SE4030 – Secure Software Development

Assignment 01

Name	Student ID
Jayasinghe J.A.M.P.	IT21268144
Wickramasinghe W.A.R.M.	IT21233494
Bandara S.Y.T.D.	IT21185052
Shabry S.M.	IT21296796

Table of Contents

About

The OneTel Mobile Accessories Shop replaces its manual system with automation to benefit the business by improving customer experiences and problem-solving. The developed automation allows the following features:

- **QR System for Item Specification:** It would be easy for a customer to reach the specifications of any product used, and they will not need to spend much time surfing the website or looking somewhere else through advertising. That will save time and resources both for the customer and the seller.

- **Live Chat for After-Sales Support:** In case of problems with their products, the customer can easily go into live chat with representatives of the store, and problems may get sorted without them actually having to visit the store. It saves time and increases convenience.

- **Order Tracking:** There will be a tracking system put in place that will keep customers updated about the status of their orders in real time. It provides full transparency with less confusion once the purchase has been made.

- **Rental Management:** The system shall require the customer to upload his or her ID and personal information to reduce losses of rental items at least the store shall track them. In cases of loss, the accountability shall fall on the customer with ease.

Generally, this automated system tends to bring OneTel's operations up to date by adding efficiency, customer satisfaction, and security.

Testing Tools

- Sonar Qube
- OWASP Zap
- Snyk

Vulnerabilities - Manual testing

NoSQL Injection Prevention

The application was vulnerable to NoSQL injection attacks. This issue arose because unvalidated user inputs were directly passed into the database query, allowing attackers to inject malicious commands.

```
// Login route
router.post('/login', (req, res) => {

    User.findOne({ email: req.body.email })
      .then(user => {
        if (user) {
          if (bcrypt.compareSync(req.body.password, user.password)) {
            const payload = {
              _id: user._id,
              first_name: user.first_name,
              last_name: user.last_name,
              email: user.email
            };
            let token = jwt.sign(payload, process.env.SECRET_KEY, { expiresIn: 1440 });
            res.send(token);
          } else {
            res.status(401).json({ error: 'Invalid password' });
          }
        } else {
          res.status(404).json({ error: 'User not found' });
        }
      })
      .catch(err => {
        res.status(500).json({ error: 'Server error' });
      });
});
```

Fix Implemented:

- Input validation has been added to prevent NoSQL injection. Specifically, the login route now checks if the email input is of the correct type (string) before querying the database. This ensures that only valid strings are used in the query, blocking any NoSQL injection attempts.

```javascript
// Login route
router.post('/login', (req, res) => {

    // Prevent NoSQL injection by validating input type
    if (typeof req.body.email !== 'string'){
        return res.status(400).json({ error: 'Invalid input type' });
    }

    User.findOne({ email: req.body.email })
      .then(user => {
        if (user) {
          if (bcrypt.compareSync(req.body.password, user.password)) {
            const payload = {
              _id: user._id,
              first_name: user.first_name,
              last_name: user.last_name,
              email: user.email
            };
            let token = jwt.sign(payload, process.env.SECRET_KEY, { expiresIn: 1440 });
            res.send(token);
          } else {
            res.status(401).json({ error: 'Invalid password' });
          }
        } else {
          res.status(404).json({ error: 'User not found' });
        }
      })
      .catch(err => {
        res.status(500).json({ error: 'Server error' });
      });
});
```

Denial of Service (DoS) via Excessive Request Flooding

A Denial of Service (DoS) attack can occur when an attacker sends an overwhelming number of requests to the server in a short period of time. This type of attack, known as Request Flooding, can degrade the server's performance or cause it to become unresponsive by exhausting resources like CPU, memory, or bandwidth.

To prevent such attacks, a rate-limiting mechanism can be applied, restricting the number of requests a client can send within a specific timeframe. If the limit is exceeded, subsequent requests from the same client are blocked for a set duration.

Fix Implemented:

- The implemented solution involves using the rate-limiting middleware from Express to protect the application from excessive request flooding. By setting a limit of 100 requests per IP address within a 15-minute window, the server can prevent a potential attacker from overwhelming the application with rapid, repeated requests. If the threshold is reached, the client will be blocked temporarily.

```javascript
const bodyParser = require('body-parser');
const express = require('express');
const mongoose = require('mongoose');
const cors = require('cors');
const app = express();
app.set('trust proxy', 1);
const rateLimit = require('express-rate-limit');

//prevent DoS attack
const limiter = rateLimit({
    windowMs: 15 * 60 * 1000, // 15 minutes
    max: 100, // Limit each IP to 100 requests per windowMs
    message: "Too many requests, please try again later."
});

app.use(limiter);

const PORT = process.env.PORT || 8070;
```

Insecure Direct Object References (IDOR)

IDOR: An application discloses an internal implementation object reference-for example, a file or record in the database-without proper authorization checks on it. This can be tampered by the attacker to provide unauthorized data access.

Fix Implemented:

- Implemented in React a ProtectedRoute component, using JWT and the role in the payload of the token to check if there was admin privilege, hence give or not access to certain routes.
- Added extra protection to the back-end routes by verification of a token and checking user's role prior to serving protected resources.

onetel > client > src > JS ProtectedRoute.js > ProtectedRoute

```
import React from 'react';
import { Navigate } from 'react-router-dom';
import jwt_decode from 'jwt-decode';

const ProtectedRoute = ({ children, roleRequired }) => {

  const token = localStorage.getItem('token');

  // If there is no token, redirect to login
  if (!token) {
    return <Navigate to="/login" />;
  }

  try {
    // Decode the token to check user role or expiration
    const decoded = jwt_decode(token);

    // Check if the token has expired
    const currentTime = Date.now() / 1000; // in seconds
    if (decoded.exp < currentTime) {
      // If expired, clear token and redirect to login
      localStorage.removeItem('token');
      return <Navigate to="/login" />;
    }

    // If the roleRequired is "admin", check if the user has the admin role
    if (roleRequired === 'admin' && decoded.role !== 'admin') {
      return <Navigate to="/" />;
    }

    // If everything checks out, allow access to the component
    return children;

  } catch (err) {
    console.error('Error decoding token:', err);
    return <Navigate to="/login" />;
  }
};

export default ProtectedRoute;
```

Route>Admin.js

```
router.post('/admin/login',verifyToken, (req, res) => {
    Admin.findOne({ email: req.body.email })
      .then(admin => {
        if (admin) {
          console.log('Stored Hashed Password:', admin.password);
          console.log('Plain Password:', req.body.password);

          // Compare the hashed password with the plain password
          if (bcrypt.compareSync(req.body.password, admin.password)) {
            const payload = {
              _id: admin._id,
              email: admin.email,
              role:"admin"
            };

            // Generate JWT token
            let token = jwt.sign(payload, process.env.SECRET_KEY, { expiresIn: 1440 });
            res.json({ token: token });  // Send token to client
          } else {
            res.status(401).json({ error: 'Invalid password' });
          }
        } else {
          res.status(404).json({ error: 'Admin not found' });
        }
      })
      .catch(err => {
        res.status(500).json({ error: 'Server error' });
      });
  });
```

Regular Expression Denial Of Service (REDOS Attack)

Regular Expression Denial of Service (REDOS) attacks exploit the inefficiencies in regular expressions (regex). When poorly designed regular expressions are used in input validation, especially with nested quantifiers, malicious actors can craft inputs that take an exponentially long time to process. An attacker can use this vulnerability to cause a program using a regular expression to hang for a very long time. This causes the system to use excessive CPU resources, effectively creating a Denial of Service (DoS) scenario.

