Group 6:
Cybersecurity Report into ScreenSort: A Movie Recommendation API

CONTENTS

1. Understanding Common Threats

Authored by Stella Dixon

The Film Recommendation API, ScreenSort, is vulnerable to a variety of security threats that could potentially compromise user data and the applications overall security. Drawing from the project files, some of the key threats to this application include **SQL Injection (SQLi), Cross-Site Scripting (XSS), Broken Authentication** and **Data Exposure**. In this section of the report we will cover specific examples of how these threats apply to ScreenSort and their potential impact.

SQL Injection (SQLi)

SQL Injection is a severe attack that targets the database through maliciously crafted input. This attack could occur specifically in the API's **/login** functionality and any other area where user input is directly incorporated into the database queries.

Impact:

- **Unauthorised Access**: Attackers could login without valid credentials, gaining access to user accounts and administrative accounts leading to the exposure of sensitive data.

- **Data Manipulation**: Attackers could alter film recommendations, user preferences, or inject content which could affect the functionality of the system.

- **Data Loss or Corruption**: Malicious SQL commands could be used to delete or corrupt database records, resulting in a significant loss of data and service disruption.

- **Privacy Violations**: Attackers could exploit these vulnerabilities to access other users preferences/personal information leading to privacy issues and a breach of user application trust.

To mitigate this we recommend using parameterized queries. The Film Recommendation API already uses SQLAlchemy which provides built in protection against SQL injection by automatically parameterized queries, however directly executing raw SQL queries can bypass SQLAlchemys protections potentially introducing vulnerabilities. We recommend making sure all database injections use SQLAlchemy's ORM methods, which abstract raw SQL and safely handle user input.

Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS) is when an attacker injects malicious scripts into a webpage that runs in another user's browser. This could happen in the search functionality or review forms, where user input is reflected back to the webpage.

Impact:

- **User Data Theft**: Attackers could steal session cookies, hijack user accounts and access personal information.

- **Session Hijacking**: Compromised sessions could allow attackers to impersonate legitimate users and access sensitive data or features.

- **Malware Disruption**: Injected scripts could redirect users to malicious sites, exposing them to further attacks.

In order to prevent XSS the system needs to validate and sanitise all user inputs before rendering them on the webpage. Using Flasks built in functions and adding Content Security Policy (CSP) will help block malicious scripts from being executed.

Broken Authentication

Broken authentication occurs when the apps login system isn't set up securely, allowing attackers to break into users accounts. In the Film Recommendation API this could occur at the **/login** feature where user sessions are managed. As the API does not properly secure session cookies or limit the number of login attempts it is vulnerable to brute force attacks or session hijacking.

Impact:

- **Service Disruption**: Repeated login attempts or hijacked accounts could disrupt the service for legitimate users, creating a poor user experience, and a lack of trust in the reliability/functioning of the service.

- **Unauthorised Access**

- **Session Hijacking**

- **Privacy Violations**

To help mitigate broken authentication risks, the API should enforce secure session handling with **secure** and **HttpOnly** in order to prevent brute force attacks from cookies and repeated login attempts. Adding Multi-Factor Authentication (MFA) would also provide additional security.

Data Exposure and Broke Access Control

Data Exposure occurs when an application does not properly restrict access to sensitive information. The **/thumbs_action** endpoint within the API allows users to modify their film

preferences. However, if the endpoint does not have the correct authentication checks an attacker could manipulate the API request to view or alter another user's data.

Impact:

- **Data Leakage**: Exposure of sensitive information could lead to broken access controls and a misuse of information, potential identity theft and legal consequences for failing to protect users privacy.

- **Privacy Violations**

- **Data Manipulation**

The system could help prevent data exposure by implementing Role Based Access Control (RBAC) and checking user permissions before granting access to sensitive endpoints. Regularly auditing the access control policies would also help ensure that the right data is only accessible to the correct authorised users.

In conclusion the Film Recommendation API faces several security threats including SQL Injection (SQLi), Cross-Site Scripting (XSS), Broken Authentication and Data Exposure. In order to protect user data, maintain the integrity of the application, and ensure a safe user experience the application should utilise input validation, secure session management and proper access controls. This would ensure both user privacy, data as well as the applications overall functionality is secure.

---

2. Secure Coding Practices

Authored by Amy Crossan

Security centric guidelines for flask applications

The security of any Flask application is of the utmost importance, the Flask framework uses third-party modules which can increase the risk of security breaches occurring. If strong security measures are not kept in mind while creating and managing a web application that makes use of Flask the application can be vulnerable to a variety of exploits; notably SQL injection, Cross-Site Request Forgery (CSRF) and cross-site scripting (XSS) attacks.

