Info2222 Security Assignment Report

530317166, 510460215

April 26, 2024

Contribution Summary

0.1 530317166

Student Ian/530317166 did the following

- User login
- Friends feature
- Chatroom's asymmetric key generation encryption and decryption.
- Message History
- Hash and Salt
- Authentication

0.2 510460215

Student Scott/510460215 did the following

- Chatroom's Hmac
- HTTPS
- tba

1 User Login Security

Contributed by Ian

1.1 Preventing XSS Attacks

To prevent XSS attacks, we ensure that all user inputs are sanitized before rendering them on any page. Flask's Jinja templates automatically escape all variable content rendered into HTML, which is crucial for preventing XSS. Below is a code snippet from our login form processing:

```python
@app.route("/login/user", methods=["POST"])
def login_user():
    if not request.is_json:
        abort(404)

    username = request.json.get("username")
    password = request.json.get("password")

    user = db.get_user(username)
    if user is None:
        return "Error: User does not exist!"

    if not check_password_hash(user.password, password):
        return "Error: Password does not match!"

    access_token = create_access_token(identity=username)
    response = jsonify({'login': True})
    set_access_cookies(response, access_token)
    return response
```

This function not only checks the validity of the user's credentials but also prevents XSS by not directly embedding user input in HTML responses.

1.2 Encryption and Secure Password Handling

Passwords are hashed first on the client side before sending and on server side using Werkzeug's security tools, ensuring that even if database access is compromised, the passwords remain secure. Below is how we handle password hashing on user registration:

```python
def insert_user(username: str, password: str, public_key: str):
    hashed_password = generate_password_hash(password)
    with Session(engine) as session:
        user = User(username=username, password=hashed_password,
    public_key=public_key)
        session.add(user)
        session.commit()
```

1.3 JWT-Based Session Management

We use JSON Web Tokens (JWT) for managing sessions. This method ensures that user sessions are stateless and securely validated on each request. Here is how we generate and validate tokens:

```python
# Create a token
access_token = create_access_token(identity=username)
response = jsonify({'login': True})
```

```
set_access_cookies(response, access_token)  # Set the JWT in a
    cookie

@app.route("/home")
@jwt_required()
def home():
    current_user = get_jwt_identity()
    # Proceed with handling the request
```

This approach ensures that each request to the server must come from an authenticated session, significantly enhancing the security of our application.

2 Friends list

Contributed by Ian

The friends list in the application is dynamically rendered using Flask and Jinja2 templates, ensuring real-time updates and security against web vulnerabilities such as XSS attacks. Below is an explanation of the mechanisms put in place to ensure the safety and integrity of this feature.

2.1 Dynamic Content Loading

```
# Flask route that handles fetching friends
@app.route("/list-friends/<username>")
def list_friends(username):
    friends = db.list_friends(username)
    return render_template("friends_list.jinja", friends=friends,
    username=username)
```

Listing 1: Dynamically Loading Friends List

2.2 Security Measures

2.2.1 XSS Prevention

All dynamic content rendered through Jinja2 templates automatically escapes any HTML tags unless explicitly marked otherwise. This behavior prevents the injection of malicious scripts, thereby safeguarding against XSS attacks.

```
<ul>
    {% for friend in friends %}
    <li>{{ friend|escape }}</li>
    {% endfor %}
</ul>
```

Listing 2: Auto-escaping in Jinja2

2.2.2 Secure Session Management

User sessions are managed using JSON Web Tokens (JWTs), which are securely stored in HTTPOnly cookies. This method prevents client-side scripts from accessing the token, reducing the risk of XSS and CSRF attacks.

```python
# Set JWT in HTTPOnly cookies
set_access_cookies(response, access_token)
```

Listing 3: JWT Session Management

2.2.3 Friendship Verification

The application verifies that friendship exists before allowing any interaction. This server-side check ensures that users can only interact with their actual friends, preventing unauthorized access.

```python
def are_friends(user1, user2):
    return db.are_friends(user1, user2)
```