Code Vulnerability- Email Validation Using Regex (REDOS Risk):

In this snippet, an email is validated using a regular expression. While this pattern seems simple, if crafted with certain input, it can lead to REDOS. For example, certain combinations of characters can cause the regex engine to take much longer to evaluate the input, resulting in performance degradation or system unavailability.

```
// Validate email format without validator(ReDos Attack)

const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
 if (!emailRegex.test(email)) {
   toast.error('Invalid email address');
   return;
 }
```

Why This Code Is Vulnerable:

1. **Complexity of Regex:** The regex includes multiple quantifiers (+), which can cause catastrophic backtracking if the input is sufficiently complex.
2. **Lack of Input Length Limitation:** Without proper constraints on the input size, the risk increases if the attacker inputs an abnormally large or specifically designed string.

Preventing REDOS: Using Validator Library:

By switching to a library like validator, the risk of REDOS is minimized because this library's functions are optimized for safety and performance. The built-in isEmail() function does not rely on complex regex patterns that are vulnerable to REDOS attacks.

```
// Validate email format

// Validate email using validator

if (!validator.isEmail(this.state.email)) {
  toast.error('Invalid email format');
  return;
}
```

How This Prevents REDOS:

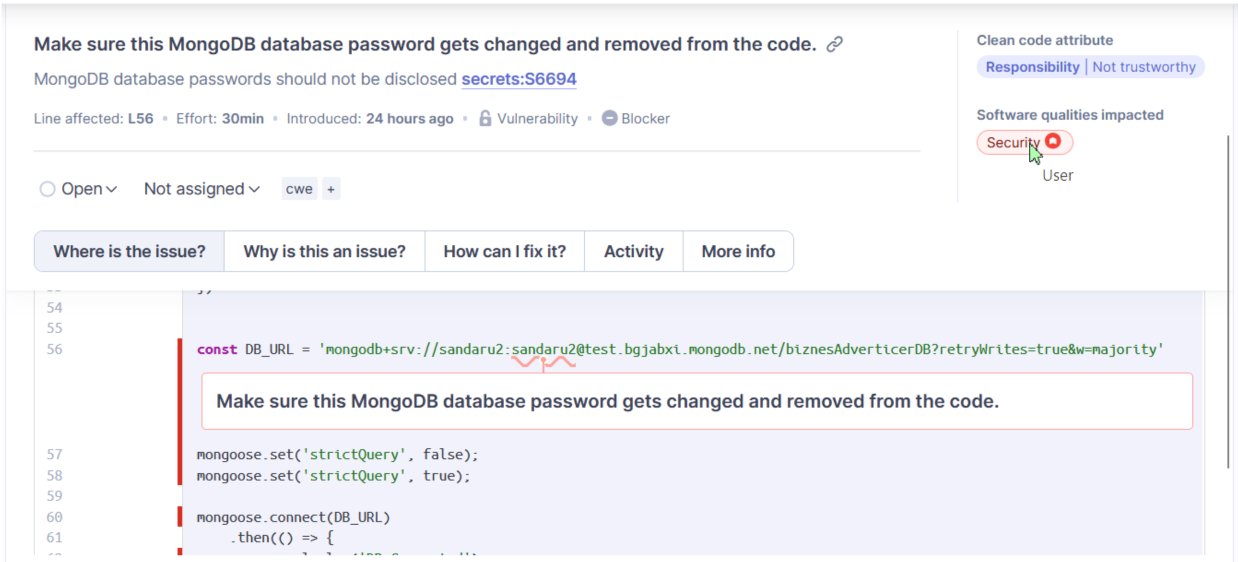
- **Pre-optimized Functions:** The validator library is built to avoid inefficiencies and vulnerabilities like REDOS by internally handling edge cases.
- **No Regex Complexity:** Instead of manually handling regex, the library uses secure methods that are designed to handle all email formats efficiently without the risk of backtracking or excessive computation time.

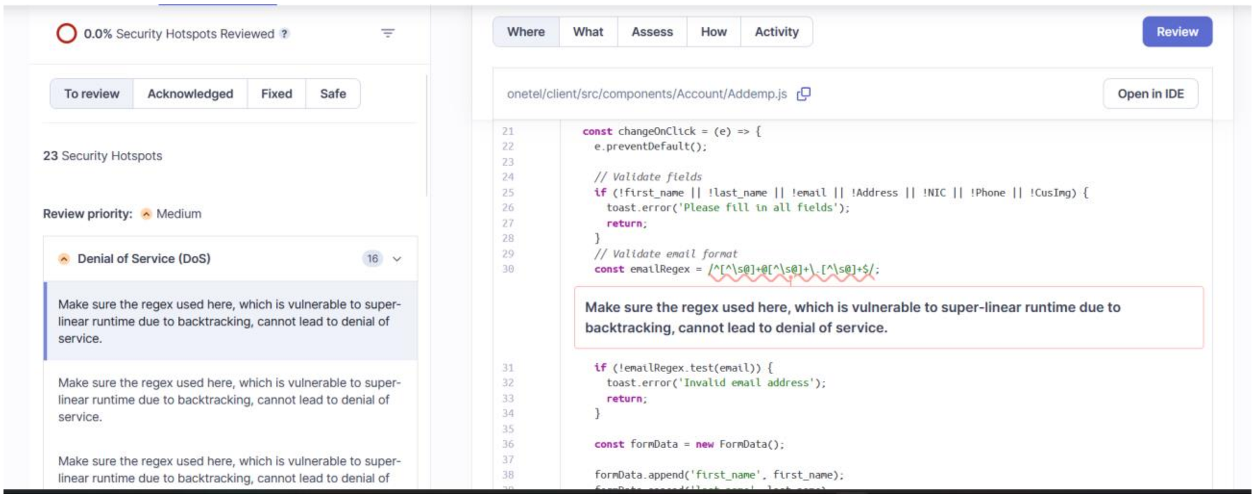
By migrating from manual regex to a well-tested library like validator, your email validation process becomes not only more reliable but also significantly more secure against REDOS attacks.

Vulnerabilities – Detected by Tools

Sensitive Data Exposure

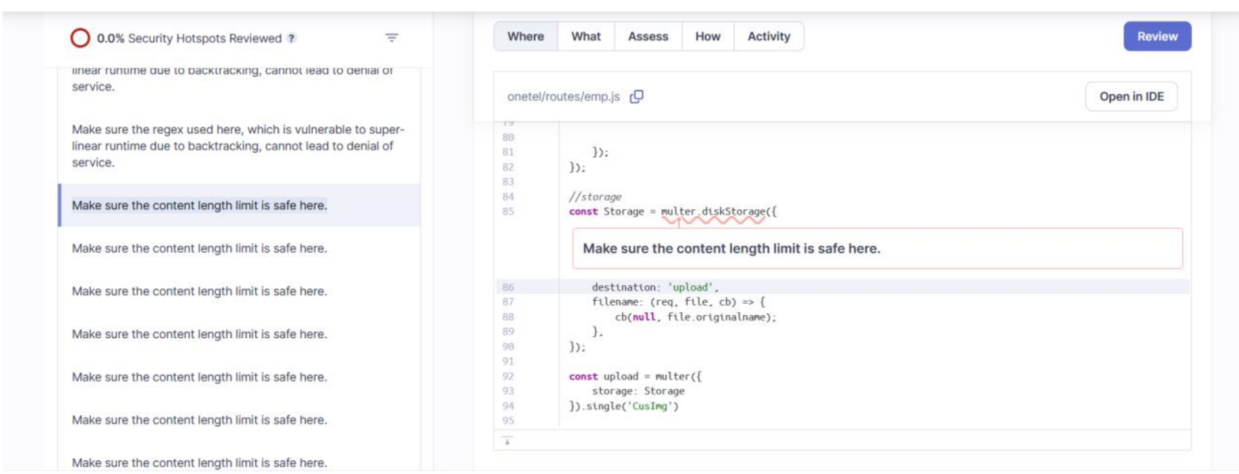
sensitive information, such as authentication credentials (like MongoDB passwords), is hardcoded in the source code. This practice poses a significant security risk, especially when the code is shared or deployed across multiple environments. If an attacker gains access to this information, they can potentially exploit it to compromise the database, access sensitive data, and undermine the security of the entire application.