While creating a Flask based application there are several key security aspects that should be kept in mind.

As a general rule while creating the code for your flask application you want to ensure you are keeping the security of your application in mind. For example if you are going to be taking in user details you must ensure that any important information they may input such as passwords, addresses or emails are securely stored with sensitive information being hashed. There are several specific practices that can be undertaken while creating

Adding general Security features

Firstly making use of a resource such as Flask-Security can greatly aid you in securing your application. Flask-Security is a resource which can be integrated with Flask applications to help secure them. Flask-Security allows for the quick and easy addition of common security mechanisms to your application. Some of these common security mechanisms it can help add include:

- Authentication
- Role and Permission management
- Login tracking and user registration

It is able to aid in adding these features through the integration of several flask extensions and libraries like:

- Flask-wtf
- Flask-login
- authlib

though there are more. Flask-Security does assume that you are using a common library for your application's database and it supports various common flask extensions that you may use for this such as SQLAlchemy.

If your application is going to be taking in inputs from the user you must treat any/all user input as both potentially malicious and as sensitive data you need to ensure it is stored securely.

Hashing

Hashing is a security measure where a string of characters or a key are changed into different values, it is used for one way encryption and values that have been hashed are very hard to decode. Hashing can be used to protect data and prevent it from being revealed or tampered with. The flask extension Flask-Hashing is one method by which you can hash data and check the hash of a value within a flask application, it utilises hashlib for the actual hashing of the data. Through this extension you can add another layer of security for your application, protecting sensitive data from tampering. The previously mentioned Flask-Security can also provide your application with hashing

Protecting against malicious attacks

CSRF attacks

It is also of the utmost importance to ensure your application's forms have precautions in place to deal with Cross-Site Request Forgery (CSRF) attacks. The main aim for attackers using this method is to find a way to perform an action on behalf of an authenticated user of a website/application without their knowledge.

Making use of wtforms in flask for your forms can help protect against CSRF attacks as it has solutions to this type of vulnerability built in. Through wtforms you can make use of its utilities for server side validation like secure_filename as aforementioned user input should not be implicitly trusted so making use of the utilities of wtforms to aid in sanitization of user input just in case of malicious intent can really aid the security of your application.

Another method you could use to protect against this type of attack is the generation and validation of CSRF tokens. A CSRF token is a random and unique value that can be checked when the form is submitted to see if the CSRF token is both valid and present. If you set up your application so every form generates a CSRF token you can add in a layer of security by checking every submitted form for a valid CSRF token and reject the submission if the CSRF token is missing or invalid.

XSS attacks

Cross-site scripting (XSS) is a significant threat facing web applications, this is a type of attack where the attacker attempts to insert malicious code into a web application. This is usually carried out through user input sections on web applications that do not have proper validation, meaning the security of user inputs is extremely important for the security of your application as a whole.

Protecting your application effectively from this type of attack involves various tools and practices to sanitise user inputs and properly manage data rendering within the application. These tools and practices include:

- Ensuring that user inputs are always escaped before being rendered
- The use of a Content security policy (A browser feature that can help with identification and mitigation of attacks)
- Avoid using inline Javascript, especially in sections of code where user data is involved
- Keeping Flask dependencies updated with the latest security updates

Sources

Cfg Cyber Security masters + slides and lecture recordings
https://owasp.org/API-Security/editions/2023/en/0x11-t10/
https://escape.tech/blog/best-practices-protect-flask-applications/

https://flask-security-too.readthedocs.io/en/stable/
https://snyk.io/blog/secure-python-flask-applications/
https://escape.tech/blog/csrf-vs-xss/
https://escape.tech/blog/cross-site-scripting-xss-in-graphql/
https://flask-hashing.readthedocs.io/en/latest/
https://builtin.com/articles/what-is-hashing
https://www.geeksforgeeks.org/password-hashing-with-bcrypt-in-flask/
https://www.sqlalchemy.org/
https://owasp.org/www-community/attacks/csrf
https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html
https://www.sqlshack.com/using-parameterized-queries-to-avoid-sql-injection/
https://techcommunity.microsoft.com/t5/sql-server-blog/how-and-why-to-use-parameterized-queries/ba-p/
https://www.stackhawk.com/blog/finding-and-fixing-sql-injection-vulnerabilities-in-flask-python/

The importance of parameterized queries

SQLInjections are a very common threat to flask applications and an effective and easy way to protect your application from this threat is through parameterized queries.

The aim of parameterized queries is to ensure that the SQL command your application runs is kept separate from the values inputted by the user. It does this by making it so any user input is passed in a query as a parameter. This means user input is treated as data, preventing attackers from being able to ‘inject’ any malicious code through user input.