Listing 4: Friendship Verification

3 Users can add friends by submitting another user's username to the server

Contributed by Ian

The functionality for users to add friends is facilitated through specific routes in the Flask application. The following Python code snippet shows the server-side handling of sending friend requests:

```python
# Route to send a friend request
@app.route("/add-friend", methods=["POST"])
def add_friend():
    if not request.is_json:
        abort(400, 'Requests must be JSON formatted.')  # Ensures
    that the request is in JSON format

    sender = request.json.get("sender")
    receiver = request.json.get("receiver")

    # Checks if both users exist
    if db.get_user(sender) is None or db.get_user(receiver) is None
    :
        return jsonify({"msg": "One or both users not found"}), 404

    db.send_friend_request(sender, receiver)
    return jsonify({"msg": "Friend request sent successfully!"}),
    200
```

Listing 5: Handling Friend Requests

This route performs several checks: - Validates that the request data format is JSON. - Ensures that both the sender and receiver are existing users in the database. - Adds a friend request to the database through a secure interface.

Security Considerations

The application incorporates various security measures to protect against common vulnerabilities:

- **Input Validation:** All input data from the user is validated to ensure it meets the expected format and type, preventing SQL Injection and other forms of input-based attacks.
- **User Authentication:** Users must be authenticated to send friend requests, ensuring that actions are performed by legitimate users.
- **XSS Prevention:** By enforcing JSON formatted data and utilizing server-side rendering of user-generated content with appropriate escaping, the application is safeguarded against Cross-Site Scripting (XSS) attacks.

4

Displaying Friend Requests *Contributed by Ian*

```
{% for request in received_requests %}
<li>{{ request.sender }} -
    <button onclick="acceptFriendRequest({{ request.id }})">Accept</button>
    <button onclick="rejectFriendRequest({{ request.id }})">Reject</button>
</li>
{% endfor %}
```

Listing 6: Displaying Friend Requests in Jinja2 Template

This segment ensures that each friend request is displayed with options to either accept or reject, directly reflecting changes made by the user in a secure and intuitive interface.

5 Secure Chat Room Functionality

Establishing a Secure Chat Room

Contributed by Ian

Users can initiate a chat by clicking on a friend's name, which triggers a request to join a chat room if they are online. The following is a high-level overview of the process implemented in the Flask application:

```
# Function to start a chat with a friend
function startChatWith(friendUsername) {
    $("#receiver").val(friendUsername);  // Sets the friend's username in the receiver input
    join_room(friendUsername);  // Triggers the join room function
}

# Function to join a chat room
```

```
function join_room(friendUsername) {
    let receiver = $("#receiver").val().trim();
    if (!isFriend(receiver)) {
        alert("You can only chat with friends.");
        return;
    }
    socket.emit("join", username, receiver, (res) => {
        if (typeof res != "number") {
            alert(res);  // Error handling
            return;
        }
        room_id = res;  // Room ID is set here
        $("#chat_box").hide();
        $("#input_box").show();
    });
}
```

Listing 7: Joining a Chat Room

This code ensures that users can only start conversations with friends by validating the friendship before initiating the chat. This validation prevents unauthorized access to chat rooms.

Secure Message Exchange

Contributed by Ian

Messages are encrypted client-side using private-key before being sent over the network. The server, acting as a middleman, cannot decipher the content of these messages due to the encryption:

```


# Function to send an encrypted message
async function send() {
    const receiver = $("#receiver").val();
    const message = $("#message").val();
    $("#message").val("");  // Clear the input after sending

    const publicKey = await fetchPublicKey(receiver);
    if (!publicKey) {
        alert('Could not fetch public key for encryption');
        return;
    }

    const encryptedMessage = await encryptMessage(publicKey,
    message);
    socket.emit("send", username, encryptedMessage, room_id);
    add_message(`You: ${message}`, "grey");  // Display the message
     on the sender's side
}
```

