Securing Your Web Application

A Deep Dive into OWASP Top 3 Security Risks

Peeyush Sharma

CMPT 415
Simon Fraser University

Date: April 22, 2023

Abstract

Web applications have become a crucial component of business operations, but this increased reliance on them has also led to an increase in security vulnerabilities, leaving businesses exposed to cyber threats. The Open Web Application Security Project (OWASP) Top 10 Web Application Security Risks list outlines the most common vulnerabilities in web applications. The report analyzes the top three vulnerabilities, which are Broken Access Control, Injection, and Cross-Site Scripting, and provides an in-depth analysis of how the HAITI HHA project mitigates these risks.

The report includes recommendations on how to prevent the vulnerabilities, such as implementing access control mechanisms, rate-limiting API and controller access, and using stateful session identifiers. The report also assesses the HAITI HHA application for Broken Access Control and finds that it has a robust access control mechanism in place. However, it recommends areas for improvement, such as implementing input validation and output encoding, and logging access control failures.

By understanding and implementing the recommended security measures, organizations can ensure the security of their web applications, protect sensitive data from cyber threats, and mitigate the risks of security breaches.

1. Introduction

Web applications have become an essential tool for businesses to interact with their customers, employees, and partners. Increased reliance on web applications has led to an increase in risk of security vulnerabilities. Cybercriminals are always searching for vulnerabilities they can exploit in web applications to breach networks, steal sensitive data, or disrupt services. Therefore, it is essential to take measures to secure web applications and mitigate the risks that can lead to security breaches.

The OWASP Top 10 Web Application Security Risks is a list of the most common web application vulnerabilities that organizations need to be aware of. The Open Web Application Security Project (OWASP) regularly updates this list to reflect the evolving threat landscape. This report provides an in-depth analysis of the top 3 OWASP Web Application Security Risks and recommends best practices for mitigating these risks.

By understanding these risks and implementing the recommended security measures, organizations can help ensure the security of their web applications and protect their sensitive data from cyber threats. Each risk will be tied to project HAITI HHA to see how the vulnerabilities applies to it and how the project mitigate these risks. This report would also assess how if the project uses recommended solutions and areas of improvement for the application.

2. OWASP top 3 vulnerabilities

Following are the top 3 security risks as per OWASP Web Application Security Risks.

2.1. Broken Access Control

2.1.1. Overview

With the use of session management, a web application can easily respond to different service requests securely based on its authorized users [5]. A user can create a session using some sort of authentication such as username and password. Access control features ensures the restriction of accessing web resources such as web pages, database tables, etc. and it is the security configuration for preventing unauthorized access from the intruders.

It is evident from the current OWASP list, Broken Access Control (BAC) has been marked as 1st rank vulnerability depending on its existence in the web applications and the adverse consequences [1]. Broken access control has been found in high profile applications such as IIS and Wordpress [5].

A study conducted found 129 applications vulnerable to broken access control out of 330 web applications where the participant sectors were Education, E-Commerce, Govt, Health and private companies. The technology used for applications varied among PHP, Java, and .Net platform [5]. Lack of awareness about harmful consequences of Broken access control consequences can be the cause of this as designers and developers are not aiming for their application to not be secure.

2.1.2. Recommendations for prevention

Following are some recommendations on how Broken Access control can be prevented [1] :-

- Implement access control mechanisms once and re-use them throughout the application, including minimizing Cross-Origin Resource Sharing (CORS) usage.
- Log access control failures, alert admins when appropriate (e.g., repeated failures).
- Rate limit API and controller access to minimize the harm from automated attack tooling.
- Stateful session identifiers should be invalidated on the server after logout. Stateless JWT tokens should rather be short-lived so that the window of opportunity for an attacker is minimized. For longer lived JWTs it's highly recommended to follow the OAuth standards to revoke access.

2.1.3. Analysis of HAITI HHA Application - Broken Access Control

Tests were conducted on our application for broken access control.