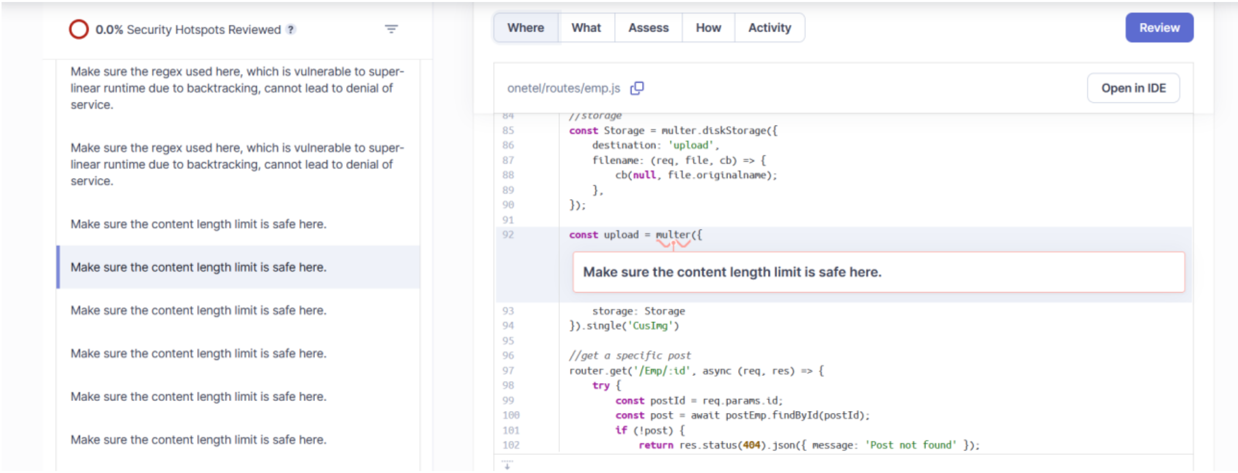




Denial of Service (DoS) via Uncontrolled Resource Consumption

When an application doesn't limit the size of the requests (e.g., file uploads, JSON bodies), attackers can send excessively large payloads, causing the system to consume all available memory, CPU, or bandwidth.






```
//this fixed accordingly sonar qube
// const upload = multer({
//      storage: Storage
// }).single('CusImg')

//fixed
const upload = multer({
    storage: Storage,
    limits: {
        fileSize: 8000000 // 8MB limit
    }
}).single('file');
```

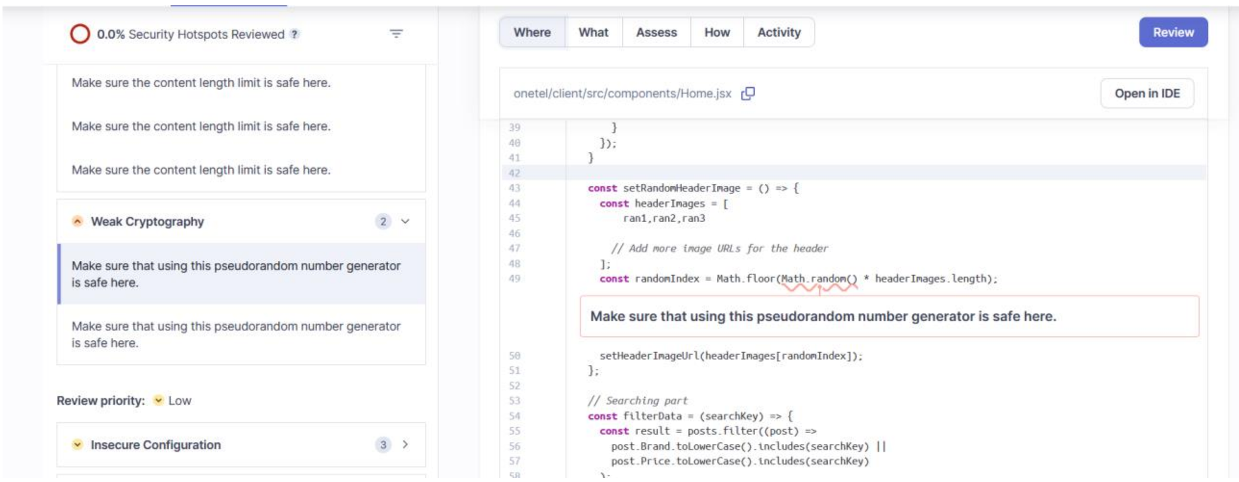
Using Weak Pseudorandom Number Generators (PRNGs)

Using pseudorandom number generators (PRNGs) is security-sensitive. For example, it has led in the past to the following vulnerabilities:

- CVE-2013-6386
- CVE-2006-3419
- CVE-2008-4102

When software generates predictable values in a context requiring unpredictability, it may be possible for an attacker to guess the next value that will be generated, and use this guess to impersonate another user or access sensitive information.

As the Math.random() function relies on a weak pseudorandom number generator, this function should not be used for security-critical applications or for protecting sensitive data. In such context, a cryptographically strong pseudorandom number generator (CSPRNG) should be used instead.

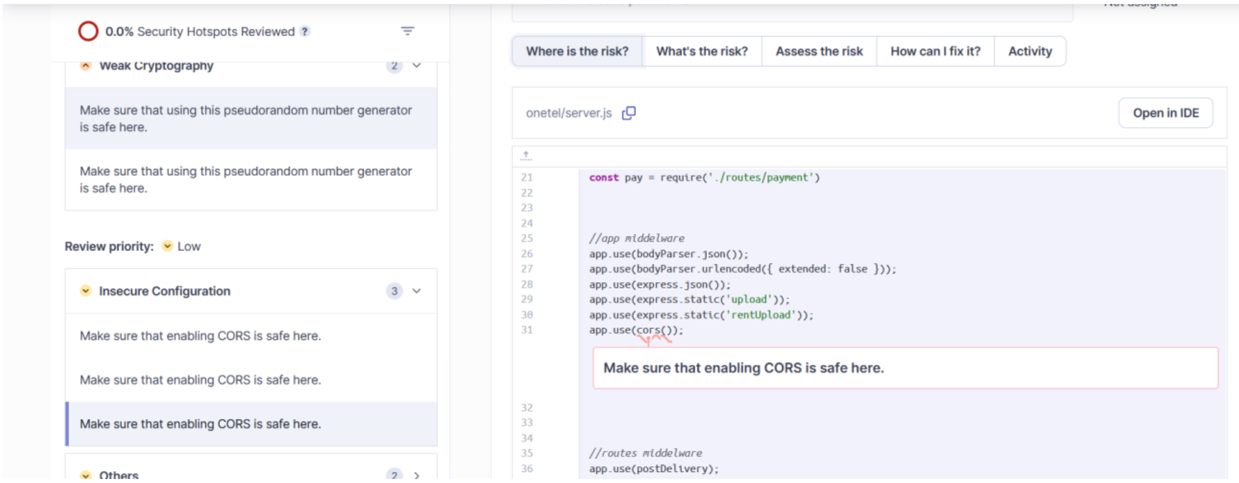


Preventing Using Weak Pseudorandom Number Generators.

```
//issue accordingly sonar qube
  // const setRandomHeaderImage = () => {
  //   const headerImages = [
  //       ran1,ran2,ran3

  //     // Add more image URLs for the header
  //   ];
  //   const randomIndex = Math.floor(Math.random() * headerImages.length);
  //   setHeaderImageUrl(headerImages[randomIndex]);
  // };

  //this is fixed one
  const setRandomHeaderImage = () => {
    const headerImages = [ran1, ran2, ran3]; // Your image URLs

    const crypto = window.crypto || window.msCrypto; // Use browser CSPRNG
    const array = new Uint32Array(1); // Generate a secure random number
    crypto.getRandomValues(array); // Get a cryptographically secure random number

    const randomIndex = array[0] % headerImages.length; // Ensure index is within array bounds
    setHeaderImageUrl(headerImages[randomIndex]); // Set random image
};
```

Permissive Cross-Origin Resource Sharing (CORS) Policy.