Listing 8: Encrypting and Sending Messages

```

    async function decryptMessage(encryptedMessage) {
        console.log("Decrypting message: ", encryptedMessage);
        if (!isBase64(encryptedMessage)) {
            console.error('Decryption failed: Invalid Base64 encoding');
            throw new Error('Invalid encrypted message data');
        }
        try {
            const privateKeyBase64 = await loadPrivateKey(username);
            if (!isBase64(privateKeyBase64)) {
                console.error('Decryption failed: Invalid private key data');
                throw new Error('Invalid private key data');
            }
            console.log("Private key loaded:", privateKeyBase64);
            const privateKeyBuffer = new Uint8Array(atob(privateKeyBase64).split('').map(char => char.charCodeAt(0)));
            const privateKey = await window.crypto.subtle.importKey(
                "pkcs8",
                privateKeyBuffer,
                {name: "RSA-OAEP", hash: {name: "SHA-256"}},
                true,
                ["decrypt"]
            );

            const decodedMessage = window.atob(encryptedMessage);
            console.log("Decoded Base64 message:", decodedMessage);
            const encryptedBuffer = new Uint8Array(decodedMessage.split('').map(char => char.charCodeAt(0)));

            const decrypted = await window.crypto.subtle.decrypt(
                {name: "RSA-OAEP"},
                privateKey,
                encryptedBuffer
            );
            const decoder = new TextDecoder();
            const decodedText = decoder.decode(decrypted);
            console.log("Decrypted text:", decodedText);
            return decodedText;
        } catch (error) {
            console.error("Decryption process error:", error);
            throw new Error('Decryption failed');
        }
    }
```

Listing 9: Decrypting Messages

The encrypted messages are only decipherable by the recipient, who possesses the corresponding private key. This mechanism ensures that sensitive information remains confidential even if intercepted during transmission.

Message Display and Authentication

Contributed by Ian

The client-side application also handles the display of incoming messages, ensuring that messages are correctly attributed to their senders:

```
# Modify the existing incoming message handler to decrypt messages
socket.on("incoming", async (data, color = "black") => {
    if (data.username !== username) {  // Check if the message is
    from another user
        const decryptedMessage = await decryptMessage(data.message)
    ;
        add_message(`${data.username}: ${decryptedMessage}`, color)
    ;
    }
});
```

Listing 10: Receiving and Decrypting Messages

HMAC Implementation

Contributed by Scott

The system uses HMAC authentication to ensure that messages in transit are not modified for malicious purposes, providing data integrity and authentication from the system. With aid from the *SubtleCrypto API*, we can setup the mechanisms that employ HMAC.

Generating a Signature

Whenever the server is initialised, it creates a new shared secret. The shared secret is a key which has the ability to sign and verify messages. It is encoded in a SHA-256 hash with the ability to be exported. It is exported into the raw format, stored as a base64 string and placed within the localstorage. If there are any errors, catch them.

```
    async function generateHMAC() {
        try {
            let Hkey = await window.crypto.subtle.generateKey(
            {
                name: "HMAC",
                hash: { name: "SHA-256" },
            },
                true,
                ["sign", "verify"],
            );
            const exportHkey = await window.crypto.subtle.exportKey
    ("raw", Hkey)
            const exportHkeyBase64 = btoa(String.fromCharCode.apply
    (null, [...new Uint8Array(exportHkey)]));
            localStorage.setItem("Hkey", exportHkeyBase64);

        } catch (error) {
            console.error("HMAC Key Generation Error: ", error);
            return null;
        }
    }
    generateHMAC();
```

Whenever a message is sent, we will create a signature along with that message which will be sent to the receiver. We firstly import our key and convert it to a bufferArray from base64. We give it the ability to sign messages. Using crypto.subtle.sign, we then apply a signature onto the message, encode it as a base64 string and one again, place it into localstorage. Return null if there are any errors.

```
    async function sign(message) {
        try {
            let encMessage = new TextEncoder().encode(message);

            let key = localStorage.getItem("Hkey");

            const importHkey = await crypto.subtle.importKey(
                'raw',
                b642ab(key),
                {name: 'HMAC',
                hash: {
                    name: 'SHA-256'
                }},
                true,
                ['sign']
            );

            const signature = await window.crypto.subtle.sign(
                "HMAC",
                importHkey,
                encMessage
            );
            const signatureBase64 = btoa(String.fromCharCode.apply(
    null, [...new Uint8Array(signature)]));
            localStorage.setItem("Signature", signatureBase64);
        } catch (error) {
            console.error("HMAC generation error: ", error);
            return null;
        }
    }
```

