ATTACKING NODE.JS APPLICATIONS

WITH SECURITY CODING PRACTICE

WWW.DEVSECOPSGUIDES.COM

Attacking NodeJS Application

· Apr 22, 2024 · 22 min read

Table of contents

References

When it comes to securing Node.js applications, understanding potential attack vectors is paramount. Node.js, with its asynchronous and event-driven architecture, offers great performance and scalability, but it also introduces unique security challenges. One common threat is injection attacks, where malicious code is injected into the application to manipulate its behavior or access sensitive data. This can include SQL injection, where attackers exploit vulnerabilities in database queries, or code injection, where they inject malicious code into input fields or parameters.

To mitigate these risks, adhering to security best practices is essential. First and foremost, input validation and sanitization are crucial. Never trust user input and always validate and sanitize data to prevent injection attacks. Utilize parameterized queries for database access to minimize the risk of SQL injection. Additionally, implement proper authentication and authorization mechanisms to control access to sensitive resources. Regularly update dependencies to patch any known vulnerabilities and utilize security tools such as static code analysis and vulnerability scanners to identify and address potential weaknesses in the codebase. By following these practices, developers can strengthen the security posture of their Node.js applications and reduce the risk of successful attacks.

Use flat Promise chains

To improve the readability and maintainability of asynchronous code in Node.js, it's essential to avoid the "Pyramid of Doom" or "Callback Hell" by utilizing Promises and async/await. Promises offer a cleaner approach to handling asynchronous operations, allowing for a flat chain of execution and easier error handling.

Consider the following code snippet illustrating "Callback Hell":

```
function func1(nam
  // operations that take time and then call the callback
}
// func2, func3, and func4 defined similarly...

func1("input1", function(err, result1) {
   if (err) {
      // error operations
   } else {
      // operations
      func2("input2", function(err, result2) {
         if (err) {
            // error operations
         } else {
            // operations
            func3("input3", function(err, result3) {
               if (err) {
                  // error operations
               } else {
                  // operations
                  func4("input4", function(err, result4) {
                     if (err) {
                        // error operations
                     } else {
                        // operations
                     }
                  });
               }
            });
         }
      });
   }
});
```

To refactor this code using a flat Promise chain:

```
function func1(nam
  // operations that take time and then resolve the promise
}
// func2, func3, and func4 defined similarly...

func1("input1")
   .then(function(result) {
      return func2("input2");
   })
   .then(function(result) {
      return func3("input3");
   })
   .then(function(result) {
      return func4("input4");
   })
   .catch(function(error) {
      // error operations
   });
```

Or utilizing async/await:

```
async function func1(name) {
  // operations that take time and then resolve the promise
}
// func2, func3, and func4 defined similarly...

(async () => {
  try {
    let res1 = await func1("input1");
    let res2 = await func2("input2");
    let res3 = await func3("input3");
    let res4 = await func4("input4");
  } catch(err) {
    // error operat
  }
})();
```

By adopting flat Promise chains or async/await syntax, the code becomes more readable, maintainable, and less prone to errors, providing a cleaner solution for handling asynchronous operations in Node.js applications.

Set request size limits

To safeguard your Node.js application from resource-intensive tasks and potential attacks leveraging large request bodies, it's crucial to set limits on request sizes. Without such limits, attackers can exploit vulnerabilities by flooding the server with oversized requests, leading to memory exhaustion or disk space filling. One effective approach to enforce request size limits is by utilizing the raw-body module, which allows you to restrict the size of request bodies for specific HTTP methods such as POST, PUT, and DELETE.

COPY

```
const contentType = require('content-type');
const express = require('express');
const getRawBody = require('raw-body');

const app = express();

app.use(function (req, res, next) {
  if (!['POST', 'PUT', 'DELETE'].includes(req.method)) {
    next();
    return;
  }

  getRawBody(req, {
    length: req.headers['content-length'],
    limit: '1kb',
    encoding: contentType.parse(req).parameters.charset
  }, function (er
    if (err) return next(err);
    req.text = string;
    next();
  });
});
```

However, applying a uniform request size limit across all requests might not be suitable for scenarios where certain requests legitimately require larger payloads, such as file uploads. Moreover, handling JSON input is more resource-intensive than multipart input due to blocking operations during parsing. Therefore, it's advisable to set different request size limits based on content types. This can be achieved conveniently using Express middleware:

```
app.use(express.urlencoded({ extended: true, limit: "1kb" }));
app.use(express.json({ limit: "1kb" }));
```

It's important to acknowledge that attackers may attempt to evade request size limits by modifying the Content-Type header. Therefore, validating the request data against the declared content type in the headers is essential before processing the request. If strict content type validation adversely impacts performance, consider validating only specific content types or requests exceeding a predefined size to strike a balance between security and efficiency.

