Node.js

Top 11 Node.js security best practices

Dawid Ziolkowski

January 5, 2021 · 8 minute read · No comments

Node.js is extremely popular nowadays, primarily as a backend server for web applications. However, in the world of microservices, you can find it pretty much everywhere, playing different and important roles in a bigger application stack.

One of the advantages of Node.js is the ability to install additional modules, which from the security point of view, provides more opportunities to open back doors. Additionally, the more popular the framework, the more chances that hackers will try to find vulnerabilities. Therefore, you should always take Node.js security seriously. In this post, you'll learn the 11 best practices for securing your Node.js application.

1. Validate user input to limit SQL injections and XSS attacks

Let's start with one of the most popular attacks, the SQL Injection. As the name suggests, a SQL injection attack happens when a hacker is able to execute SQL statements on your database. This becomes possible when you don't sanitize the input from the frontend. In other words, if your Node.js backend takes the parameter from the user-provided data and uses it directly as a part of the SQL statement. For example:

```
connection.query('SELECT * FROM orders WHERE id = ' + id, function (error, results, fields) {
  if (error) throw error;
  // ...
});
```

node-security2.md hosted with ❤ by GitHub view raw

The above query is SQL injection vulnerable. Why? Because the `id` parameter is taken directly from the frontend. Instead of sending just the `id`, the attacker can manipulate the request and send SQL commands with it. Instead of sending just `4564` (the id of the order), the attacker can send `4564; DROP TABLE ORDERS;` and Node.js will wipe your database. Bummer!

How do you avoid that? There are a few ways, but the basic idea is to not blindly pass parameters from the frontend to the database query. Instead, you need to validate or escape values provided by the user. How to do it exactly depends on the database you use and the way you prefer to do it. Some database libraries for Node.js perform escaping automatically (for example **node-mysql** and **mongoose**). But you can also use more generic libraries like **Sequelize** or **knex**.

SEARCH

Input your search Search

RECENT POSTS

Democratizing security: The next step in Sqreen's journey

Heroku Security: Securing your Heroku application

Streaming data with Amazon Kinesis

What is a Content Security Policy (CSP) and why is it important?

CIS 20 overview and what not to miss

CATEGORIES

Dev (56)

DevOps (14)

Go (5)

Java (4)

JavaScript (7)

Node.js (15)

PHP (3)

Python (18)

Ruby (16)

Security (136)

Sqreen Product (35)

XSS attacks

Cross-Site Scripting (XSS) attacks work similarly to SQL injections. The difference is that instead of sending malicious SQL, the attacker is able to execute JavaScript code. The reason for that is the same as before, not validating input from the user.

```
app.get('/find_product', (req, res) => {
  ...
  if (products.length === 0) {
    return res.send('<p>No products found for "' + req.query.product + '"</p>');
  }
  ...
});
```

node-security.md hosted with ❤ by GitHub view raw

As you can see in the snippet above, whatever the user puts in the search field, if not found in the database, will be sent back to the user in an unchanged form. What that means is that if an attacker puts JavaScript code instead of the product name in your search bar, the same JavaScript code will be executed.

How do you fix that? Again, validate the user input! You can use **validatorjs** or **xss-filters** for that.

2. Implement strong authentication

Having a broken, weak, or incomplete authentication mechanism is ranked as the second most common vulnerability. It's probably due to the fact that many developers think about authentication as "we have it, so we're secure." In reality, weak or inconsistent authentication is easy to bypass. One solution is to use existing authentication solutions like Okta or OAuth.

If you prefer to stick with native Node.js authentication solutions, you need to remember a few things. When creating passwords, don't use the Node.js built-in crypto library; use **Bcrypt** or **Scrypt**. Make sure to limit failed login attempts, and don't tell the user if it's the username or password that is incorrect. Instead, return a generic "incorrect credentials" error. You also need proper session management policies. And be sure to implement 2FA authentication. If done properly, it can increase the security of your application drastically. You can do it with modules like **node-2fa** or **speakeasy**.

3. Avoid errors that reveal too much

Next on the list is error handling. There are a few things to consider here. First, don't let the user know the details, i.e., don't return the full error object to the client. It can contain information that you don't want to expose, such as paths, another library in use, or perhaps even secrets. Second, wrap routes with the catch clause and don't let Node.js crash when the error was triggered from a request. This prevents attackers from finding malicious requests that will crash your application and sending them over and over again, making your application crash constantly.

Speaking of flooding your Node.js app with malicious requests, don't directly expose your Node.js app to the Internet. Use some component in front of it, such as a load balancer, a cloud firewall or gateway, or old good nginx. This will allow you to rate limit DoS attacks one step before they hit your Node.js app.

