Week- 4

Backend Engineering Launchpad

Pawan Panjwani

Airtribe

SECURITY AND PERFORMANCE
Node JS

Airtribe

Agenda

- Difference between encryption and hashing
- Common security risks
- Node JS Security Features
- Package Management, Authenticate and Authorize, Input sanitization, validation
- Error Handling, Logging and security testing
- Measuring performance, Load testing, Profiling and optimization

Question and answers

Add: Encryption vs Hashing (1)

1. ***Encryption*** is a process of converting data into a secret code so that it can only be read by authorized parties who have the decryption key.
2. ***Hashing***, on the other hand, is a one-way process of converting data into a fixed-length string of characters, where the output cannot be reversed back to the original input.

Add: Encryption vs Hashing (2)

1. ***Encryption*** can be reversed. Encryption is used to protect data that needs to be decrypted later, such as passwords or sensitive data. In Node.js, encryption can be achieved using various algorithms, such as AES or RSA
2. ***Hashing*** is a one way process .hashing is used to verify the integrity of data or to store passwords securely. hashing can be achieved using algorithms such as SHA-256 or bcrypt.

Common security risks – Node JS (1)

- Injection attacks, such as SQL injection and command injection
- Cross-site scripting (XSS) attacks
- Insecure authentication and authorization mechanisms
- Cross-site request forgery (CSRF) attacks
- Insecure handling of sensitive data, such as passwords and API keys
- Vulnerable third-party dependencies and packages

Common security risks – Node JS (2)

- Insufficient input validation and sanitization
- Improper error handling and logging
- Lack of security testing and monitoring
- Insufficient encryption and data protection techniques

Let’s start with SQL/Command Injection Attack

SQL Injection and command injection

```
1  app.post("/records", (request, response) => {
2    const data = request.body;
3    const query = `SELECT * FROM users WHERE id = (${data.id})`;
4    connection.query(query, (err, rows) => {
5      if(err) throw err;
6      response.json({data:rows});
7    });
8  });
```

Can you spot issues with this piece of code ? How is this code vulnerable to SQL Injection and command injection attack ?

Airtribe

Common SQL Injection attacks

```
1) Always true (or 1=1) attacks

  - SELECT * FROM Users WHERE UserId = 105 OR 1=1;

2) Query stacking attacks

  - SELECT * FROM products WHERE id = 10; DROP members--

3) Data exfiltration (or query comment) attacks

  SELECT * FROM health_records WHERE date = '22/04/1999'; -- ' AND id = 33
```

Common ways in which attackers can penetrate our databases and perform malicious commands

Airtribe

Preventive Measures for SQL Injection

1) Input sanitization and masking

```
function validateForm() {
  let x = document.forms["form"]["email"].value;
  const re = /^(([^<>()[\]\\.,;:\s@"]+(\.[^<>()[\]\\.,;:\s@"]+)*)|(".+"))@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9

  if (x == "") {
    alert("Email must be filled out");

    return false;
  } else if (re.test(String(x).toLowerCase()) == false) {
    alert("Email must be valid");

    return false;
  }
  return true;
}
```

Validate user input to ensure it meets specific criteria, such as length and format. Also, handle errors gracefully and provide helpful error messages to prevent attackers from exploiting vulnerabilities.

Airtribe

Preventive Measures for SQL Injection

1) Secure Data Layer - Use Parameterized querying

```
app.post("/records", (request, response) => {
  const data = request.body;
  connection.query('SELECT * FROM health_records where id = ?', [data.id], (err, rows) => {
    if(err) throw err;
    response.json({data:rows});
  });
});
```

Parameterized queries are a safe way to pass user input to SQL statements. They allow you to separate user input from SQL code, making it harder for attackers to inject malicious code.

Airtribe

Is that it ? Are we safe ?

```
app.post("/auth", function (request, response) {
  var username = request.body.username;
  var password = request.body.password;
  if (username && password) {
    connection.query(
      "SELECT * FROM accounts WHERE username = ? AND password = ?",
      [username, password],
      function (error, results, fields) {
        //do something
      }
    );
  }
});
```

This code seems secure, help identify if it would have any issues ?