Methodology:

Testing involved the use of four different user roles to ensure that access was appropriately limited based on role :-

- User
- Admin
- Head of Department
- Medical Director

During the testing process, attempts were made to bypass access control checks by modifying the URL, internal application state, or the HTML page. Additionally, direct GET and POST calls were made to the server using Postman to access APIs without proper authentication.

Assessment Findings:

- Despite these attempts, the application's security measures held strong, and access was consistently denied with a 401 Unauthorized response when appropriate authentication was not provided. These tests served to validate the effectiveness of the access control measures in place and provide assurance that unauthorized access to sensitive data is effectively prevented.
- Upon going through the code, the application has a very strong access control mechanism. Code snippet in Figure 1 on next page shows the route defined in an Express application that handles a POST request to the '/login' endpoint. The route uses a middleware function called 'requireLocalAuth' to authenticate the user before processing the request.

  Once the user is authenticated, the route handler retrieves the user's credentials from the request body and searches for a matching user in the MongoDB database using the Mongoose package. If a matching user is found, the user's details are converted to a JSON object and a JSON Web Token (JWT) is generated for the user.
- All the subsequent requests after logging in requires the above JWT token to be authenticated along with approriate user role before accessing any API's. Code snippet in Figure 2 on next page represents an Express route that handles DELETE requests to delete a case study. The route is mounted on the '/:id' endpoint and uses middleware that **requires JWT authentication and authorization as an Admin or Medical Director** to access the endpoint.

```
const router = Router();

router.post('/login',
    requireLocalAuth, async (req:
    RequestWithUser, res: Response)
    => {
  const user = req.body;
  const mongooseUser = await
      UserCollection.findOne({
      username: user.username });
  const jsonUser = await
      mongooseUser!.toJson();
  const token = mongooseUser!.
      generateJWT();
  res.cookie('jwt', token, {
      httpOnly: true });
  res
    .status(HTTP_OK_CODE)
    .json({ success: true, isAuth:
        true, user: jsonUser,
        csrfToken: req.body._csrf })
        ;
});
```

Figure 1. Code for Login

- The application also invalidates the JWT token as per the recommended best practices for prevention. Figure 3 represents an Express route that handles POST requests to logout a user and invalidate the JWT token.

Areas of Improvement

- Currently, the application does not log unsuccessful attempts to login or access API's. It would be nice to log access control failures and alert admins when there are repeated failures.
- The application can make use of Rate limiting API strategy to mitigate the impact of automated attacks on an application. Rate limiting is implemented by limiting the number of requests that can be made within a specific time period. The limit is typically set based on the application's usage patterns and the expected number of requests per user. If a client exceeds the limit, they are blocked or temporarily denied access.

  In Express, rate limiting can be implemented using third-party middleware such as 'express-rate-limit'. This middleware can be used to limit the number of requests per IP address within a specific time period. It can also be customized to include options such as response headers, error messages, and white-listed IP addresses.

```
router.delete(
  '/:id',
  requireJwtAuth,
  roleAuth(Role.Admin, Role.
      MedicalDirector),
  (req: RequestWithUser, res:
      Response, next: NextFunction)
      => {
    const caseId = req.params.id;
    CaseStudyCollection.
        findByIdAndRemove(caseId)
      .exec()
      .then((data: any) => {
        if (!data) {
          return next(new NotFound(`
              No case study with id
              ${caseId} found`));
        }

        return deleteUploadedImage(
            data.imgPath);
      })
      .then(() => res.sendStatus(
          HTTP_NOCONTENT_CODE))
      .catch((err: any) =>
        next(new InternalError(`
            Delete case study id ${
            caseId} failed: ${err}`)
            ),
      );
  },
);
```

Figure 2. Code for Deleting Case Studies

2.2. Cryptographic Failures

2.2.1. Overview

Cryptographic errors happen when encryption techniques are not used or used incorrectly. Some reasons for cryptographic failures include but not limited to [4]:-

- Sensitive information may be transmitted in clear text over a network or stored in databases or files in plaintext
- Use of legacy or weak encryption algorithms
- Mismanagement of encryption keys

```
router.post(
  '/logout',
  requireJwtAuth,
  (req: RequestWithUser, res:
      Response, next: NextFunction)
      => {
    res.cookie('jwt', 'invalidated-
        jwt-token');
    req.logout((err) => {
      if (err) {
        return next(err);
      }
    });
    logger.debug('User successfully
        logged out');
    res.send(true);
  },
);
```

Figure 3. Code for Logging out a User

- Use of insecure or default encryption keys or re-use of compromised keys

Cryptographic Failures has been ranked 2nd as a security vulnerability based on its existence in the web applications and the adverse consequences. [2].