CORS is a browser security feature that restricts web applications from making requests to a different domain than the one that served the web page. By default, the **Same-Origin Policy** blocks cross-origin requests to protect against attacks such as **Cross-Site Request Forgery (CSRF)** or **Cross-Site Scripting (XSS)**. However, having a permissive CORS policy (e.g., using a wildcard * in Access-Control-Allow-Origin) can expose your web application to security risks. Attackers could exploit this vulnerability to make unauthorized cross-origin requests, potentially compromising user data or application functionality.



Preventing Permissive Cross-Origin Resource Sharing (CORS) Policy vulnerabilities.

```
// fix the issue
// Define trusted origins for CORS policy
const corsOptions = {
    origin: ['http://localhost:3000'],  // Replace this with the trusted domains
    optionsSuccessStatus: 200 // For legacy browser support
  };

  // Apply CORS middleware with restricted origins
router.use(cors(corsOptions));  // UPDATED: Restrict CORS to trusted origins only
```

Unverified Remote Resource Inclusion

Using remote artifacts without integrity checks can lead to the unexpected execution of malicious code in the application.

On the client side, where front-end code is executed, malicious code could:

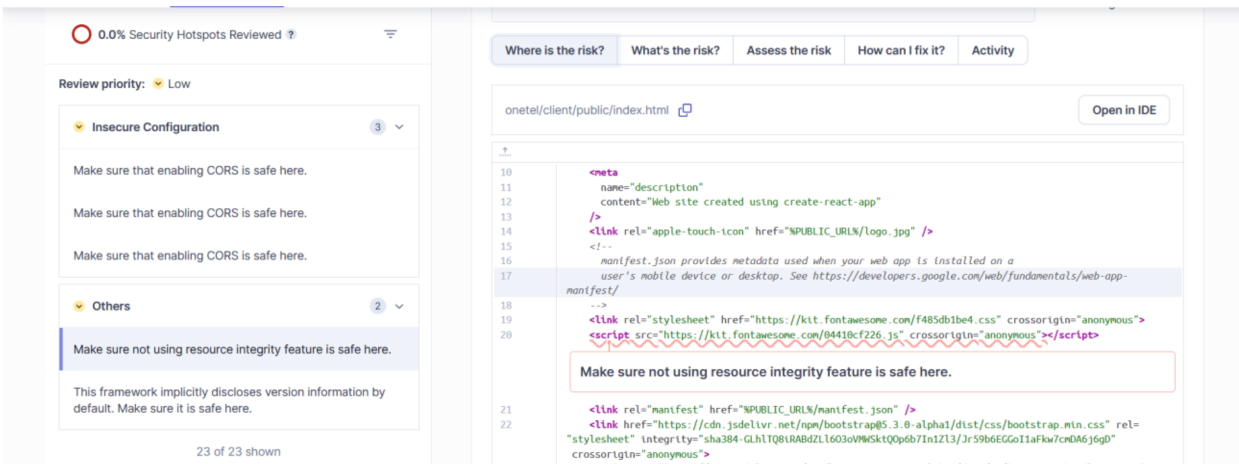
- impersonate users' identities and take advantage of their privileges on the application.
- add quiet malware that monitors users' session and capture sensitive secrets.
- gain access to sensitive clients' personal data.
- deface, or otherwise affect the general availability of the application.
- mine cryptocurrencies in the background.

Likewise, a compromised software piece that would be deployed on a server-side application could badly affect the application’s security. For example, server-side malware could:

- access and modify sensitive technical and business data.
- elevate its privileges on the underlying operating system.
- Use the compromised application as a pivot to attack the local network.

By ensuring that a remote artifact is exactly what it is supposed to be before using it, the application is protected from unexpected changes applied to it before it is downloaded.
Especially, integrity checks will allow for identifying an artifact replaced by malware on the publication website or that was legitimately changed by its author, in a more benign scenario.

Important note: downloading an artifact over HTTPS only protects it while in transit from one host to another. It provides authenticity and integrity checks **for the network stream** only. It does not ensure the authenticity or security of the artifact itself.



Preventing Unverified Remote Resource Inclusion

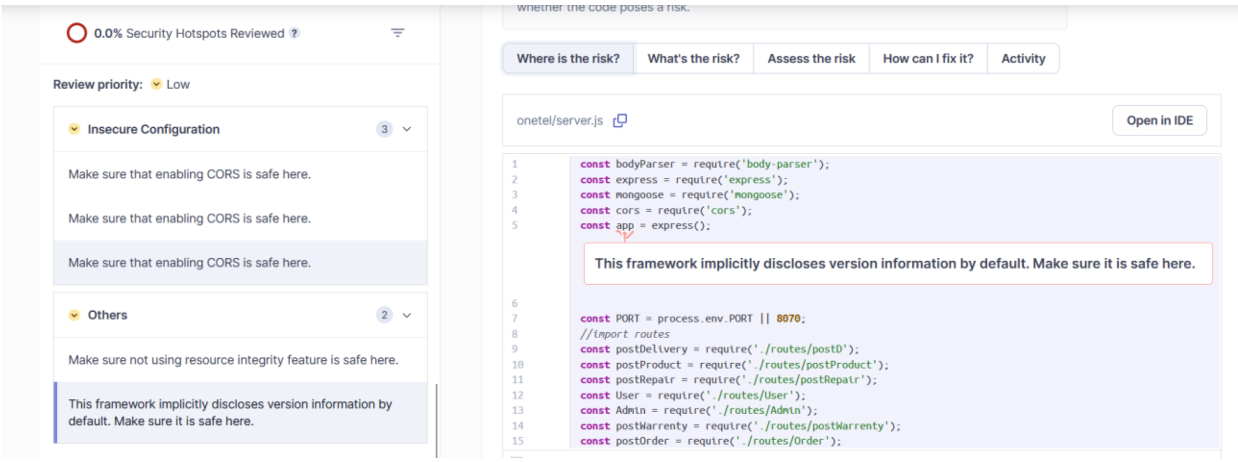
```
<!-- this is issue acording to sonarqube -->
<!-- <script src="https://kit.fontawesome.com/04410cf226.js" crossorigin="anonymous"></script> -->

<!-- fix the isuue according to sonarqube -->
<script src="https://kit.fontawesome.com/04410cf226.js"
integrity="sha384-oqVuAfXRKap7fdgcCY5uykM6+R9GqQ8K/uxy9rx7HNQlGYl1kPzQho1wx4JwY8wC" crossorigin="anonymous"></script>

```

Version Disclosure

In general, it is recommended to keep internal technical information within internal systems to control what attackers know about the underlying architectures. This is known as the "need to know" principle.

The most effective solution is to remove version information disclosure from what end users can see, such as the "x-powered-by" header.
This can be achieved directly through the web application code, server (nginx, apache) or firewalls.

Disabling the server signature provides additional protection by reducing the amount of information available to attackers. Note, however, that this does not provide as much protection as regular updates and patches.
Security by obscurity is the least foolproof solution of all. It should never be the only defense mechanism and should always be combined with other security measures.