Securing coding practices for ScreenSort

ScreenSort as a film recommendation application that also takes in user data in the form of user profiles and watch history will require effective secure coding practices to ensure the safety and integrity of both user data and the application itself.

For securing this application I would ensure the guidelines I have outlined above are adhered to within its code with particular focus on parameterized queries, hashing and protection against malicious attacks. I would focus on these areas as due to the application taking in user input I would want to ensure that the application itself was protected incase an attacker wanted to try exploit a user input vulnerability and any data added by users for their profile, particularly passwords, was safely hashed so it couldn't be tampered with.

---

3. HTTPS Implementation

Authored by Gabriella Booth

The implementation of HTTPS in any web application is of the utmost importance as it works to improve on HTTP by meeting two of the criteria of the CIA triad for a secure connection. HTTPS authenticates the connection between the user and server and ensures confidentiality and integrity in the transfer of data between the two.

At this time, ScreenSort does not have a domain and runs on an HTTP address of "http://127.0.0.1:5000". Due to this, it does not take advantage of the security that an HTTPS connection provides such as:

- Encryption
- Search Engine Optimisation (SEO)
- Data Integrity
- Authentication

In addition, it opens itself up to attacks from malicious actors who can exploit the HTTP connection and its vulnerabilities. For example, users make an account on ScreenSort to provide personalised recommendations and, though encryption methods are used in the program, using HTTP may cause the interactions between the user and server to be intercepted and their login details accessed.

With ScreenSort making use of a user account creation and log-in system, the encryption that HTTPS can provide would be very beneficial to the program. This way it would build on the encryption that is already present within the program by HTTPS making use of encryption algorithms through protocols such as SSL, or its upgraded version TLS, so that any data being transferred is not in plain text and stops any malicious actor from reading the data if it is intercepted.

Using HTTPS also provides data integrity by minimising the chances of data being modified if it is intercepted. This is due to hashing being used to detect if any changes have occurred and terminating the connection if so.

As ScreenSort is not on HTTPS and does not have the added benefit of encryption, using Burp Suite you can intercept packages from the log-in page which display the username and password in plain text rather than an encryption. Not only could a malicious actor then use these login details to access the system but they could also modify these to or use methods such as SQL injection to modify the login and gain access as another user - such as an admin account.

```
POST /login HTTP/1.1
Host: 127.0.0.1:5000
Content-Length: 167
Cache-Control: max-age=0
sec-ch-ua: "Chromium";v="129", "Not=A?Brand";v="8"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Windows"
Accept-Language: en-GB,en;q=0.9
Origin: http://127.0.0.1:5000
Content-Type: application/x-www-form-urlencoded
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/129.0.6668.71 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: http://127.0.0.1:5000/login
Accept-Encoding: gzip, deflate, br
Cookie: session=
.eJwljklqBDEMAP_icw5abEmezzSWJZEQSKB75hTy9zTkWAUF9d000vN6b4_n-cq3dnxEezQHRPQgTUfBneWInSB8MnXlvsm3dKldLFjqwwlqDjPWQeW8u0SnHJghps6-ZpEHg6BFCJAGWU7h2FAD192RwawuEu7b2z3yuvL8v8Eb93XW8fz-zK9b80RSmsm5zFeCA9vG2WeMGsoeabhYsf3-AfHEP5o.ZwWi6w.h4MgfTHuNZtmH_qLo7WtNkUJ8Vg
Connection: keep-alive

csrf_token=IjM5M2Y3Mjl1M2Vh0GJhZTBiMDM4YzE5NDlkNWY1NzNiZGU4MWEzNzEi.ZwWi2A.b_rzklU2wkJn4cpq77_uVwLQ-cE&email=johndoe%40example.com&password=Password123%21&submit=Login
```

