Security Audit of the Safe Places Server and Application

Revision 1.1
Updated June 7, 2020
Author(s): Aiyan Jiang <aiyan.jiang@outlook.com>

Table of Contents

Overview

The Safe Places server may be deployed in countless jurisdictions around the world and may be relied on by millions to accurately notify them if they encounter someone later tested positive. With widespread usage comes an increasing demand for security and accuracy of information.

> We should therefore make security and privacy first-class citizens in our development of the Safe Places server.

Since the Safe Places application and server are both open-source, attackers can trivially identify and exploit security flaws.

Rather than relying on the public health authority's integration team to secure the service, Safe Place should strive to employ best security practices and follow high-level guidelines that the public health authority can adopt.

Link to Jira epic: Patch security vulnerabilities in the Safe Places backend and frontend.

Discovered Vulnerabilities

The following vulnerabilities have been discovered in the Safe Places application.

Each vulnerability includes a detailed description of the issue, coupled with a recommended solution to the problem.

JWT sidejacking

Risk level: High

Issue: Attackers can steal a legitimate user's JSON Web Token (JWT) and use it to make requests, potentially accessing sensitive data and performing unauthorized actions.

Solution: We can add an additional layer of protection to token theft through token contextualization.

1. We generate a random string during the authentication phase and send it to the client as a hardened cookie. The cookie must have the flags `HttpOnly`, `Secure`, and `SameSite`.
2. We store a SHA-256 hash of the random string in the JWT as context.
3. Client requests from the SPL Web App send both the JWT and cookie context.

4. On the server, we hash the cookie context string and compare it to the corresponding value in the JWT. If they are different, then we reject the request.

JWT storage in browser local storage

Risk level: Medium

Issue: The JWT returned by the server after a successful login is stored in the browser local storage. The local storage is susceptible to cross-site scripting (XSS) attacks, as the local storage can be accessed using Javascript.

Therefore, a hacker can inject a script that reads and steals the JWT.

> **Note:** A vulnerability leading to a successful XSS attack can be either in the source code or in any third-party JavaScript code (such as bootstrap, jQuery, or Google Analytics) included in the application.

Solution: JWTs should be stored in a browser cookie with the `Secure` and `HttpOnly` flags. These flags ensure the cookie is sent securely over HTTPS and cannot be accessed using Javascript.

To mitigate cross-site request forgery (CSRF) attacks that may be exposed after browser cookie usage, we can employ a CSRF token. See the cross-site request forgery section for more information.

Credential stuffing

Risk level: High

Issue: The Safe Place server is vulnerable to brute-force login attacks in which the attacker tries many different passwords, some from other security breaches, to log in.

Solution: The backend should rate-limit login requests and return an appropriate response (see RFC 6585 section 4) to indicate the restriction in place. An example response would be

```
HTTP/2 429 Too Many Requests
Content-Type: application/json
Retry-After: 15

{
  “error”: “Too many requests. Please try again later.”
}
```

The rate limiting can be achieved by using an in-memory database, such as Redis, to record the request timestamps from an IP address after every request. The database is queried with every request to determine whether the cooldown has passed.

Note: the cooldown time can be increased with every request, either linearly or in a fibonacci-like sequence.

LDAP injection into login

Risk level: Very High

Issue: The Express authentication middleware runs an LDAP search and filter using the `username` and `password` provided in the login request. Neither are checked for special characters used in distinguished names (DNs) or search filters.

Similar to the consequences of SQL injection, LDAP injection can grant the attacker access to an account without knowing the correct credentials.

Example: Let the search filter be

```
`(&(username=${username})(password=${password}))`
```

An attacker can submit the username admin)(&) and any password, which would result in the filter

```
`(&(username=admin)(&))(password=AnyPasswordHere))`
```

Since only the first filter is processed by the LDAP server, the query is always true.

> Just like that, the attacker can access the admin account *without ever needing the correct password.*

Solution: Enforce a strict charset for the username. Typical username restrictions require the username to match the regular expression

```
^[a-zA-Z0-9_-]{3,20}$
```

Moreover, escape all special characters in the password. In Node.js, this can be achieved through an escape library with an LDAP encoding function. All variable substitutions derived from user input should be passed through this function before being incorporated into the search query or filter.