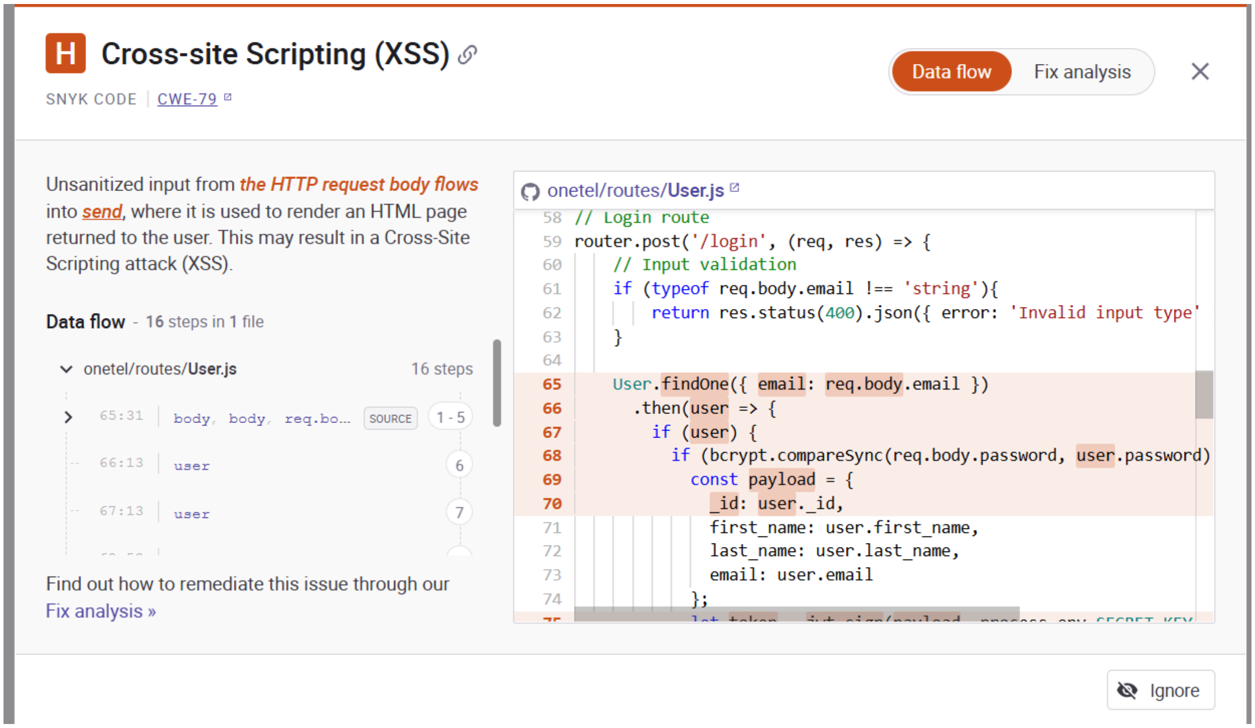
Preventing Version Disclosure :

```
//fix accooprding to sonarqube
const helmet = require('helmet');
const app = express();
```

```
//fix accoriding to sonarqube
app.use(helmet()); // Use Helmet for added security
app.disable("x-powered-by"); // Disable X-Powered-By header
```

Cross-site Scripting (XSS)

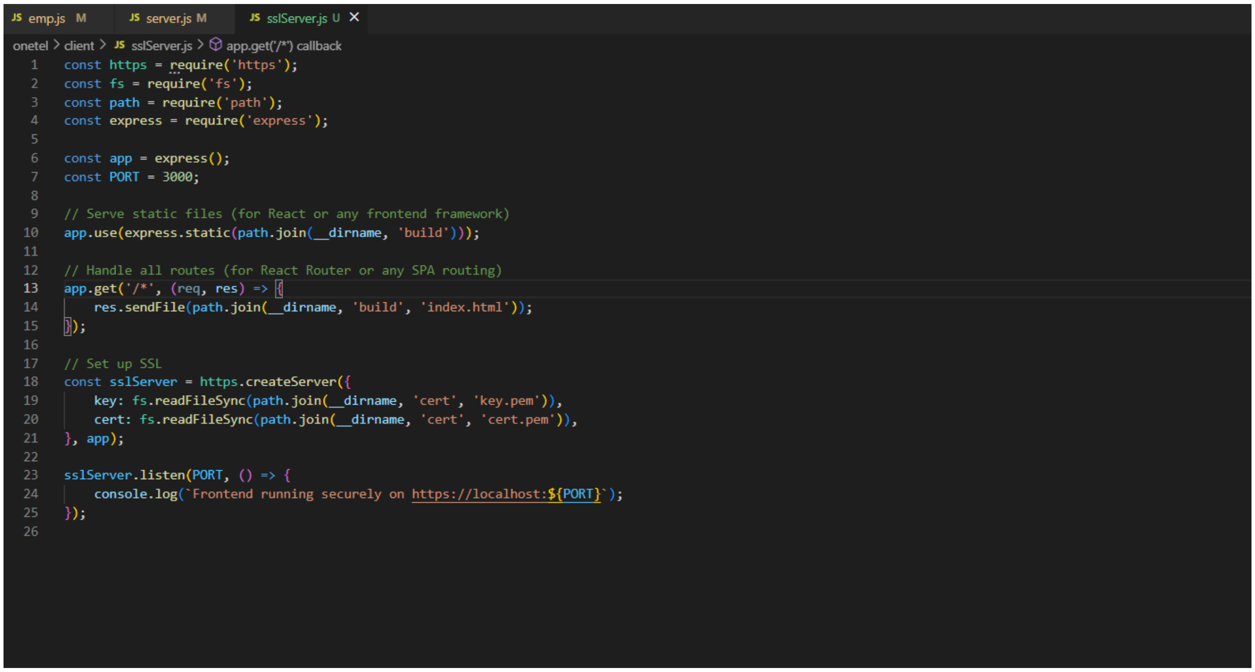
The application is vulnerable to Cross-Site Scripting (XSS) attacks due to the unsanitized input being received from the HTTP request body. Specifically, user inputs, such as email addresses in the login functionality, are processed without adequate validation or sanitization before being reflected back in the application's responses. This oversight can allow an attacker to inject malicious scripts that are executed in the context of the user’s browser. If exploited, such an attack could lead to unauthorized actions on behalf of the user, session hijacking, and exposure of sensitive data.

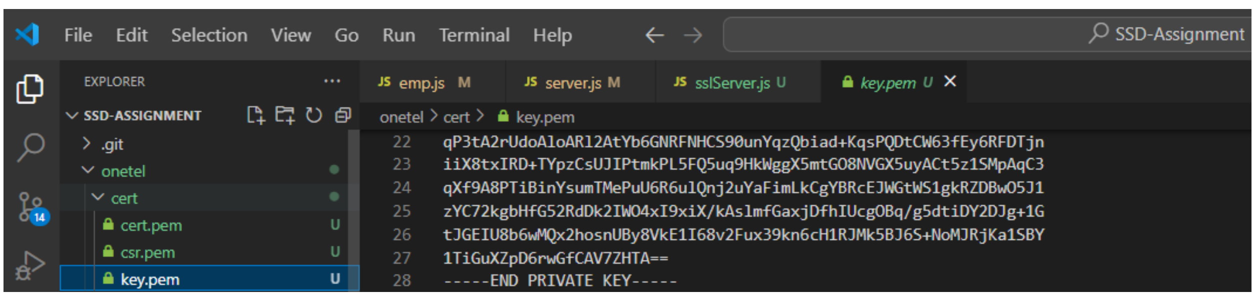


Preventing Cross-site Scripting (XSS)

```javascript
//Fix XSS
// Login route
router.post('/login', (req, res) => {
  // Input validation
  if (typeof req.body.email !== 'string') {
    return res.status(400).json({ error: 'Invalid input type' });
  }

  // Sanitize email input to prevent XSS
  const sanitizedEmail = sanitizeHtml(req.body.email);

  User.findOne({ email: sanitizedEmail })
    .then(user => {
      if (user) {
        if (bcrypt.compareSync(req.body.password, user.password)) {
          const payload = {
            _id: user._id,
            first_name: user.first_name,
            last_name: user.last_name,
            email: user.email
          };
          let token = jwt.sign(payload, process.env.SECRET_KEY, { expiresIn: 1440 });
          res.send(token);
        } else {
          res.status(401).json({ error: 'Invalid password' });
        }
      } else {
        res.status(404).json({ error: 'User not found' });
      }
    })
    .catch(err => {
      res.status(500).json({ error: 'Server error' });
    });
});
```