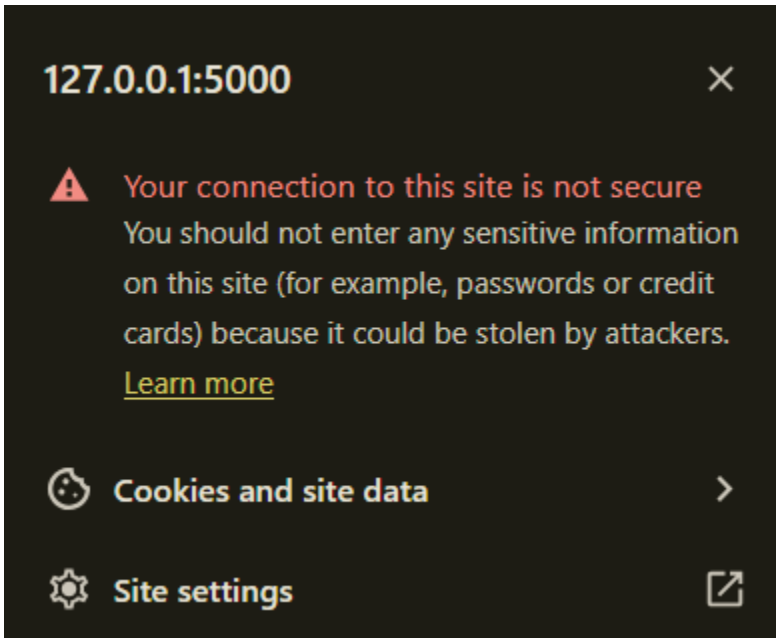

```
csrf_token=
IjM5M2Y3Mjl1M2Vh0GJhZTBiMDM4YzE5NDlkNWY1NzNiZGU4MWEzNzEi.ZwWi2A.b_rzklU2wkJn4cpq77_uVwLQ-cE&
email=johndoe%40example.com&password=Password123%21&submit=Login
```


SSL Certificates

Alongside HTTPS providing encryption in data transfers between the user and server and data integrity, it also provides authentication through the use of SSL certificates.

Any web page with a secure, HTTPS connection will have a digital certificate (an SSL certificate) to confirm that the web page is the one it is claiming to be. This again works to meet the criteria of ensuring confidentiality, integrity and authenticity of data so that a user can feel confident in using the web page and trusting their data is secure.

If a website does not have an SSL certificate (such as ScreenSort), there is a warning from the user's browser to advise it is not a secure connection and data could be stolen. This is demonstrated in the screenshot below when loading up ScreenSort.



The HTTPS connection is secured through a process called the “SSL handshake” which involves the following steps:

- A secure session request is sent from the user’s system to the server
- The server responds with the SSL certificate and the server’s public key
- The user’s system authenticates the certificate
- The user’s system creates a random symmetric key and encrypts it using the public key from the server
- The user system and server use the symmetric key to encrypt the user data throughout the session

These handshake steps making use of the SSL certificate ensure secure communication by using the certificate to authenticate the validity of the website and only allowing a connection to be made if both the user’s system and server trust each other. In addition, the limited validity period of the certificate provides further security as there is a short window for malicious actors to exploit vulnerabilities and compromise the certificate.

The SSL certificate also helps to prevent Man-In-The-Middle attacks due to the authentication checks that happen that will prevent a connection if the webpage does not match up with the certificate provided or if it is detected that data has been manipulated when transferred.

It would not be possible to obtain an SSL certificate for ScreenSort at this time due to it running on HTTP and not having a domain name. However, if it got to the point of going live with a full domain name, the following steps would need to be taken to obtain a certificate and install this.

- Create a Certificate Signing Request (CSR)
    - This is generated on the server to include all of the information on the domain
    - The Certificate Authority (CA) will use this information when verifying the web page
- Submit the CSR to the CA
    - Generally this is reviewed quite quickly but could take a few days depending on the type of certificate and details provided
    - If the CA confirm that the website is valid and a legitimate page, the SSl certificate will be issued
- Install the SSL certificate to the whole website
    - How to install the SSL certificate is dependent on the web server
    - With ScreenSort using Flask this could be installed by downloading the certificate and using the code below

```
context = ("certificate absolute path", "key absolute path")
app.run(host="0.0.0.0", port=443, ssl_context=context)
```

Once the certificate is installed, it would be important to regularly update the SSL certificate as well so it does not go out of its validity period and it continues to show as a legitimate web page.

Enforcing HTTPS in a Flask Application

Though the SSL certificate would display that ScreenSort is a legitimate web page, enforcing HTTPS for the application is still important to ensure any visitors the website are coming through on a secure connection.

One way of doing this would be to have the ScreenSort program manually redirect any traffic to HTTPS no matter where the traffic originates from. This could be done by using the decorator "*@app.before_request*" as shown in the code below.

```
@app.before_request
def before_request():
    if not request.is_secure:
        url = request.url.replace( _old: "http://",  _new: "https://",  _count: 1)
        code = 301
        return redirect(url, code-code)
```

---

4. Authentication and Authorisation

Authored by Jemma Lowman

Authentication and authorisation are two critical components of securing any web application, especially one like ScreenSort**,** that handles user data and personalised preferences.

Authentication

Authentication refers to the process of verifying a user's identity. It ensures that users are who they claim to be by requiring them to provide valid credentials, such as a username and password. This step is crucial to prevent unauthorised access to the system.

Authorisation

Authorisation occurs after authentication and involves determining which resources or actions a user is allowed to access. It ensures that once users are authenticated, they are restricted to accessing only the data and functionalities they are permitted to. For instance, users might be able to see general movie recommendations but not administrative features like adding new films.