Do not block the event loop

In Node.js, maximizing performance and maintaining responsiveness relies on adhering to its event-driven, non-blocking I/O architecture. This architecture enables high throughput and simplifies programming by eliminating the need for threads. However, blocking operations can disrupt this flow, hindering the event loop and potentially degrading ap issue, it's crucial to perform all blocking op ne event loop remains unblocked.

```
const fs = require('fs');

// Perform blocking operation asynchronously
fs.readFile('/file.txt', (err, data) => {
  // Perform actions on file content
});

// This synchronous operation can disrupt the event loop if /file.txt is large
fs.unlinkSync('/file.txt');
```

Even when asynchronous operations are employed, race conditions can still arise if subsequent code relies on the completion of asynchronous tasks. For instance, consider the scenario where file deletion occurs before processing the file content, leading to unexpected behavior. To prevent such race conditions and maintain the desired sequence of operations, it's essential to encapsulate dependent operations within the same callback.

```
const fs = require('fs');

// Perform file read and subsequent actions in the same callback to maintain order
fs.readFile('/file.txt', (err, data) => {
  // Perform actions on file content
  fs.unlink('/file.txt', (err) => {
    if (err) throw err;
  });
});
```

By encapsulating related operations within the same asynchronous callback, you ensure that they execute in the correct sequence, mitigating race conditions and preserving the integrity and performance of your Node.js application.

Perform input validation

Input validation is paramount for securing your Node.js application against various types of attacks, including SQL injection, cross-site scripting (XSS), command injection, and more. Failure to validate input properly can leave your application vulnerable to exploitation. To mitigate these risks, input should be thoroughly sanitized and validated against expected formats or a list of accepted inputs.

```
const validator = require('validator');

// Example of input validation using the validator module
const userInput = req.body.username;

if (validator.isAlphanumeric(userInput)) {
  // Proceed with safe operation
} else {
  // Handle invalid input
}
```

One effective approach to simplify input validation in Node.js applications is by utilizing specialized modules such as `validator` and `express-mongo-sanitize`. These modules provide convenient functions for validating and sanitizing input data, helping developers mitigate common security risks associated with user input.

```
const express = re
const { sanitizeQuery } = require('express-mongo-sanitize');
const validator = require('validator');

const app = express();

// Middleware to sanitize query parameters
app.use(sanitizeQuery());

// Route handler with input validation
app.get('/user', (req, res) => {
  const { username } = req.query;

  if (validator.isAlphanumeric(username)) {
    // Proceed with safe operation
    res.send('Valid username');
  } else {
    // Handle invalid input
    res.status(400).send('Invalid username');
  }
});

app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

It's crucial to understand that JavaScript's dynamic nature and the parsing behavior of frameworks like Express can lead to varied representations of input data. Therefore, thorough validation is necessary to ensure that input data conforms to the expected format and does not contain any malicious content. By employing robust input validation techniques and leveraging specialized modules, you can enhance the security posture of your Node.js application and mitigate the risks associated with untrusted user input.

Perform output es

Performing output escaping is essential to prevent cross-site scripting (XSS) attacks, which occur when malicious scripts are injected into web pages and executed in the context of unsuspecting users. By escaping HTML and JavaScript content displayed to users, you can neutralize any potentially harmful code and ensure that it is treated as plain text by the browser.

One approach to output escaping in Node.js is to use libraries such as `escape-html` or `node-esapi`, which provide functions for encoding HTML entities and escaping special characters.

COPY

```
# Install escape-html library
npm install escape-html
```

COPY

```
const escapeHtml = require('escape-html');

// Example of output escaping using escape-html
const userInput = '<script>alert("XSS attack!");</script>';

const safeOutput = escapeHtml(userInput);
console.log(safeOutput); // Output: &lt;script&gt;alert(&quot;XSS
attack!&quot;);&lt;/script&gt;
```

Using the `escape-html` library, HTML entities in the input string are replaced with their respective encoded representations, preventing any HTML tags or JavaScript code from being interpreted by the browser as executable code.

Alternatively, you can use the `node-esapi` library for output escaping:

```
# Install node-esa
npm install node-esapi
```

```
const ESAPI = require('node-esapi').ESAPI;

// Example of output escaping using node-esapi
const userInput = '<script>alert("XSS attack!");</script>';

const safeOutput = ESAPI.encoder().encodeForHTML(userInput);
console.log(safeOutput); // Output: &lt;script&gt;alert(&quot;XSS
attack!&quot;);&lt;/script&gt;
```

Both libraries provide reliable mechanisms for escaping HTML and JavaScript content, helping to mitigate the risk of XSS attacks by ensuring that user-supplied input is properly sanitized before being displayed to other users. By incorporating output escaping into your Node.js application, you can enhance its security and protect against common web security vulnerabilities.

Perform application activity logging

Logging application activity is essential for debugging, monitoring, and security purposes. In Node.js, you can achieve comprehensive application logging using modules like Winston, Bunyan, or Pino. These modules offer functionalities for logging to different destinations, handling uncaught exceptions, and enabling easy querying of logs.

```
# Install Winston module
npm install winston
```

```
const winston = re

// Configure logger with console and file transports
const logger = winston.createLogger({
  transports: [
    new winston.transports.Console(),
    new winston.transports.File({ filename: 'application.log' })
  ],
  level: 'verbose'
});

// Example usage of logger
logger.info('Application started');
logger.error('An error occurred');
```

In the above example, we configure a logger with both console and file transports. Logs with severity level 'verbose' and above will be recorded. The logs will be output to the console and saved in a file named 'application.log'.

You can further customize logging by adding additional transports for specific purposes. For example, you can save errors to a separate log file:

```
const logger = winston.createLogger({
  transports: [
    new winston.transports.Console(),
    new winston.transports.File({ filename: 'application.log' }),
    new winston.transports.File({ filename: 'error.log', level:
'error' }) // Separate log file for errors
  ],
  level: 'verbose'
});
```

This configuration ensur   om general application logs, making it easier to

By implementing application activity logging in your Node.js application, you not only facilitate debugging and monitoring but also enhance its security posture by providing valuable insights into potential security incidents. Additionally, these logs can be utilized for feeding Intrusion Detection/Prevention Systems (IDS/IPS) and facilitating incident response efforts.

Monitor the event loop

Monitoring the event loop is crucial for ensuring the responsiveness and availability of your Node.js application, especially under heavy network traffic conditions or potential denial of service (DoS) attacks. The `toobusy-js` module provides a convenient way to monitor the event loop's workload and respond accordingly when the server becomes overloaded.

COPY

```
# Install toobusy-js module
npm install toobusy-js
```

COPY

```
const toobusy = require('toobusy-js');
const express = require('express');
const app = express();

// Middleware to monitor event loop
app.use(function(req, res, next) {
    if (toobusy()) {
        // Log if necessary
        res.status(503).send("Server Too Busy");
    } else {
        next();
    }
});
```

In the above example, we utilize the `toobusy-js` module as middleware in an Express application. This middleware checks the current state of the event loop and, if it determines that the server is too busy (based on configured thresholds), it responds with a 503 status code and the message "Server Too Busy".

By incorporating event loop monitoring with `toobusy-js`, you can proactively manage server load and prevent potential degradation of service quality under high traffic conditions. This helps to ensure that your Node.js application remains responsive and available to users even during periods of increased demand or potential DoS attacks.

Take precautions against brute-forcing

Protecting against brute-force attacks is crucial for safeguarding the security of your Node.js application, particularly on sensitive endpoints such as login pages. Fortunately, several modules are available in the Node.js ecosystem to help mitigate this threat, including `express-bouncer`, `express-brute`, and `rate-limiter`. Depending on your requirements, you can choose the most suitable module and configure it to enhance your application's resilience against brute-force attacks.

```
# Install express-bouncer module
npm install express-bouncer

# Install express-brute module
npm install express-brute

# Install rate-limiter module
npm install rate-limiter
```

```
const express = re
const bouncer = require('express-bouncer');
const ExpressBrute = require('express-brute');
const RateLimiter = require('rate-limiter');
const svgCaptcha = require('svg-captcha');

const app = express();

// Configure express-bouncer
bouncer.whitelist.push('127.0.0.1'); // Allowlist an IP address
bouncer.blocked = function (req, res, next, remaining) {
    res.status(429).send("Too many requests have been made. Please
wait " + remaining/1000 + " seconds.");
};
// Route to protect
app.post("/login", bouncer.block, function(req, res) {
    if (LoginFailed) { }
    else {
        bouncer.reset(req);
    }
});