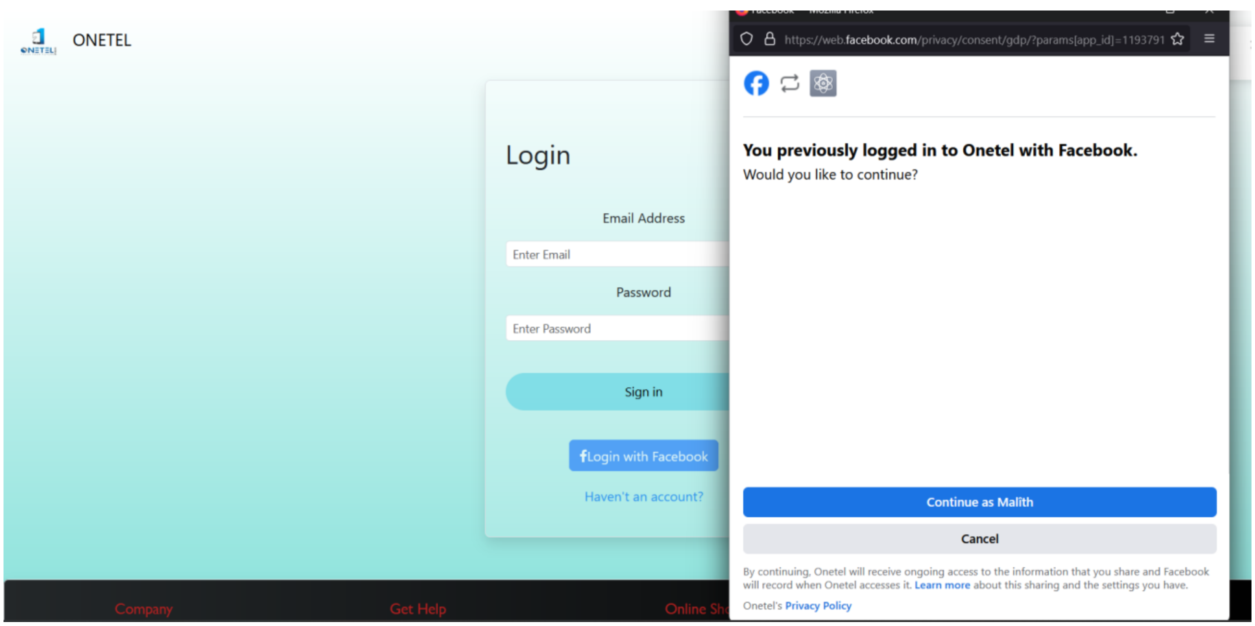
Add SSL Certificate



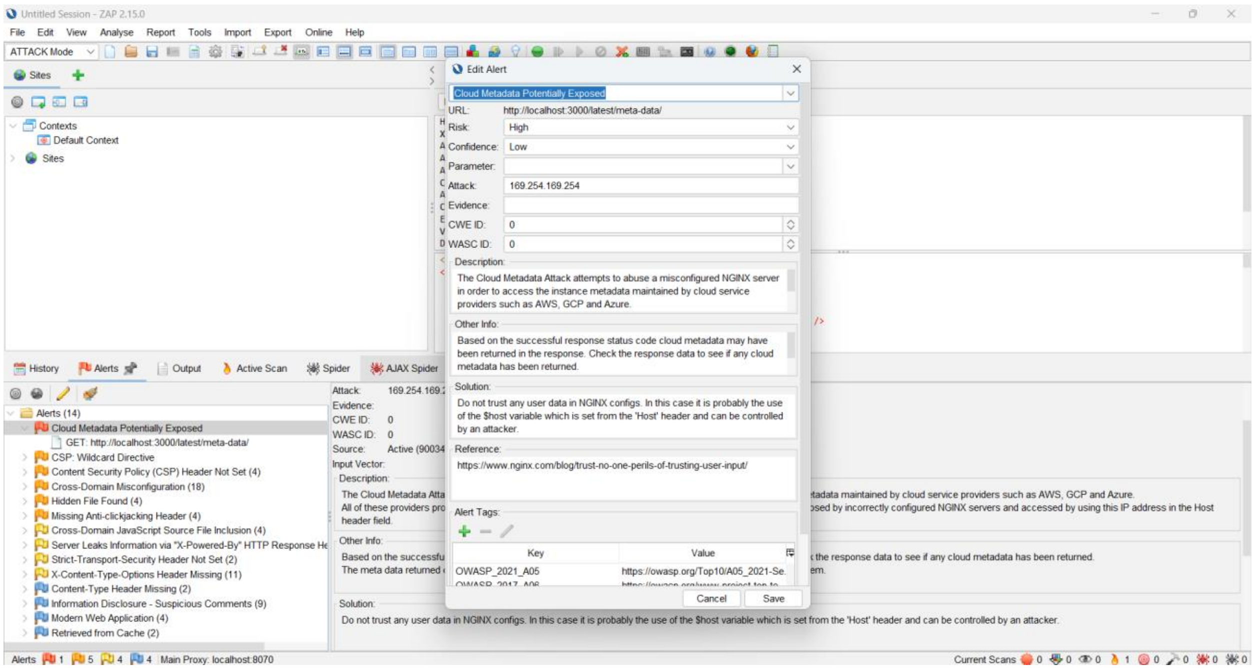


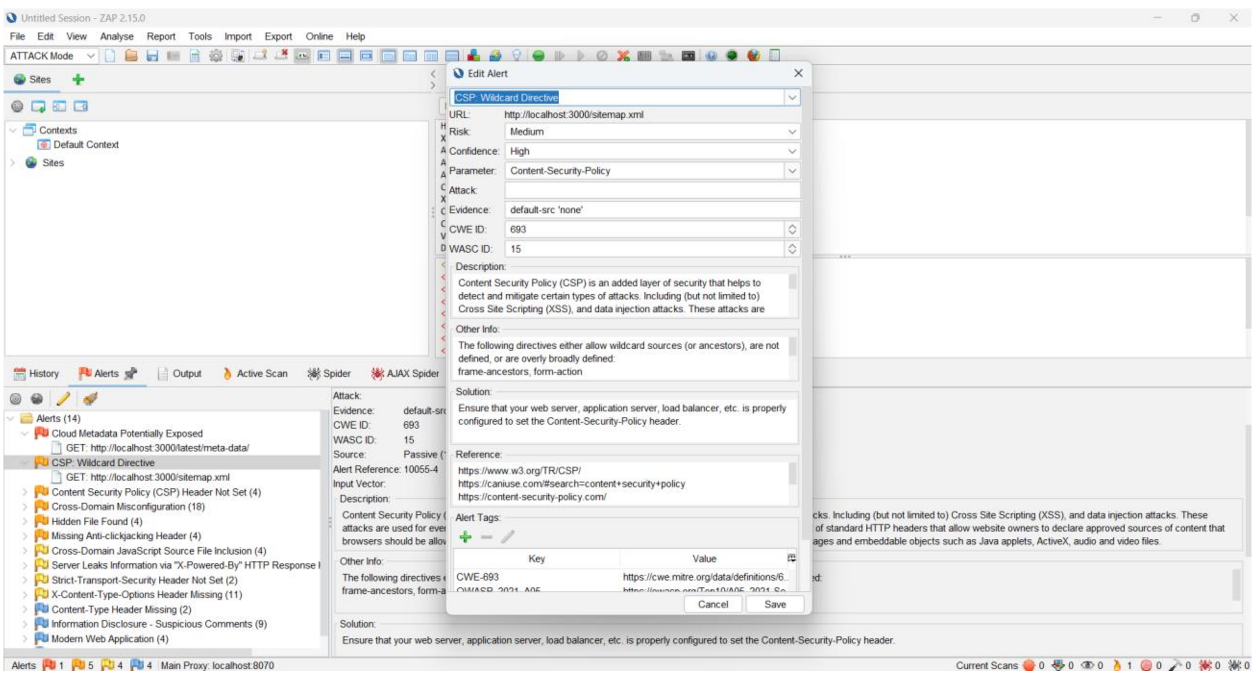
OAuth connect-based grant

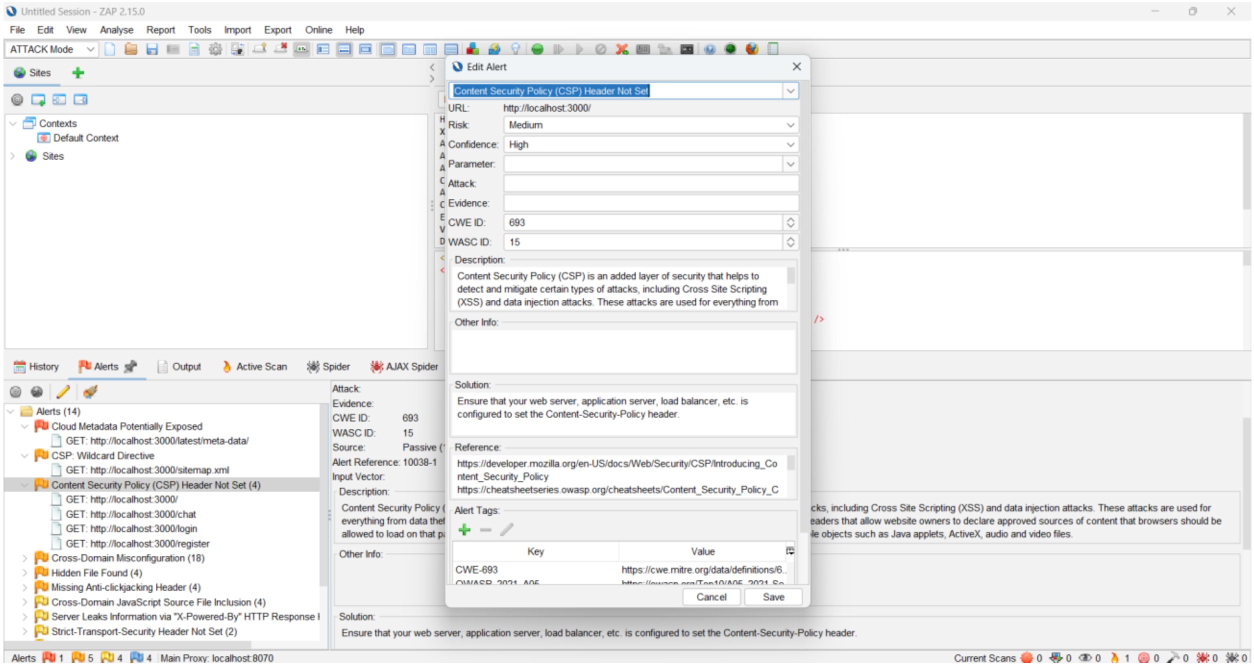
```
        <div className='form-group'>
          <label htmlFor='password'>Password</label><br /><br />
          <input
            type='password'
            className='form-control form-control-sm'
            name='password'
            placeholder='Enter Password'
            value={this.state.password}
            onChange={this.onChange}
          />
        </div><br /><br />
        <button className='btn logbtn'>Sign in</button>
        <br /><br />

        {/* Facebook Login Button */}
        <FacebookLogin
          appId="1193791145246680"
          autoLoad={false}
          fields="name,email,picture"
          callback={this.responseFacebook}
          icon="fa-facebook"
          textButton="Login with Facebook"
          cssClass="btn btn-primary btn-block mt-3"
        />

        <br /><br />
        <Link className="link" to="/register">Haven't an account?</Link>
      </h6>
    </form>
  </div>
```

