CS 745 Project

Team Name: Encryption_Squad

Team Members:

Arnab Bhakta (23M0835)

Santanu Sahoo (23M0777)

Soumik Dutta (23M0826)

Atul Kumar (23M0764)

Department of Computer Science and Engineering, IITB

** Collaborated with Team NullByte for webserver and proxy implementation **

Task 1: Cross-Site Scripting

Configuration:

We've set up a virtual machine called vm-server-a, running a bulletin board web application built using the Flask framework. It's important to note that this application lacked adequate input sanitization measures. Both legitimate users and potential attackers had access to the same board, creating an opportunity for XSS attacks.

Attack:

The attacker inputs JavaScript code into the text field of the bulletin board instead of regular text. When the victim visits the website, their browser executes the script, triggering an alert. This exploit enabled the injection of malicious scripts into the web application.

Mitigation:

To counter the XSS vulnerability, we implemented input sanitization measures within the Flask web application. Specifically, we activated input sanitization functionality, which automatically escapes any input containing HTML tags. This action effectively reduces the risk of script execution and prevents potential XSS attacks.

Observation:

After implementing the input sanitization measures, we noticed that the script appeared as plain text, and the browser refrained from executing it. This successful mitigation strategy effectively guards against XSS attacks.

Task 2: Cross-Site Request Forgery

Configuration:

In Task 2, we utilized a virtual machine named vm-server-a to host a bulletin board web application. Additionally, we introduced another web application, such as phpBB, controlled by an attacker, possibly running on a separate virtual machine. Within the bulletin board web application, posts were equipped with a delete button capable of removing the associated post solely based on the post ID.

Exploitation:

The vulnerability emerged when the attacker embedded the URL for deleting other users' posts into their own site. By enticing unsuspecting users to click on this URL, the browser inadvertently sent the request to the Flask web application along with the user's login cookie, thus enabling the unauthorised deletion of posts.

Remediation:

To counter the CSRF vulnerability and thwart unauthorised post deletions, we implemented a robust CSRF token mechanism within the Flask web application. This CSRF token is dynamically generated and unique to each user session. Consequently, cross-site requests cannot predict or guess this token beforehand. Consequently, attempts to access sensitive endpoints like /delete without the correct CSRF token are thwarted, significantly enhancing the security posture of the application.

Observation:

Once the token mechanism is implemented, the attacker can no longer predict the URL and thus cannot generate a link beforehand. This prevention of CSRF attacks occurred because there was no way for the attacker to ascertain the CSRF token in advance.

Task 3: Implementation of Google Single Sign-On (SSO)

- ❖ Installation of Single-Sign-On Authentication Module:
  - ➢ We installed a Single-Sign-On authentication module on both our web applications, namely vm-server-A and vm-server-B.
  - ➢ The module chosen for implementation was Google SSO.
- ❖ Integration of Google SSO:
  - ➢ Google SSO was integrated into both application-A and application-B.
  - ➢ Users were provided with the option to log in using their Google credentials.
  - ➢ Upon logging in, users were authenticated via Google SSO and granted access to the respective applications.
- ❖ Configuration of SSO Client at Google Cloud:
  - ➢ We configured the SSO client at Google Cloud to enable authentication for our web applications.
  - ➢ The steps involved in this configuration process were as follows:
    - ■ Creating OAuth 2.0 Credentials: We created OAuth 2.0 credentials in the Google Cloud Console, including the client ID and client secret.
    - ■ Choosing Client Type: As our applications were server-side applications, we selected the "web" client type for the SSO client configuration.
    - ■ Specifying Redirect URIs: We specified the redirect URIs for both vm-server-A and vm-server-B in the SSO client configuration.
    - ■ Setting Scopes: The appropriate scopes were configured to grant necessary permissions for accessing user information during the authentication process.
    - ■ Downloading Client Configuration: Once the SSO client was configured, we downloaded the client configuration file containing the necessary credentials.
- ❖ Configuration of Flask Server and Python Code:

    - We configured the Flask server and implemented the necessary Python code to enable Google SSO authentication.
    - Key configuration and code parts included:
        - Setting Up Flask App: Creation of Flask app object, setting secret key, and configuring CORS.
        - Database Configuration: Configuration of database URI using SQLAlchemy.
        - OAuth 2.0 Flow Setup: Initialization of OAuth 2.0 flow using Flow class from google_auth_oauthlib.
        - Login Route Implementation: Implementation of /login route to initiate the login process and redirect users to Google OAuth consent screen.
        - Callback Route Implementation: Implementation of /callback route to handle callback from Google after successful authentication, exchange authorization code for tokens, and verify ID token.
        - Logout Route Implementation: Implementation of /logout route to clear session data and log the user out.
        - Protected Area Route Implementation: Implementation of /protected_area route that requires authentication using a decorator (login_is_required) and renders the protected area.
        - Error Handling: Implementation of appropriate error handling for cases such as unauthorized access or mismatched states.
- Testing the SSO Functionality:
    - We thoroughly tested the SSO functionality by logging into both application-A and application-B using Google SSO.
    - The login process was smooth, and users were successfully authenticated without any issues.

Task 4: Installing a Self-Signed Certificate on the Server

Generating the Self-Signed Certificate

1. Create a Private Key

Generate a 2048-bit private key using the command: openssl genrsa -out server-a-selfsigned.key 2048

2. Generate a Certificate Signing Request (CSR)

Create a CSR using the private key. You'll be prompted to enter details like country, state, locality, organization, etc. Alternatively, provide this information via an input file (input.txt):

openssl req -new -key server-a-selfsigned.key -out server-a-selfsigned.csr < input.txt

3. Generate the Self-Signed Certificate

Use the CSR to generate a self-signed certificate, signed with the private key:

openssl x509 -req -in server-a-selfsigned.csr -signkey server-a-selfsigned.key -out server-a-selfsigned.crt

Configuring Apache

1. Update Apache Configuration File

In the Apache configuration file (e.g.,etc/apache2/sites-available/your_site.conf), add the necessary SSL configuration within the VirtualHost block. Specify the SSLCertificateFile and SSLCertificateKeyFile directives to point to the generated server-a-selfsigned.crt and server-a-selfsigned.key files, respectively

2. Ensure SSL Configuration

Set SSLEngine to "on" to enable SSL/TLS encryption. Include additional configurations like SSLCertificateChainFile for intermediate certificates, SSLProtocol for allowed protocol versions, and SSLCipherSuite for acceptable encryption algorithms.

3. Restart Apache

Save the configuration file and restart Apache using sudo systemctl restart apache2 to apply the changes.

This configuration enables Apache to use the self-signed certificate for establishing secure HTTPS connections, enhancing the security of the website's communications.

Task 5: Using a Client Certificate for Authentication

Generating the Client Certificate

1. Create a Client Key

Generate a client key using the command: openssl genrsa -out client.key 2048

2. Generate a Certificate Signing Request (CSR)

Create a CSR using the client key:

```
openssl req -new -key client.key -out client.csr < input.txt
```

3. Generate the Client Certificate

Use the CSR to generate a client certificate signed by the server's

self-signed certificate authority (CA):

```
   openssl x509 -req -in requests/client.csr -CA server-a-selfsigned.crt
-CAkey server-a-selfsigned.key -CAcreateserial -out requests/client.crt
-days 365
```

Configuring the Browser

To enable client certificate authentication, the client certificate was imported into web browsers like Google Chrome, Mozilla Firefox, and Microsoft Edge. The specific steps involved navigating to the certificate management settings in each browser, importing the client.crt certificate and client.key private key into the "Personal" or "Your Certificates" tab.

Once imported, the web browsers were configured to use the client certificate for authentication when accessing websites that require it, ensuring secure communication between the client and server.

Task 6: Transparent SSL Proxy

Setup

- The existing vm-server-a virtual machine was maintained.

- A new virtual machine called vm-proxy was introduced.

- mitmproxy was installed and executed on vm-proxy.

- On vm-client, the client's environment was configured to route all traffic through vm-proxy by setting the http_proxy and https_proxy environment variables.

- The client and server certificates were transferred to the proxy, and mitmproxy was configured to utilize them.

With this setup, all traffic passing between the client and server can be decrypted and inspected at the proxy.

Use Case

By implementing a transparent SSL proxy using mitmproxy, a mechanism was established for intercepting and analyzing encrypted traffic between the client and server. This setup facilitates monitoring and analyzing communication between the two endpoints, enabling insights into data exchanges and potential security vulnerabilities.