// Configure express-brute
const store = new ExpressBrute.MemoryStore(); // Store state locally
(do not use in production)
const bruteforce = new ExpressBrute(store);
app.post('/auth',
    bruteforce.prevent, // Error 429 if we hit this route too often
    function (req, res, next) {
        res.send('Success!');
    }
);

// Configure rate-limiter
const limiter = new RateLimiter();
limiter.addLimit('/login', 'GET', 5, 500); // Login page can be
```

```
requested 5 times at max within 500 seconds

// CAPTCHA generation using svg-captcha
app.get('/captcha', function (req, res) {
    const captcha = svgCaptcha.create();
    req.session.captcha = captcha.text;
    res.type('svg');
    res.status(200).send(captcha.data);
});

// Account lockout using mongoose (example)
// Refer to documentation or blog posts for detailed implementation

app.listen(3000, () => {
    console.log('Server is running on port 3000');
});
```

By incorporating these modules and techniques into your Node.js application, you can effectively mitigate the risk of brute-force attacks and enhance the security of sensitive endpoints. Additionally, implementing features such as CAPTCHA usage and account lockout can further strengthen your application's defenses against malicious actors.

Use Anti-CSRF tokens

Using anti-CSRF (Cross-Site Request Forgery) tokens is crucial for protecting your Node.js application against CSRF attacks, which aim to perform unauthorized actions on behalf of authenticated users. While `csurf` is a popular Express middleware for mitigating CSRF attacks, recent security concerns have emerged, prompting the deprecation of the package. It's essential to use alternative CSRF protection packages to ensure the security of your application.

```
# Install an alter                                       ., csrf)
npm install csrf
```

COPY

```
const express = require('express');
const csrf = require('csrf');
const app = express();

// Generate CSRF tokens
const tokens = new csrf();

// Middleware to set CSRF token in response headers
app.use(function(req, res, next) {
    const secret = 'your-secret-key'; // Replace with your secret key
    const token = tokens.create(secret);
    res.setHeader('X-CSRF-Token', token);
    next();
});

// Route to handle state-changing requests
app.post('/change-password', function(req, res) {
    // Validate CSRF token
    const secret = 'your-secret-key'; // Replace with your secret key
    if (tokens.verify(secret, req.headers['x-csrf-token'])) {
        // Proceed with the request
        res.send('Password changed successfully');
    } else {
        // Handle invalid CSRF token
        res.status(403).send('CSRF token validation failed');
    }
});

app.listen(3000, () => {
    console.log('Server is running on port 3000');
});
```

In the above example, we use the `csrf` package as an alternative to `csurf` to generate and verify CSRF tokens. The middleware sets the CSRF token in the response headers for state-changing requests. Upon receiving a state-changing request, the server verifies the CSRF token before processing the request, ensuring protection against CSRF attacks.

Prevent HTTP Parameter Pollution

Preventing HTTP Parameter Pollution (HPP) is crucial for ensuring the integrity and security of your Node.js application. HPP attacks occur when attackers exploit ambiguities in the interpretation of HTTP parameters by sending multiple parameters with the same name, potentially causing unpredictable behavior.

To mitigate the risk of HPP attacks in your Express application, you can utilize the `hpp` module, which automatically resolves parameter pollution issues by selecting only the last parameter value submitted.

COPY

```
# Install the hpp module
npm install hpp
```

COPY

```
const express = require('express');
const hpp = require('hpp');
const app = express();

// Middleware to prevent HTTP Parameter Pollution
app.use(hpp());
```

```
// Route handlers.
```

By incorporating the `hpp` middleware into your Express application, you ensure that only the last parameter value submitted for each parameter name in `req.query` and/or `req.body` is considered. This helps to eliminate ambiguity and prevents potential security vulnerabilities arising from HTTP Parameter Pollution.

With this simple configuration, you bolster the security of your Node.js application by proactively mitigating the risk of HPP attacks, thereby safeguarding against potential exploitation of parameter ambiguities.

Do not use dangerous functions

Avoiding the use of dangerous JavaScript functions and modules is essential for maintaining the security of your Node.js application. Certain functions, such as `eval()` and `child_process.exec()`, pose significant security risks, particularly when handling user input, as they can lead to remote code execution vulnerabilities and arbitrary command execution on the server.

Similarly, modules like `fs` (filesystem) and `vm` (V8 Virtual Machine) require special care to prevent security vulnerabilities. Improperly sanitizing user input before passing it to these modules can expose your application to file inclusion, directory traversal, and other serious security threats.

While it's not always possible to entirely avoid using these functions and modules, they should be used judiciously and with appropriate safeguards in place, especially when dealing with untrusted user input. Employing strict input validation, input sanitization, and sandboxing techniques can help mitigate the risks associated with their usage.