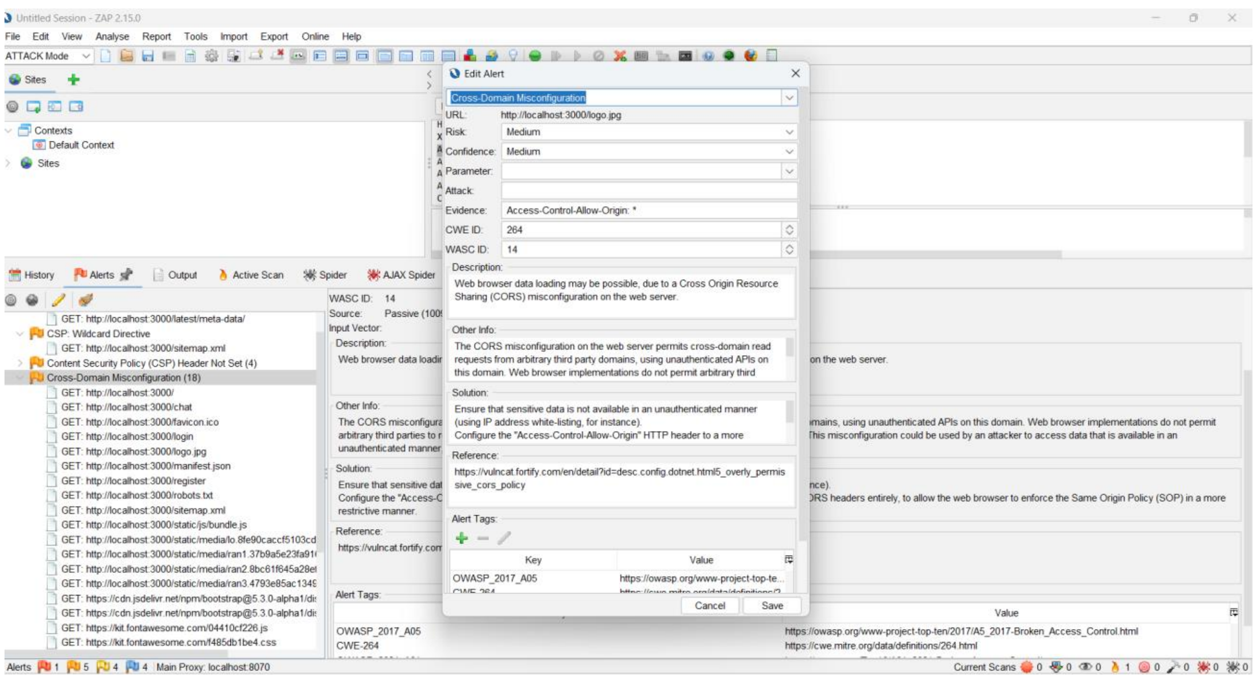


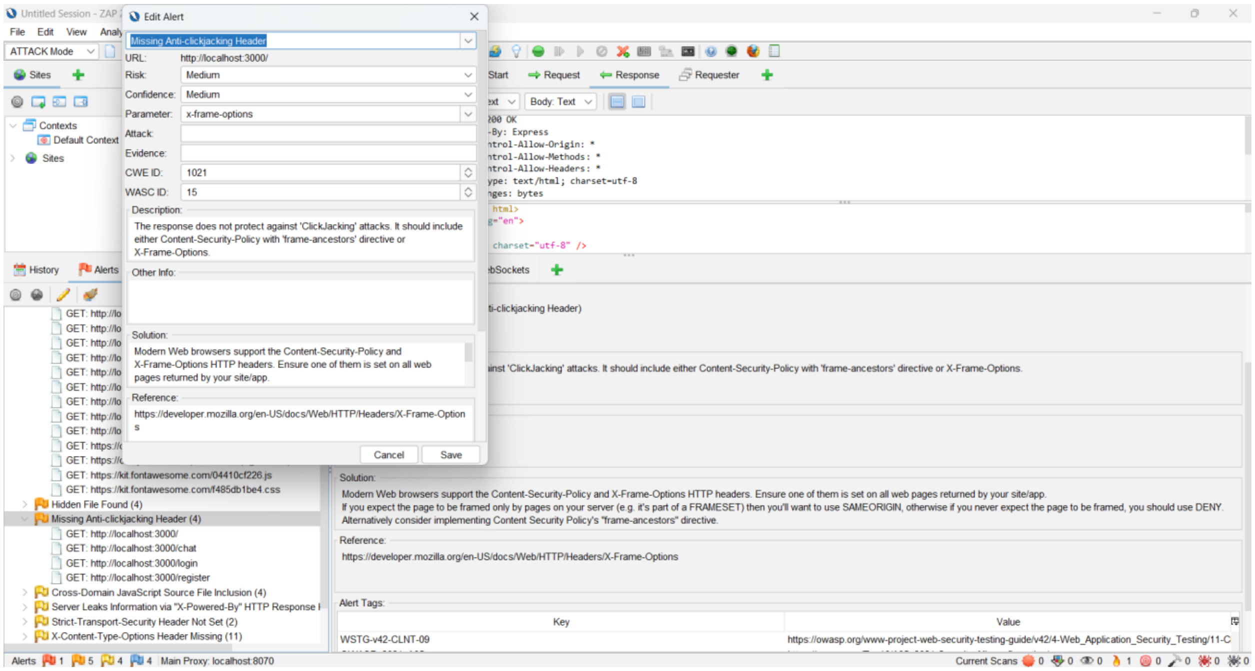
Other vulnerabilities






Untitled Session - ZAP 2.15.0
File Edit View Analyse Report Tools Import Export Online Help
ATTACK Mode
Edit Alert
Content Security Policy (CSP) Header Not Set
URL: http://localhost:3000/
Risk: Medium
Confidence: High
Parameter:
Attack:
Evidence:
CWE ID: 693
WASC ID: 15
Description:
Content Security Policy (CSP) is an added layer of security that helps to detect and mitigate certain types of attacks, including Cross Site Scripting (XSS) and data injection attacks. These attacks are used for everything from
Other Info:
Solution:
Ensure that your web server, application server, load balancer, etc. is configured to set the Content-Security-Policy header.
Reference:
https://developer.mozilla.org/en-US/docs/Web/Security/CSP/Introducing_Content_Security_Policy
https://cheatsheetseries.owasp.org/cheatsheets/Content_Security_Policy_C
Alert Tags:
Key
Value
CWE-693
https://cwe.mitre.org/data/definitions/6...
Cancel
Save
Sites
Contexts
Default Context
Sites
History
Alerts
Output
Active Scan
Spider
AJAX Spider
Alerts (14)
Cloud Metadata Potentially Exposed
GET: http://localhost:3000/latest/meta-data/
CSP: Wildcard Directive
GET: http://localhost:3000/sitemap.xml
Content Security Policy (CSP) Header Not Set (4)
GET: http://localhost:3000/
GET: http://localhost:3000/chat
GET: http://localhost:3000/login
GET: http://localhost:3000/register
Cross-Domain Misconfiguration (18)
Hidden File Found (4)
Missing Anti-clickjacking Header (4)
Cross-Domain JavaScript Source File Inclusion (4)
Server Leaks Information via "X-Powered-By" HTTP Response I
Strict-Transport-Security Header Not Set (2)
Solution:
Ensure that your web server, application server, load balancer, etc. is configured to set the Content-Security-Policy header.
Alerts 1 5 4 4 Main Proxy: localhost:8070
Current Scans 0 0 0 1 0 0 0 0


Untitled Session - ZAP 2.15.0
File Edit View Analyse Report Tools Import Export Online Help
ATTACK Mode
Edit Alert
Cross-Domain Misconfiguration
URL: http://localhost:3000/logo.jpg
Risk: Medium
Confidence: Medium
Parameter:
Attack:
Evidence: Access-Control-Allow-Origin: *
CWE ID: 264
WASC ID: 14
Description:
Web browser data loading may be possible, due to a Cross Origin Resource Sharing (CORS) misconfiguration on the web server.
Other Info:
The CORS misconfiguration on the web server permits cross-domain read requests from arbitrary third party domains, using unauthenticated APIs on this domain. Web browser implementations do not permit arbitrary third
Solution:
Ensure that sensitive data is not available in an unauthenticated manner (using IP address white-listing, for instance).
Configure the "Access-Control-Allow-Origin" HTTP header to a more
Reference:
https://vulncat.fortify.com/en/detail?id=desc.config.dotnet.html5_overly_permissive_cors_policy
Alert Tags:
Key
Value
OWASP_2017_A05
https://owasp.org/www-project-top-te...
Cancel
Save
GET: http://localhost:3000/latest/meta-data/
CSP: Wildcard Directive
GET: http://localhost:3000/sitemap.xml
Content Security Policy (CSP) Header Not Set (4)
Cross-Domain Misconfiguration (18)
GET: http://localhost:3000/
GET: http://localhost:3000/chat
GET: http://localhost:3000/favicon.ico