4. Run automatic vulnerability scanning

So far I described a few obvious must-dos. The Node.js ecosystem, however, consists of many different modules and libraries that you can install. It's very common to use a lot of them in your projects. This creates a security issue; when using code written by someone else, you can't be 100 percent sure that it's secure. To help with that, you should run frequent automated vulnerability scans. They help you find dependencies with known vulnerabilities. You can use **npm audit** for the basic check, but consider using one of the tools described here.

5. Avoid data leaks

Remember what we said before about not trusting the frontend? You not only shouldn't trust what comes from the frontend but also what you are sending to it. It's easier to send all data for a particular object to the frontend and only filter what to show there. However, for an attacker, it's very easy to get the hidden data sent from the backend.

For example, imagine you want to show a list of the users who signed up for an event. You execute a SQL query to get all users for that particular event and send that data to the frontend, and there you filter it to only show the first and last name. But all the data you don't want to show (like users' birth dates, phone numbers, email addresses, etc.) is easily accessible via the browser developer console. This leads to data leaks.

How do you solve it? Only send the data that's required. If you only need first and last names, retrieve only those from the database. This creates a little bit more work, but it's definitely worth it.

6. Set up logging and monitoring

You may think that logging and monitoring, while important, aren't really related to security, but that isn't true. Of course, the goal is to make systems secure from the beginning, but in reality, it requires an ongoing process. And for that, you need logging and monitoring. Some hackers may be interested in making your application unavailable, which you can find out without logging. But some hackers will prefer to remain undetected for a longer period of time. For such cases, monitoring logs and metrics will help you spot that something is wrong. With only basic logging, you won't get enough information to understand if weird-looking requests are coming from your own application, a third-party API, or from a hacker.

7. Use security linters

We talked about automatic vulnerability scanning before, but you can go one step further and catch common security vulnerabilities even while writing the code. How? By using linter plugins like **eslint-plugin-security**. A security linter will notify you every time you use unsafe code practices (for example using eval or non-literal regex expressions).

8. Avoid secrets in config files

Writing secure code from the beginning will definitely help, but it won't make your application bulletproof if you end up storing plain text secrets in your config files. This practice is unacceptable even if you store the code in a private repository. Importing secrets from environment variables is the first step, but it's not a perfect solution either. To be more confident that your secrets aren't easily readable, use secret management solutions like **Vault.** Whenever using Vault isn't possible, encrypt your secrets when you store them and be sure to rotate them on a regular basis. Many CI/CD solutions allow you to securely store secrets and securely deploy them.

9. Implement HTTP response headers

Many less common attacks can be avoided by adding additional security-related HTTP headers to your application. The most basic mechanisms like CORS will improve the security of your API, but consider using modules like **helmet**, which will add even more headers in order to secure your application. Helmet can implement eleven different header-based security mechanisms for you with one line of code:

```
app.use(helmet());
```

If you want to know the details, here's the full list.

10. Don’t run Node.js as root

In the world of Docker and microservices, we often forget about how Node.js is actually executed. It’s easy to just spin up a Docker container and assume it’s isolated from the host machine so it’s secure. But using Docker doesn’t mean that running Node.js as root is not a problem anymore. Combine the ability to run any JavaScript code via an XSS attack with Node.js running as root and you’ll end up with unlimited hacking capabilities.

11. Protect and observe your Node.js apps in production

Shifting left and securing your Node.js applications early is good practice, but with evolving and changing environments, it’s nearly impossible to fully secure everything in your app. That’s why shifting right and protecting and monitoring your applications in production is important as well. Application Security Management tools can help you get visibility into your Node.js apps, and detect and block attacks in real time, ensuring that attackers aren’t getting around your secure coding efforts.

Summary

Securing web applications is important, but tight deadlines sometimes prevent us from properly executing at any given stage. That’s why it’s important to consider security at every step of the software development lifecycle, from conception all the way to production.

We only covered the most important best practices here, and if you want to learn more, you can download The Node.js Security Handbook.

—-

This post was written by Dawid Ziolkowski. Dawid has 10 years of experience as a Network/System Engineer at the beginning, DevOps in between, Cloud Native Engineer recently. He’s worked for an IT outsourcing company, a research institute, telco, a hosting company, and a consultancy company, so he’s gathered a lot of knowledge from different perspectives. Nowadays he’s helping companies move to cloud and/or redesign their infrastructure for a more Cloud Native approach.

Dawid Ziolkowski

RELATED TAGS

best practices, **Node.js**, **SQL injections**, **XSS**

Subscribe

Be the First to Comment!

0 COMMENTS

By the Sqreen Team ❤️