In this system, authentication confirms user identity, while authorisation ensures that users can only interact with data according to their role and privileges.

Principles of Strong User Authentication

To ensure secure authentication, several principles are followed:

- ***Password Strength and Security***
    - **Strong password policies**: Users should be required to set strong passwords that include a mix of uppercase and lowercase letters, numbers, and special characters. Passwords should meet a minimum length requirement, and users should be encouraged to change them periodically.
    - **Hashing and salting**: Storing plaintext passwords is a significant security risk. Instead, passwords should be hashed using algorithms like SHA-256 or bcrypt, combined with salting, to protect against password reversals, even in the event of a database breach.
- ***Two-Factor Authentication (2FA)***
    - Implementing **two-factor authentication (2FA)** adds an extra layer of security by requiring a second factor, such as a one-time passcode sent via email or SMS, in addition to the user's password.
- ***Secure Session Management***
    - **Session Tokens**: After successful authentication, the system should generate a secure session token (such as a JWT or session ID) that the user's browser can store in cookies. This token is then required for every subsequent request the user makes.
    - **Token expiration and renewal**: To mitigate session hijacking risks, session tokens should expire after a defined period of inactivity. Users should be required to re-authenticate when the session expires.
- ***Preventing Brute-Force Attacks***
    - **Login throttling and CAPTCHA**: Limit the number of failed login attempts by temporarily locking the account after several failed tries. CAPTCHA can also be added to prevent automated brute-force attacks.

Plan for Proper Authorisation and Access Control

After users are authenticated, proper authorisation ensures they can only access functionalities aligned with their role.

- ***Role-Based Access Control (RBAC)***
    - **Role-Based Access Control (RBAC)** is a widely adopted authorisation method that assigns permissions based on the user's role. In a Movie Recommendation API, typical roles might include:
        - **Admin**: Full access, including the ability to add or remove films, manage users, and view all recommendations.
        - **Registered User**: Can view, rate, and receive personalised movie recommendations.

        - **Guest User**: Limited access to general movie recommendations without any personalised features or data modification privileges.

By implementing RBAC, the system can ensure that only users with the appropriate role can access certain endpoints. For example:

- /admin/add_movie: Accessible only by Admin users.
- /recommendation: Accessible by registered users for personalised recommendations.
- /movies: Open to guest users for general recommendations.

Principle of Least Privilege (PoLP)

The **principle of least privilege** states that users should only have the minimum access rights necessary to perform their functions. Admins, for example, will have broader access rights than regular users, but even they should only be granted permissions essential to their role.

Fine-Grained Permissions

Beyond role-based access, **fine-grained permissions** allow for more granular control over what actions users can perform. For example, while a user might be able to rate a movie, they may not have permission to add new films. This adds flexibility to access control.

Audit Logging and Monitoring

To ensure transparency and accountability, sensitive actions (such as deleting or modifying data) should be logged. This includes recording who performed the action, what was affected, and when it occurred.

Role of Flask-Login in Managing User Sessions

Flask-Login is a lightweight Flask extension that simplifies managing user sessions within web applications. It plays an essential role in session management for the Movie Recommendation API, supporting user authentication, session expiry, and login state tracking – this function is already included in the initial code of the project.

- ***Flask-Login Features***
    - **Session Management**: Flask-Login keeps track of logged-in users by maintaining session cookies. After a successful login, the system assigns a session cookie to the user, which is validated with each subsequent request.
    - **User Loaders**: Flask-Login provides a user_loader callback, enabling the system to load the currently authenticated user from the session. This allows for the retrieval of user-specific data, such as personalised movie recommendations, with each request.

- **Login Required Decorators**: The @login_required decorator in Flask-Login can be used to protect routes that should only be accessible to logged-in users. For instance, to access personalised movie recommendations:
- **Session Expiry and Security**: Flask-Login supports session expiry, enhancing security by automatically logging users out after a certain period of inactivity. The extension also supports the 'remember me' feature for prolonged sessions.

Integrating Flask-Login in the Movie Recommendation System

In the Movie Recommendation API, Flask-Login serves several key functions:

- After a user logs in, Flask-Login manages the session, ensuring the user remains authenticated for subsequent requests.
- Access to critical routes, such as those handling personalised movie recommendations, is restricted to authenticated users.
- Admin functionalities, such as adding or removing films, can be further protected by integrating role-based checks with Flask-Login to verify the user's permissions.
- Flask-Login's @login_required decorator ensures that unauthorised users are blocked from accessing sensitive routes.