SQL injection via Knex raw queries

Risk level: Low

Issue: Javascript string templating is used for some Knex.js raw queries. See this example line in the Safe Places backend code:

```
const results = await this.raw(`SELECT ST_SetSRID(ST_MakePoint(${longitude},
${latitude}),4326) AS point, now() AS time`);
```

See the GitHub permalink to this line. Although this query is only used internally and shielded from external injection, Safe Places should adhere to the practice of extracting parameters from the query.

Solution: Pass the variables as a parameter binding in Knex raw or use Knex's query builder.

```
const results = await this.raw( ‘ SELECT ST_SetSRID(ST_MakePoint(?, ?),4326) AS
point, now() AS time ’ , [longitude, latitude]);
```

Correspondingly, in the service.js file:

```
raw(string, params) {
  return knex[this._scope].raw(string, params);
}
```

Lack of content security policy

Risk level: Low

A content security policy is important in the mitigation against XSS attacks. A CSP-compatible browser will follow the policy and prevent the loading of resources that are not allowed by the policy.

Currently, the Safe Places server does not send a content security policy to the Safe Places application. An example CSP would be

```
Content-Security-Policy: default-src 'self' *.trustedpha.org; img-src *
```

The policy allows the loading of all resources from the site's own origin as well as subdomains. Moreover, it allows the loading of images from any origin.

> The key component of the policy is that it *prevents the loading of scripts from unauthorized origins.* Therefore, the Safe Places application possesses resilience against XSS attacks.

Refer to the MDN technical specification for more information.

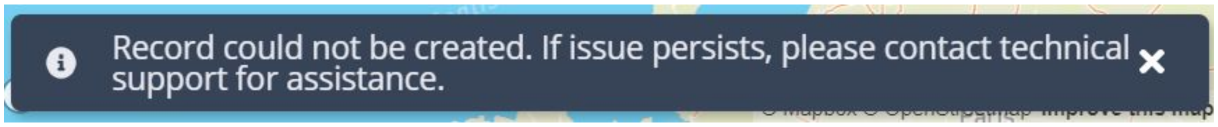
Possible Future Vulnerabilities

Current application interaction

Behavior recorded at 2020-06-06T23:00:00-04:00.

The JWTs are issued from the server with an expiration of 3600 seconds, or 1 hour. The Safe Places application uses the token to access protected endpoints.

Currently, token expiration is handled with a poor user experience. For example, if the "add new record" button is clicked after the token's expiration, the following message appears:


Record could not be created. If issue persists, please contact technical support for assistance.

The desired behavior is for the application to redirect the user to a login page for the server to issue a new JWT.
However, this can be a pain for the user to log in again after every hour of usage. Therefore, we foresee several possibilities that may be taken by the front end team to improve user experience:

1. A new JWT will be issued after every successful request by an existing JWT. The new JWT with a later expiration date would replace the existing JWT.

2. A refresh token will be issued along with a JWT after the initial successful login. This refresh token has no expiration date but can be blacklisted by the Safe Places server. Its purpose is to allow the web application to receive a new JWT when nearing the expiration date.

Both of these paths are extremely common in user-centric web applications on the internet, as it provides a positive user experience. However, both require additional security measures.

The following are recommended security measures in case one of these paths is chosen.

Refresh token attack surface

If a refresh token is used, then store the token as a browser cookie using the `HttpOnly` and `Secure` flags. The server should maintain an in-memory database of blacklisted refresh tokens, which should be referenced on every refresh request.

Cross-site request forgery

Currently, all access tokens are stored in the browser's local storage. Hence, there is no issue with CSRF. However, if they are later saved as browser cookies, then the double-submit cookie pattern should be employed to protect against CSRF attacks.

1. The user receives a CSRF token as a secure browser cookie after a successful login.
    a. The token needs to have sufficient entropy and be a cryptographically strong pseudorandom value.
    b. The token should also be stored in an appropriate claim in the JWT. This allows for stateless verification of the CSRF token.
2. On every request, the Safe Places application fetches the CSRF token browser cookie using Javascript and sends it in the `X-CSRF-TOKEN` header.
3. The Safe Places server compares the CSRF token stored in the JWT with that sent in the request header. If they are different, then the request is rejected.

Security Resources

For more information about these attack surfaces, refer to the following useful resources on security practices:

- OWASP Cheat Sheet Series
- OWASP Top Ten Web Application Security Risks
- OWASP Web Security Testing Guide