2.2.2. Recommendations for prevention

Following are some recommendations on how Cryptographic Failures can be prevented [2] :-

- Classify data processed, stored, or transmitted by an application. Identify which data is sensitive according to privacy laws, regulatory requirements, or business needs.
- Don't store sensitive data unnecessarily. Discard it as soon as possible or use PCI DSS compliant tokenization or even truncation. Data that is not retained cannot be stolen.
- Make sure to encrypt all sensitive data at rest.
- Ensure up-to-date and strong standard algorithms, protocols, and keys are in place; use proper key management.
- Encrypt all data in transit with secure protocols such as TLS with forward secrecy (FS) ciphers, cipher prioritization by the server, and secure parameters. Enforce encryption using directives like HTTP Strict Transport Security (HSTS).

2.2.3. Analysis of HAITI HHA Application - Cryptographic failures

An assessment was conducted on the application in accordance with the guidelines set by the GDPR and industry-standard cryptographic best practices.

Methodology:
The assessment covered the following areas:

- Data protection principles: The assessment evaluated how the application handles personal data and whether it complies with the GDPR's data protection principles.
- Cryptographic mechanisms: The assessment evaluated the cryptographic mechanisms used by the application to protect data in transit and at rest. This included an assessment of the strength of encryption, key management practices, and other cryptographic controls.

Assessment Findings:
The assessment revealed the following findings:

- GDPR Compliance: Assessment found that the application meets the requirements as there is no personal identifiable information (PII) being collected in any of the forms. The only instance where PII was found was in the case studies section. However, we assume that HHA has obtained proper permissions from the individuals involved before highlighting their cases on the application
- Cryptographic mechanisms: The application uses HTTP which is an unencrypted protocol that sends data in plain text format. As a result the application is vulnerable to eavesdropping, interception, and modification of the data in transit. This means that any sensitive data sent over HTTP can be intercepted and stolen by attackers. Also, none of the data (except user password) is encrypted at rest.

Areas of Improvement

- While the application does not collect personal identifiable information, there should still be an exercise done to classify all the information stored in the application and then handle data according to GDPR compliance measures in the event that the application does collect such information in the future. This includes providing clear and concise privacy policies, obtaining explicit consent for data collection, implementing mechanisms for data subject requests. Additionally, for case studies where personal identifiable information is shared, proper consent should be obtained from individuals and mechanisms should be built to handle data access and erasure requests.

- Sensitive data, such as case studies, stored in the application is not encrypted at rest. There should be strong encryption mechanisms to protect data at rest, such as encryption of the database. This will ensure that even if the data is stolen, it cannot be accessed without the appropriate decryption keys. Decryption can slow down the availability of data, so it is necessary to identify classify data and only encrypt data that is necessary.
- To address the cryptographic weaknesses, It is recommended that HTTPS encryption should be implemented throughout the application. This would help to protect sensitive data in transit and mitigate the risk of eavesdropping, interception, and modification. HTTPS provides a secure channel between the client and the server, ensuring that any data transmitted is encrypted and cannot be read or modified by an attacker.

2.3. Injection

2.3.1. Overview

Injection has been ranked 3rd by OWASP as a security vulnerability based on its existence in the web applications and the adverse consequences [3]. With the advancement of big data and cloud computing technologies, NoSQL databases are gaining popularity. Similar to typical SQL injections, NoSQL injections are simply one of several types of injection attacks [6]. Some reasons for NoSQL injection include but not limited to [3]:-

- The application does not validate, filter, or sanitize user-supplied data.
- Application makes use of dynamic queries or non-parameterized calls without context-aware escaping.
- Hostile data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records.
- The application directly uses or concatenates hostile data in SQL or commands, allowing for malicious code to be executed in dynamic queries, commands, or stored procedures.