Implementing strong authentication and authorisation mechanisms is vital for securing ScreenSort. Authentication principles, such as password security, session management, and protections against brute-force attacks, help prevent unauthorised access. Role-Based Access Control (RBAC), combined with the principle of least privilege, ensures that users are only granted access to functionalities appropriate for their role. Finally, Flask-Login (which is already included in the original code) simplifies session management by enabling secure authentication and session handling, ensuring that only authenticated and authorised users can access the system's key features.

---

5. Protecting Your Application Data

Authored by Jemma Lowman and Hana Devonald-Davis

Database security involves protecting the data stored within a database from various threats such as unauthorised access, data leaks, SQL injections, and tampering. Securing application databases involves implementing multiple layers of protection, ranging from restricting access to the database server to encrypting data in transit and at rest, and employing secure coding practices to prevent common vulnerabilities such as SQL injection.

In the context of a Flask-based movie recommendation API, such as ScreenSort, the primary goal is to secure data such as user profiles, preferences, reviews, and any authentication

credentials. Failing to protect this data can lead to privacy violations, loss of user trust, and potential legal ramifications.

Best Practices for Securing User Data

To safeguard user data in ScreenSort, the following best practices are recommended:

a. Network Security

- **Restricting IP Access**: The database server should be configured to accept connections only from trusted IP addresses (such as the web server hosting the Flask application or a specific VPN). This ensures that only authorised entities can access the database, reducing the potential attack surface.
  Implementation:
    - Use firewalls or cloud-based security services (e.g., AWS Security Groups) to whitelist specific IP addresses.
    - Leverage Virtual Private Network (VPN) services to limit access to internal networks and databases. Implement a VPN for administrative access to the database. This ensures that any remote access to the database is encrypted and only allowed through a secure, private network
- **Encryption**: All traffic between the client (API consumers) and the server must be encrypted using HTTPS/SSL to protect data during transit. For database communication, the connection should be encrypted using technologies like TLS.
- **Implementation Strategy**:
    - Enable HTTPS on the Flask API by setting up SSL certificates.
    - Use database-specific features, such as enforcing encrypted connections between the Flask app and the database (e.g., SSLMode for PostgreSQL or MySQL).
- **Firewall Configuration:**
    - Use a firewall to block all unnecessary inbound and outbound traffic. Only allow specific services (e.g., the database service on port 3306) and IP ranges to access the database server.
- **Network Segmentation**:
    - Segment the application and database servers into separate network zones. This prevents attackers who compromise the application layer from gaining direct access to the database. For instance, the database server can be placed behind a dedicated subnet or VLAN that only allows communication from the application server.

b. Authentication and Authorisation

- **Multi-Factor Authentication (MFA)**: MFA adds an additional layer of security by requiring users to authenticate with more than just a password. This reduces the likelihood of account compromise through password-based attacks.
  Implementation:

    - Implement MFA for administrative access to the database and for users accessing sensitive data (e.g., admin dashboards or API keys).
    - Use third-party identity providers (such as OAuth, Auth0, or Firebase Authentication) that support MFA for user authentication.
- **Role-Based Access Control (RBAC)**: Not all users should have the same level of access. Implementing RBAC ensures that users can only access the data and resources relevant to their role.
  Implementation:
    - Define roles such as "admin," "editor," or "viewer," and assign specific privileges based on these roles.
    - Restrict database queries so that users only interact with data relevant to their role (e.g., a regular user can only view their own movie preferences, not those of others).
- **Strong Password Policies**: Enforce strong password policies, including minimum length, complexity, and expiration policies for both user accounts and database administrator accounts.
    - In Flask backend, password validation techniques will be used to ensure that user accounts have strong passwords.
    - Minimum requirements, e.g. Minimum length of 8 Characters, minimum number of uppercase letters, minimum number of lowercase numbers, at least one digit and ‘special character’

c. Database Security Features

- **Principle of Least Privilege**: The application should follow the principle of least privilege, where users (and the application itself) only have the minimum level of access necessary to perform their tasks.
  Implementation:
    - Use database accounts with limited permissions. For example, the application should not have administrative access to the database (e.g., no DROP TABLE permissions).
    - Create separate database accounts for different purposes (e.g., read-only for regular queries, write access for administrative tasks).
- **SQL Injection Protection**: SQL injection attacks occur when attackers input malicious SQL queries into form fields or request parameters, allowing them to execute unintended commands on the database. Preventing SQL injection is a critical aspect of database security.
  Implementation:
    - Use **parameterised queries** or **prepared statements** to ensure user inputs are always treated as data and not executable SQL code.
    - Perform input validation on the server-side to sanitise and validate user inputs.
    - Use Flask security libraries like Flask-SeaSurf to prevent Cross-Site Request Forgery (CSRF), which can facilitate SQL injection attacks.