Lastly, whenever the receiver gets the message, we verify the message by once again, importing our shared secret with the ability to verify messages, and applying crypto.subtle.verify along with the messages signature to return a promise on whether the message has been tampered with. We will return a boolean on the according result.

```
async function verify(message) {
    try {
        let encMessage = new TextEncoder().encode(message);

        let sig = localStorage.getItem("Signature");
        let key = localStorage.getItem("Hkey");

        const importHkey = await crypto.subtle.importKey(
            'raw',
            b642ab(key),
            {name: 'HMAC',
            hash: {
                name: 'SHA-256'
            }},
            true,
                ['verify']
            );
            let result = await window.crypto.subtle.verify(
                "HMAC",
                importHkey,
                b642ab(sig),
                encMessage
            );
            if(result) {
                return true;
            } else {
                console.log("Verified False!")
                return false;
            }
    } catch (error) {
            console.error("Verification Error: ", error);
            return null;
    }
};
```

6 Hash and Salt

Contributed by Ian

The system employs a hashing mechanism where the password provided by the user is transformed using a cryptographic hash function. This hash function is designed to be one-way, meaning that it is computationally infeasible to reverse the hash value to retrieve the original password.

Password Hashing

The password is hashed using the SHA-256 hashing algorithm. This algorithm converts the password into a fixed-size string of characters, which is stored in the database instead of the actual password.

```
let hashedPassword = CryptoJS.SHA256(password).toString();
```

Salting

A unique salt is generated for each password. This salt is a random string added to the password before it is hashed. The purpose of the salt is to prevent attackers from using precomputed hash tables to crack the password.

```
const salt = window.crypto.getRandomValues(new Uint8Array(16));
const saltBase64 = btoa(String.fromCharCode(...salt));
```

Storing Hash and Salt

Both the hash and the salt are stored in the database. The hash is used to verify the user's password at login, and the salt is required to hash the user password in the same way for future verifications.

```
// Store hashed password and salt in database
db.insert_user(username, hashedPassword, saltBase64,
    publicKeyBase64);
```

Server-side Implementation

On the server side, the hashed password and salt are handled securely to prevent any leaks or unauthorized access.

```
def insert_user(username: str, password: str, salt: str, public_key
    : str):
    hashed_password = generate_password_hash(password + salt)
    with Session(engine) as session:
        user = User(username=username, password=hashed_password,
    public_key=public_key)
        session.add(user)
        session.commit()
```

7 HTTPS

Contributed by Scott

HTTPS is a more secure version of HTTP, allowing for the use of SSL/TLS protocols in web applications. While it may be trivial to generate a private key as well as the root certificate using openssl, it is quite difficult to ensure you have a trusted root certificates list. To ease such a process, I employed the use of the git repository, *mkcert.org*. The repository allows for a more streamlined, ease of use API to generate a custom PEM file.

```
mkcert example.com "*.example.com" example.test localhost 127.0.0.1
    ::1
```

Once run, we are given two files:

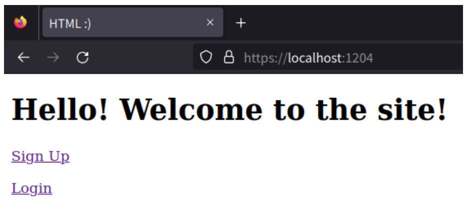
```
example.com+5-key.com
example.com+5.pem
```

With this, our local CA is currently installed in the Firefox and/or Chromium trust store. However, there are still a couple of steps to take in order to fully validate our local CA certificates.