Airtribe

Lets Evaluate

```
body: "username=admin&password[password]=1"

//In this example, the attacker is passing in a
//value that gets evaluated as an Object instead of a String value, and results in the following SQL query:

// SELECT * FROM accounts WHERE username = 'admin' AND password = `password` = 1
// The password = `password` = 1 part evaluates to TRUE and is a 1=1 attack.
```

Airtribe

No escape to sanitization and type checking

```
app.post("/auth", function (request, response) {
  var username = request.body.username;
  var password = request.body.password;
  // Reject different type
  if (typeof username != "string" || typeof password != "string"){
   response.send("Invalid parameters!");
   response.end();
   return;
  }
  if (username && password) {
   connection.query(
    "SELECT * FROM accounts WHERE username = ? AND password = ?",
    [username, password],
    function (error, results, fields) {
     // do something
    }
   );
  }
});
```

Airtribe

Let’s take a look at third party dependency vulnerability

Third party dependency vulnerability (1)

- **The cross-site scripting (XSS) vulnerability in serialize-javascript:** In 2018, a popular package called serialize-javascript was found to have a vulnerability that allowed attackers to inject malicious code into a website, leading to cross-site scripting (XSS) attacks. The vulnerability existed in version 1.1.1 and earlier, affecting a large number of Node.js applications.

Third party dependency vulnerability (2)

- **The event-stream attack: In 2018, the event-stream package**, which was used as a dependency in many Node.js applications, was compromised by a malicious actor. The attacker added malicious code to the package, which was designed to steal cryptocurrency from users who were using a specific wallet. This vulnerability affected not only the event-stream package but also all the applications that were using it as a dependency.

Third party dependency vulnerability (3)

- **The npm vulnerability in tar: In 2018**, a vulnerability was discovered in the tar package, which is used by the npm package manager to extract and compress packages. The vulnerability allowed an attacker to execute arbitrary code on a user's system by creating a malicious package with a specially crafted filename. The vulnerability existed in versions 2.2.1 to 4.4.2 of the tar package, affecting a large number of Node.js applications that were using the npm package manager.

Preventive Measures – Dependency vulnerability (1)

1. **Regularly update dependencies:** Keep your dependencies up-to-date to ensure that you are using the latest and most secure versions of packages.
2. **Use reputable packages:** Use packages from reputable sources and maintainers with a good track record.
3. **Limit the number of dependencies:** Minimize the number of dependencies you use in your application.

Preventive Measures – Dependency vulnerability (2)

1. **Monitor for vulnerabilities:** Use security tools like **npm audit** and **snyk** to automatically detect and fix vulnerabilities in your dependencies.
2. **Create a vulnerability management process:** Develop a process for managing vulnerabilities in your application's dependencies.

Let’s take a look at weak authentication and authorization vulnerability

Authentication and authorization vulnerability (1)

- **The Equifax breach:** In 2017, Equifax, one of the largest credit reporting agencies in the United States, suffered a massive data breach that exposed the personal information of 143 million consumers. The breach was caused by a vulnerability in Apache Struts, a framework used by Equifax's web applications, which allowed attackers to gain access to the sensitive data. One of the contributing factors was the use of weak authentication and authorization mechanisms in the Equifax web applications.

Authentication and authorization vulnerability (2)

- **The MongoDB ransomware attacks:** In 2017, a wave of attacks targeted insecure MongoDB databases that were exposed to the internet without proper authentication and authorization mechanisms. Attackers would scan the internet for insecure MongoDB instances and then encrypt the data stored in those databases, demanding a ransom to decrypt it. This attack highlighted the importance of properly securing databases with strong authentication and authorization mechanisms.

Authentication and authorization vulnerability (3)

- **The Twitter OAuth vulnerability:** In 2013, a vulnerability was discovered in Twitter's OAuth implementation that allowed attackers to hijack user accounts. The vulnerability was caused by a flaw in the authentication flow that allowed attackers to bypass the authorization step and gain access to user accounts. This vulnerability was caused by weak authentication and authorization mechanisms and highlighted the importance of properly securing user authentication and authorization in web applications.

Preventive Measures – Authentication and Authorization (1)

1. **Use secure authentication protocols:** Use secure authentication protocols like OAuth 2.0, OpenID Connect, and JSON Web Tokens (JWTs) to authenticate users. These protocols provide strong authentication mechanisms that are resistant to attacks like brute force, phishing, and session hijacking.
2. **Use strong passwords:** Enforce strong password policies that require users to create strong passwords and change them regularly. Use password hashing algorithms like bcrypt to securely store passwords.

Preventive Measures – Authentication and Authorization (2)

1. **Implement two-factor authentication:** Implement two-factor authentication (2FA) to provide an additional layer of security for user accounts. 2FA requires users to provide two forms of authentication, such as a password and a security token, to access their accounts.
2. **Limit access to resources:** Use access control mechanisms to limit access to sensitive resources. Only allow authenticated and authorized users to access resources that they need to perform their tasks.

Preventive Measures – Authentication and Authorization (3)

1. **Enforce role-based access control:** Implement role-based access control (RBAC) to control access to resources based on user roles. This ensures that users only have access to resources that they are authorized to access.
2. **Audit and monitor authentication and authorization events:** Implement logging and monitoring mechanisms to track authentication and authorization events. This helps detect and respond to security incidents in real-time.