GET: http://localhost:3000/login
GET: http://localhost:3000/logo.jpg
GET: http://localhost:3000/manifest.json
GET: http://localhost:3000/register
GET: http://localhost:3000/robots.txt
GET: http://localhost:3000/sitemap.xml
GET: http://localhost:3000/static/js/bundle.js
GET: https://kit.fontawesome.com/04410cf226.js
GET: https://kit.fontawesome.com/f485db1be4.css
OWASP_2017_A05
CWE-264
https://owasp.org/www-project-top-ten/2017/A5_2017-Broken_Access_Control.html
https://cwe.mitre.org/data/definitions/264.html
Alerts 1 5 4 4 Main Proxy: localhost:8070



Best Practices

Security-first Design & Development (Security by Design)

Description: Security considerations should be implemented right from the beginning of the SDLC. A security-first mindset supports the goal of making an application resilient against most common vulnerabilities and attacks.

Prevention

- Clearly define security requirements within the design phase.
- Plan on possible attack vectors, such as injection attacks or data breaches, while designing a feature.
- Follow secure coding guidelines and practices to avoid introducing vulnerabilities.

Use of Environment Variables for Sensitive Information

Description: Never hard-code sensitive credentials like API keys, database passwords, and secret keys directly into the source code. They always must be kept in environment variables.

Prevention:

- Use .env files during development and use environment variables during production.
- Ensure that the .env file was added to the .gitignore to avoid committing into version control.
- AWS Secrets Manager, HashiCorp Vault - Secret Management Services: Implement secret management services that securely store, manage, and rotate sensitive information in production.

Code Reviews and Pair Programming

Description: This is reviewing each other's code before merging to the main branch-code reviews. It consists of having two developers collaborating on one task-pair programming. Such activities would help identify practices such as hardcoded secrets, bad error handling, or other security flaws.

How to Avoid It:

Regular code reviews will find hardcoded secrets, poor validation, and other common security mistakes. Pair programming introduces immediate feedback that reduces the likelihood of security issues being missed during the coding phase.

DevSecOps Approach

Description: DevSecOps integrates security practices into DevOps. In this respect, it ensures that security considerations are taken along the whole development and deployment cycle.

How to Avoid:

- Implement code security checks like hardcoded credential scanning in your CI/CD pipeline.
- Automate security testing and vulnerability scanning throughout development, from code commit to deployment.

CI/CD with Security Gates

Description: CI/CD pipeline automates the testing of code and its deployment. Moreover, embedding security checks within the pipeline ensures that most vulnerabilities are detected before the code reaches production.

Prevention:

- Use the security gates part of the CI/CD process that will scan for vulnerabilities like Hard Coded Credentials and Insecure Dependencies.
- Tools like Jenkins, GitLab CI, and CircleCI can integrate with security scanners and SAST tools to catch vulnerabilities well in advance.