d. Data Encryption

- **Encryption at Rest**: Sensitive data such as user passwords and personal information should be encrypted in the database to prevent unauthorised access in the event of a data breach.
  Implementation:
    - Use strong encryption algorithms (e.g., AES-256) to encrypt sensitive data stored in the database.
    - Hash passwords using secure algorithms such as **bcrypt** or **Argon2** to ensure user passwords are not stored in plain text.
- **Encryption in Transit**: Use SSL/TLS to secure communication between the Flask application and the database, preventing attackers from intercepting sensitive data during transmission.
- **Encryption Key Management**
    - Access Control: Enforce role-based access control to limit who can decrypt sensitive data.
    - Key Rotation: Regularly rotate encryption keys to minimize risks of key exposure.

e. Database Monitoring and Logging

- **Auditing and Monitoring**: Regular monitoring of database access logs can help detect suspicious activities, such as repeated failed login attempts, unusual query patterns, or access from unauthorised IP addresses.
  Implementation:
    - Enable logging on the database server to track all access and query execution events.
    - Use tools like **AWS CloudWatch**, **ElasticSearch**, or **Splunk** to aggregate logs and monitor for anomalies.
    - Deploy an Intrusion Detection System (IDS) to detect potential security breaches.
- **Intrusion Detection System (IDS)**
    - Deploy database-specific IDS (e.g., **OSSEC**, **Snort**) to detect SQL injection, brute-force attacks, or privilege escalation attempts.
    - Use behavioural anomaly detection and configure alerts for immediate response to detected intrusions.
- **Data Integrity Monitoring**
    - Implement **Change Data Capture (CDC)** to track and log all modifications to sensitive data in real-time.
    - Store logs in an immutable format to ensure they cannot be tampered with, preserving integrity for forensic analysis.

f. Regular Security Audits and Patch Management

- **Database Security Audits**: Regularly audit database configurations, access controls, and user permissions to ensure compliance with security best practices.

Implementation

    - Access Control Review: Regularly audit database user roles, privileges, and access levels to ensure least privilege is enforced. Disable unused accounts and revoke unnecessary permissions.
    - Configuration Review: Check database configurations for security best practices (e.g., secure password storage, encryption settings, IP whitelisting) and ensure compliance with organisational security policies.
- **Patch Management**: It is essential to apply security patches and updates to both the database and the Flask application's dependencies to address known vulnerabilities. **Implementation**:
    - Use dependency management tools like pipenv or poetry to keep Python packages up to date.
    - Regularly review security bulletins for the database platform (e.g., PostgreSQL, MySQL) and apply security patches promptly.
    - Vulnerability Scanning Tools: Integrate automated tools like Snyk or Dependabot to continuously scan Flask application dependencies for known vulnerabilities and suggest updates.

Securing ScreenSort requires a multi-layered approach involving network security, authentication mechanisms like MFA, and database-specific security measures such as least privilege, encryption, and SQL injection prevention. By combining these strategies, the risk of unauthorised access, data breaches, and attacks is significantly reduced. Implementing these best practices ensures the protection of user data, fosters user trust, and maintains the integrity of the application.

---

6. Logging in and Monitoring analysis for Screensort

Authored by Hana Devonald-Davis

For the **ScreenSort: Movie Recommendation System**, logging and monitoring are crucial to ensuring the security of the application, especially given its integration with external APIs (TMDb), user authentication, and database connections. Here's how these concepts can be specifically applied to this app:

Role of Logging and Monitoring in Enhancing Security for ScreenSort

1. **User Authentication and Profile Management**: Logging can track authentication events (e.g. logins, failed login attempts, password resets) and ensure no unauthorised access. Monitoring tools can detect unusual login behaviour such as repeated failed attempts, which could indicate a brute-force attack.
2. **API Usage**: Since the app integrates with TMDb API, logging API requests and responses helps to monitor for anomalies, such as unauthorised access to the API or excessive requests, which could indicate abuse or attempted data scraping.
3. **Database Security**: Monitoring the interaction between the app and the MySQL database is critical. Logs should record all SQL queries, especially those that modify

user data or recommendation algorithms, to detect any SQL injection or other database attacks.

Purpose of Logging in Recording Critical Events for ScreenSort

Logging helps track and record key events that are critical for security and debugging in ScreenSort:

- **Authentication Events**: Logs should record successful and failed login attempts, account creation, password changes, and logouts. This can help identify compromised accounts or attempts to break into the system.
- **API Requests**: Record all outgoing and incoming API requests to and from the TMDb API, including the status of responses. This will help in debugging any issues with the third-party service as well as identifying misuse or tampering with API keys.
- **Database Queries**: Log database interactions to track changes to user profiles, watch histories, and the recommendations engine. This will help track any data manipulations and can be crucial for post-incident forensics in case of a breach.
- **Critical Errors**: Any server or application errors (e.g., failure to connect to MySQL, Flask app crashes, etc.) should be logged to ensure that issues can be addressed promptly.