In most linux distros (in this case, ubuntu) we must install our root CA certificate into the trust store. This is simply done by copying our newly generated PEM file into the root CA certificate directory in our distro. After copying our file, we update our CA certificates accordingly.

```
sudo cp example.com+5.pem /usr/local/share/ca-certificates/example.
    com+5.crt
sudo update-ca-certificates
```

Lastly, we employ the use of Nginx to complete our ssl installation by editing our virtual host file. In doing so we, firstly define our new server name (in this cae, example.com). specify the location of the new server. Considering we are



Hello! Welcome to the site!

Sign Up

Login

Figure 1: Working HTTPS Connection

running the program locally, we set it to localhost on port 1204. To configure the ssl parameters, we define the port we will be listening on. In this case, 443 as it is the standard port for secured connections. Lastly, we define our newly created PEM and PEM-key file by giving the server the path to each file respectively.

```
server {
    server_name example.com;
    location / {
        proxy_pass http://127.0.0.1:1204/;
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_set_header X-Forwarded-Port $server_port;
        proxy_set_header Host              $host;
        proxy_set_header X-Forwarded-For "IP"
        expires 1M;
        access_log off;
        add_header Cache-Control "public";
    }
    listen [::]:443 ssl http2;
    listen 443 ssl http2;
    ssl_certificate /path/to/example.com+5.pem
    ssl_certificate_key /path/to/example.com+5-key.com
}
```

Lastly, we simply add our certfile and keyfile to our socketio.run function on app start.

```
if __name__ == '__main__':
    socketio.run(app, host = 'localhost', port = 1204,
                 keyfile = 'example.com+5-key.pem',
                 certfile = 'example.com+5.pem')
```

And with this, we have now set up a HTTPS connection to our locally hosted server.

8 Authentication

Contributed by Ian

JWT Authentication

JSON Web Tokens (JWT) are used to manage secure routes on our server. This token-based authentication strategy ensures that once a user is logged in, they receive a token that must be included in the headers of all subsequent requests requiring authentication.

```python
# Configure JWT
app.config['JWT_SECRET_KEY'] = 'your_jwt_secret_key'
jwt = JWTManager(app)

@app.route("/home")
@jwt_required()
def home():
    current_user = get_jwt_identity()  # Get the identity of the
    current user from JWT
    return render_template("home.jinja", username=current_user)
```

Login and Token Issuance

Upon login, a check is performed to ensure the user's credentials are correct. A JWT is then generated and sent to the user's browser as a cookie, which must be presented on subsequent requests.

```python
@app.route("/login/user", methods=["POST"])
def login_user():
    username = request.json.get("username")
    password = request.json.get("password")
    user = db.get_user(username)
    if not check_password_hash(user.password, password):
        return jsonify({"login": False, "msg": "Password does not
    match!"}), 401
    access_token = create_access_token(identity=username)
    response = jsonify({'login': True, "msg": "Login successful"})
    set_access_cookies(response, access_token)
    return response
```

Securing Routes

All critical routes are secured using the `@jwt_required()` decorator, which ensures that no unauthenticated requests can access these endpoints. This protection is extended to any action that modifies data, retrieves sensitive information, or could potentially expose user data.

```python
@app.route("/add-friend", methods=["POST"])
@jwt_required()
def add_friend():
    sender = get_jwt_identity()
    receiver = request.json.get("receiver")
    db.send_friend_request(sender, receiver)
    return "Friend request sent successfully!", 200
```

Authentication Tests

Specific tests were conducted focusing on authentication:

1. **Credential Testing:** Automated scripts attempted to use common passwords and previously breached credentials to access the system.

2. **Token Manipulation:** JWT tokens were modified to test the integrity checks and session management.

3. **Route Authorization:** Automated tools and manual testing were used to access protected routes without proper authentication.

```python
# Example of an automated test for route authorization
import requests

def test_unauthenticated_access():
    urls = [
        "http://example.com/api/protected",
        "http://example.com/api/admin",
        "http://example.com/api/delete-user"
    ]
    for url in urls:
        response = requests.get(url)
        assert response.status_code == 401, f"Unauthorized access allowed for {url}"

test_unauthenticated_access()
```