Security Hardening Report for A Web Application

Introduction

Ensuring the security of web applications is crucial to protect against common threats like SQL Injection, Cross-Site Scripting (XSS), and authentication bypasses. This report outlines the steps taken to harden as well as minimize attack surface a Flask based web application.

1. Enforcing HTTPS

```
Talisman(app)  # Enforces HTTPS
...
app.run(ssl_context=('cert.pem', 'key.pem'))  # Enable HTTPS
```

Purpose: Ensures that all traffic to and from the web application is encrypted.

Explanation: The `Talisman` extension is used to enforce HTTPS by setting HTTP headers that force secure connections. Additionally, the `ssl_context` argument is passed when running the Flask app, specifying the use of an SSL certificate (`cert.pem`) and key (`key.pem`), which enables HTTPS for the server.

2. Sanitization and Input Validation

```
class SecureForm(FlaskForm):
    username = StringField('Username', validators=[DataRequired(),
Length(min=3, max=20)])
```

Purpose: Ensures that user input is valid and secure.

Explanation: The `FlaskForm` class from `flask_wtf` is used to define a form with an input field for the username. The input is validated with two checks:

- `DataRequired`: Ensures the field is not empty.
- `Length(min=3, max=20)`: Enforces a length restriction to ensure the username is within a safe and expected range.

3. SQL Injection Protection

cur.execute("SELECT * FROM users WHERE username = ?", (username,)) # Secure Query
Purpose: Prevents SQL injection attacks by using parameterized queries.

Explanation: In the function `get_user`, the SQL query is written using a parameterized approach (`?` placeholder) to prevent SQL injection attacks. This ensures that user input (`username`) is treated as data and not executable code.

4. Multi-Factor Authentication (MFA)

```
from flask_security import Security, SQLAlchemyUserDatastore
from flask_limiter import Limiter
```

Purpose: Add an additional layer of security for user authentication (though MFA itself isn't fully implemented here, this portion suggests it).

Explanation: While MFA is not directly shown in this code, the `Security` module from `flask_security` can be used to implement multi-factor authentication (MFA). MFA typically requires users to provide something they know (password) and something they have (e.g., a code sent to their mobile device).

The use of `flask_security` along with other security mechanisms like rate limiting (shown below) provides a solid foundation for adding MFA.

5. Rate Limiting to Prevent Brute Force Attacks

```
@limiter.limit("4 per minute")  # Rate limiting to prevent brute-force
```

Purpose: Prevents brute-force login attempts by limiting the number of requests a user can make in a given time period.

Explanation: The `flask_limiter` extension is used to rate-limit the `/login` route, allowing only four requests per minute from the same IP address. This prevents attackers from trying multiple password combinations in a short period.

6. Securing HTTP Headers

```
Talisman(app, content_security_policy={
    'default-src': "'self'",
    'img-src': "'self' data:",
    'script-src': "'self'"
})
```

Purpose: Protects the application against certain types of attacks like cross-site scripting (XSS) by controlling which external resources can be loaded by the browser.

Explanation: The `content_security_policy` (CSP) header is set with specific directives:

- `'default-src': "'self'"`: Restricts resources to be loaded only from the same origin.
- `'img-src': "'self' data:"`: Allows images to be loaded only from the same origin or inline (base64 encoded).
- `'script-src': "'self'"`: Restricts JavaScript to be loaded only from the same origin.

7. Best Session Management Practices

```
app.config.update(
    SESSION_COOKIE_SECURE=True,  # Only send over HTTPS
    SESSION_COOKIE_HTTPONLY=True,  # Restrict JavaScript access
    SESSION_COOKIE_SAMESITE='Strict'  # Protect against CSRF
)
```

Purpose: Implements best practices for session security to prevent session hijacking and cross-site request forgery (CSRF).

Explanation: The application configuration ensures that:

- `SESSION_COOKIE_SECURE=True`: The session cookie is only sent over secure HTTPS connections.
- `SESSION_COOKIE_HTTPONLY=True`: The cookie cannot be accessed via JavaScript, reducing the risk of XSS attacks.
- `SESSION_COOKIE_SAMESITE='Strict'`: Prevents the cookie from being sent in cross-site requests, which protects against CSRF attacks.

Conclusion

This report outlines key security enhancements implemented in a web application to mitigate vulnerabilities. The security measures include enforcing HTTPS, validating user input, implementing MFA, securing headers, and applying best session management practices. These improvements enhance the application's resilience against cyber threats.