How Monitoring Tools or Services Would Contribute to Threat Detection

For **ScreenSort**, monitoring tools can play a crucial role in ensuring the app's security and performance:

1. **Real-Time Threat Detection**: Monitoring tools (such as **Splunk** or **Datadog**) can provide real-time alerts when suspicious activity is detected, such as:
    - Multiple failed login attempts (potential brute force attack).
    - Sudden spikes in API calls to TMDb, which might indicate an automated attack or abuse of the API key.
    - Unusual database queries, such as repeated or mass changes to user profiles or watch histories.
2. **Performance Monitoring**: Monitoring tools can help identify if the app experiences sudden performance drops (e.g., high CPU/memory usage), which may indicate a DoS attack or other malicious activities affecting app availability.
3. **API Key Monitoring**: Since the TMDb API key is a critical part of the application, monitoring tools can be configured to alert on any unusual API key usage, such as requests originating from unexpected locations.

Additional Tools for Logging and Monitoring

To effectively manage logging and monitoring for the **ScreenSort** app, here are additional tools and services that could be used:

1. **SIEM (Security Information and Event Management)**:

    - **Splunk**: A powerful tool for aggregating, analysing, and alerting based on log data. It can be used to monitor Flask logs, MySQL queries, and TMDb API interactions in real-time.
    - **ELK Stack (Elasticsearch, Logstash, Kibana)**: An open-source alternative to Splunk, which can provide centralised logging, search functionality, and data visualisation.
2. **Application Performance Monitoring (APM)**:
    - **Datadog**: Can monitor Flask app performance and database interactions, tracking metrics like request rates, error rates, and response times.
    - **New Relic**: Another APM tool that can track the performance of the ScreenSort app in real-time, detecting anomalies that may indicate security issues.
3. **Cloud Monitoring (if deployed on the cloud)**:
    - **AWS CloudWatch** (if hosting on AWS) or **Azure Monitor** (for Azure-hosted apps) can monitor infrastructure and app-level logs, including API usage and database connections.
4. **Intrusion Detection System (IDS)**:
    - **Snort**: A network-based IDS that can detect and log potentially malicious network traffic, such as SQL injection attempts or unauthorised access attempts to the ScreenSort app.

Handling Software Updates and Patches for ScreenSort

To ensure that **ScreenSort** remains secure and up-to-date, a robust patch management process is critical:

1. **Automated Updates**: Use a package management tool like **Pipenv** or **Poetry** to ensure that all Python dependencies (as listed in Requirements.txt) are up to date with security patches.
    - Regularly update libraries like Flask, SQLAlchemy, and any third-party modules that interact with APIs (e.g., requests for handling TMDb API calls).
2. **Database and Server Patches**: Ensure that MySQL Workbench (or any SQL program in use) is regularly updated with security patches. For this, **MySQL Enterprise Monitor** can be used to keep track of any new patches and vulnerabilities.
3. **Testing Before Deployment**: Updates and patches (especially to dependencies like Flask, SQL libraries, or machine learning models) should first be tested in a staging environment to ensure compatibility with the app before going live.
4. **Continuous Monitoring After Updates**: After applying patches or updates, monitoring tools should be used to track any sudden changes in performance or unexpected behaviours. This ensures that the patches do not introduce new issues.
5. **Backup and Rollback Plan**: Before deploying any updates or patches, maintain a backup of the current system and database. A rollback plan ensures that you can revert to a stable version in case the updates cause issues or unexpected failures.

For **ScreenSort**, logging and monitoring are critical for identifying potential security threats, ensuring system performance, and maintaining compliance with data protection. By leveraging

tools like SIEMs, IDS systems, and automated patch management systems, the app can achieve better security, detect anomalies in real time, and ensure the ongoing integrity of its recommendation algorithms, user data, and API integrations.

---

7. Conclusion

In this report we have explored various methods through which our Flask application ScreenSort's security could be enhanced. Discussing the variety of threats an application like ScreenSort may have to contend with and the impact they could have and laying out guidelines for ensuring the code of the application has protections built in against such threats. To the role of HTTPS in ensuring confidentiality and security in the transfer of data from user to server and the importance of authentication and authorisation in keeping user data safe. Then the importance of ensuring the application's data is secure and the best practices to ensure its safety to finally the important role logging and monitoring play in keeping application data safe. Through the understanding and use of these methods, the security of ScreenSort can be improved and the user's trust in the platform would increase.