Let’s take a look at cross site scripting XSS attacks

What is cross site scripting attack ?

Cross-site scripting (XSS) attacks are a type of web application security vulnerability that allows an attacker to inject malicious code into a web page viewed by other users. The attack occurs when an attacker is able to inject malicious code, typically in the form of a script, into a web page viewed by other users. This can happen through vulnerable input fields, such as search fields, contact forms, or comments sections, that don't properly sanitize user input.

XSS Vulnerabilities (1)

- **Reflected XSS:** A Node.js application that uses query parameters in the URL without proper input validation and sanitization is vulnerable to reflected XSS attacks. For example, if the application displays a search query in the search results page without sanitizing the input, an attacker could inject a malicious script in the search query and execute it in the victim's browser.

XSS Vulnerabilities (2)

- **Stored XSS:** A Node.js application that allows users to post comments, messages or other user-generated content without proper input validation and sanitization is vulnerable to stored XSS attacks. For example, if the application allows users to post messages in a chat room without sanitizing the input, an attacker could inject a malicious script in a message and execute it in the browsers of all users who view that message.

XSS Vulnerabilities (3)

- **DOM-based XSS:** A Node.js application that uses client-side JavaScript to manipulate the DOM (Document Object Model) without proper input validation and sanitization is vulnerable to DOM-based XSS attacks. For example, if the application allows users to submit their names and displays a welcome message that includes the user's name without sanitizing the input, an attacker could inject a malicious script in their name and execute it in the victim's browser.

Let’s take a look at Xss Vulnerable code

```
1  // vulnerable route that displays user input without sanitization
2  app.get('/search', (req, res) => {
3    const query = req.query.q;
4    res.send(`<p>Search results for: ${query}</p>`);
5  });
```

In this example, the **get** route allows the user to input a search query via the **q** parameter. However, the input is not validated or sanitized, which leaves the application vulnerable to an XSS attack. An attacker could inject a malicious script in the search query, which would then be executed in the victim's browser when the search results page is displayed.

Lets avert the vulnerability

```
1  const xss = require('xss-clean');
2
3  // route that sanitizes user input before displaying it
4  app.get('/search', xss(), (req, res) => {
5    const query = req.query.q;
6    res.send(`<p>Search results for: ${query}</p>`);
7  });
```

In this updated example, we've added the xss-clean middleware to the get route. This middleware sanitizes user input before it's rendered in the response, which helps prevent XSS vulnerabilities in the application.

Airtribe

Preventive Measures – XSS (1)

1. **Validate and sanitize user input:** Use libraries like helmet, xss-clean, and sanitize-html to sanitize user input and prevent XSS vulnerabilities.

1. **Use Content Security Policy (CSP) headers:** Implementing a content security policy (CSP) header can help limit the sources of content that are allowed to be loaded on a page and limit the types of scripts that can be executed.

Preventive Measures – XSS (2)

1. **Use HTTP-only cookies:** HTTP-only cookies can help prevent cross-site scripting attacks by preventing client-side JavaScript from accessing cookie data.

1. **Use a security-focused Node.js framework:** Use a security-focused Node.js framework like Helmet or Express.js to help protect against XSS attacks.

Preventive Measures – XSS (3)

1. **Regular security audits and testing:** Regular security audits and testing can help identify and mitigate vulnerabilities that could be exploited by XSS attacks. Use automated security testing tools, perform manual security audits, and keep up-to-date with the latest security best practices.

Security Testing

Security Testing Methods

- **Penetration Testing:** Penetration testing, also known as pen testing, is a method of evaluating the security of a Node.js application by attempting to exploit vulnerabilities and weaknesses in the system.
- **Vulnerability Scanning:** Vulnerability scanning is a process of scanning the Node.js application and its dependencies for known security vulnerabilities.
- **Code Review:** Code review is a process of reviewing the source code of a Node.js application to identify potential security vulnerabilities.

Application performance

Performance metrics (1)

1) Response Time: Response time is a key performance metric for Node.js applications, as it measures how quickly the application responds to requests from clients.
2) Throughput: Throughput is a measure of the number of requests that the Node.js application can handle per unit of time.
3) Memory Usage: Memory usage is an important performance metric for Node.js applications, as Node.js runs on a single thread and has a limited amount of memory available.

Performance metrics (2)

1) CPU Usage: CPU usage is another important performance metric for Node.js applications, as Node.js is a CPU-bound application and can become a bottleneck if CPU usage is too high
2) Error Rates: Error rates are a measure of the number of errors that occur in the Node.js application per unit of time

Measuring the performance metrics

- Profiling
- Load Testing
- Code Reviews
- Monitoring
- Benchmarking
- Logging

Thank You!

Airtribe