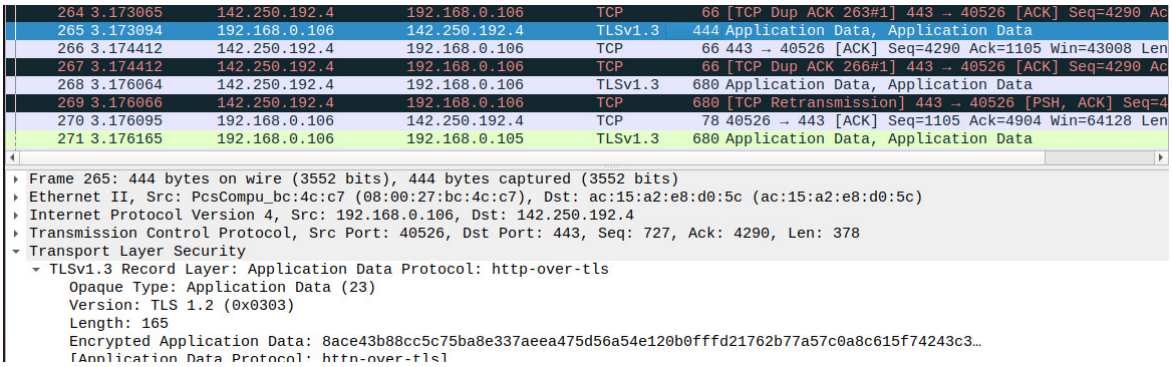

Figure 6.6: Data transfer (request) from proxy to server

- Server returns a response to the proxy in packet 268 shown in figure 6.7
- Proxy acting as a server returns a response to the client in packet 271 shown in figure 6.8

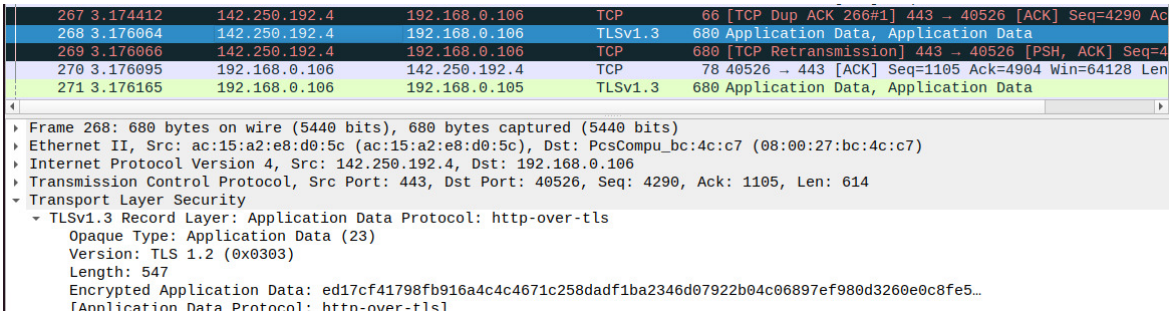

Figure 6.7: Data transfer(response) from server to proxy

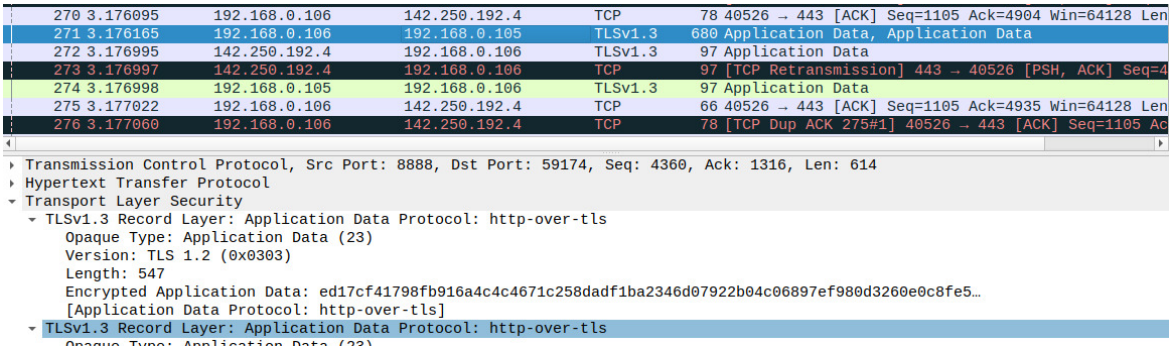

Figure 6.8: Data transfer(response) from proxy to client