2.3.2. Recommendations for prevention

Following are some recommendations on how Injection can be prevented:

- Keep data separate from commands and queries to avoid injection.
- It is recommended to use a safe API that avoids using the interpreter or provides a parameterized interface. Object Relational Mapping Tools (ORMs) can also be used.
- Implement server-side input validation to prevent attacks. However, some applications require special characters.
- Use escape syntax to escape special characters in any residual dynamic queries.
- Use SQL controls such as LIMIT to prevent mass disclosure of records in case of an injection attack.

2.3.3. Analysis of HAITI HHA Application - NoSQL Injection

An assessment was conducted on the application to check if the application is vulnerable to NoSQL injection attacks.

Methodology:
The security analysis was conducted using manual testing which involved examining the application's source code to identify any areas where user input is accepted and processed, and then attempting to inject malicious input to determine if the application is vulnerable to NoSQL injection attacks.

Assessment Findings:

- The application uses Mongoose library which significantly reduces the risk of NoSQL injection vulnerabilities. Mongoose provides a schema-based solution for modeling application data and provides a built-in mechanism for validating and sanitizing user inputs. Additionally, Mongoose supports parameterized queries, which further reduces the risk of injection vulnerabilities.
- There is no server side validation done on the input while submitting forms. While Mongoose can reduce the risk of injection vulnerabilities, it's still critical to properly validate and sanitize user inputs on the server-side. If input validation is not implemented properly, it can lead to

security issues such as allowing attackers to bypass client-side validation and sending invalid data to the server, or triggering unexpected errors.

Areas of Improvement

- Implement server-side validation: While the Mongoose library provides some protection against NoSQL injection, it is still recommended to implement server-side validation to ensure that user inputs meet expected data types and formats. This will help to prevent any potential security vulnerabilities that may arise from user inputs that are not properly sanitized.
- Keep the Mongoose library up to date: It is important to ensure that the Mongoose library is updated to the latest version to ensure that any potential security vulnerabilities are addressed.

3. Conclusion

The HAITI HHA project, which is the focus of this report, underwent tests to ensure that access was appropriately limited based on the user's role. The tests showed that the application had a strong access control mechanism in place, and the security measures held strong, consistently denying access with a 401 Unauthorized response when appropriate authentication was not provided. The application also required JWT token authentication and authorization to access endpoints, which ensured the security of sensitive data.

Despite the robust security measures, the report recommended some areas of improvement. For instance, the application could implement input validation and output encoding to prevent injection attacks. Additionally, logging access control failures and alerting administrators when appropriate could help improve the security of the application further.

In conclusion, by understanding the OWASP Top 10 Web Application Security Risks and implementing recommended security measures, organizations can mitigate the risks of security breaches, protect sensitive data from cyber-attacks, and ensure the security of their web applications.

References

[1] *A01:2021 – Broken Access Control*. URL: `https://owasp.org/Top10/A01_2021-Broken_Access_Control/`.

[2] *A02:2021 - Cryptographic Failures*. URL: `https://owasp.org/Top10/A02_2021-Cryptographic_Failures/`.

[3] *A03:2021 – Injection*. URL: `https://owasp.org/Top10/A03_2021-Injection/`.

[4] Essohanam Djeki et al. "Preventive Measures for Digital Learning Spaces' Security Issues". In: *2022 IEEE Technology and Engineering Management Conference (TEMSCON EUROPE)*. 2022, pp. 48–55. DOI: `10.1109/TEMSCONEUROPE54743.2022.9801945`.

[5] Hassan M. *Quantitative assessment on broken access control vulnerability in web applications, International Conference on Cyber Security and Computer Science 2018*. Oct. 18, 2020.

[6] Sivakami Praveen, Alysha Dcouth, and A S Mahesh. "NoSQL Injection Detection Using Supervised Text Classification". In: *2022 2nd International Conference on Intelligent Technologies (CONIT)*. 2022, pp. 1–5. DOI: `10.1109/CONIT55038.2022.9848017`.