```
// Example of avoi[illegible]s

// Avoid using eval() function
const userInput = req.query.code;
const result = eval(userInput); // Avoid this usage

// Instead, consider using safer alternatives, such as Function
constructor
const func = new Function(userInput);
const result = func();

// Avoid using child_process.exec() for executing arbitrary commands
const command = req.query.command;
const { exec } = require('child_process');
exec(command, (error, stdout, stderr) => {
    // Handle the result
});

// Instead, consider using more secure alternatives, such as
child_process.spawn()
const command = req.query.command;
const { spawn } = require('child_process');
const child = spawn(command);
child.stdout.on('data', (data) => {
    // Handle the result
});

// Exercise caution when using fs module with user input
const fileName = req.query.fileName;
const fs = require('fs');
fs.readFile(fileName, (err, data) => {
    // Handle the file content
});

// Ensure proper input validation and sanitization before using user
input with fs module
```

```
const fileName = req.query.fileName;
const allowedFileN
if (allowedFileNames.includes(fileName)) {
    fs.readFile(fileName, (err, data) => {
        // Handle the file content
    });
} else {
    // Handle invalid file name
}
```

By avoiding the direct usage of dangerous functions and modules and implementing appropriate security measures, you can minimize the risk of security vulnerabilities and protect your Node.js application from potential exploitation.

Use appropriate security headers

Implementing appropriate security headers is essential for enhancing the security posture of your Node.js application and mitigating various common attack vectors. The `helmet` package provides a convenient way to set these headers in your Express application.

COPY

```
# Install the helmet package
npm install helmet
```

COPY

```
const express = require("express");
const helmet = require("helmet");

const app = express();

// Add various HTTP security headers using helmet middleware
app.use(helmet());
```

```
// Strict-Transpo
app.use(helmet.hsts()); // default configuration
// Custom configuration
app.use(
  helmet.hsts({
    maxAge: 123456,
    includeSubDomains: false,
  })
);

// X-Frame-Options header
app.use(helmet.frameguard()); // default behavior (SAMEORIGIN)

// X-XSS-Protection header
app.use(helmet.xssFilter()); // sets "X-XSS-Protection: 0"

// Content-Security-Policy (CSP) header
app.use(
  helmet.contentSecurityPolicy({
    // Directives
  })
);

// X-Content-Type-Options header
app.use(helmet.noSniff());

// Cache-Control, Surrogate-Control, Pragma, Expires headers
const nocache = require("nocache");
app.use(nocache());

// X-Download-Options header
app.use(helmet.ieNoOpen());

// Expect-CT header
const expectCt = require('expect-ct');
app.use(expectCt({ maxAge: 123 }));
app.use(expectCt({ enforce: true, maxAge: 123 }));
app.use(expectCt([illegible]Uri:
'http://example.com'}));

// X-Powered-By header
app.use(helmet.hidePoweredBy());
// Optionally set a custom value
app.use(helmet.hidePoweredBy({ setTo: 'PHP 4.2.0' }));

// Start the server
app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

By incorporating these security headers into your Express application using the `helmet` middleware, you enhance its resilience against various common web security threats, such as cross-site scripting (XSS) attacks, clickjacking, MIME type sniffing, and more. Additionally, you minimize the risk of information leakage and improve the overall security posture of your application.

Listen to errors when using EventEmitter

When utilizing EventEmitter in your Node.js application, it's crucial to handle errors effectively to prevent uncaught exceptions that may crash your application. Errors within an EventEmitter object are typically propagated through an error event. If there are no listeners attached to this error event, the error will be thrown, resulting in an unhandled exception.

To ensure robust error handling with EventEmitter, always listen for error events and handle them appropriately.

COPY

```
const events = require('events');
```

```
// Define a custom EventEmitter class
function MyEventEm
    events.EventEmitter.call(this);
}

// Inherit from EventEmitter
require('util').inherits(MyEventEmitter, events.EventEmitter);

// Define a function that may emit an error
MyEventEmitter.prototype.someFunction = function(param1, param2) {
    // Simulate an error
    const err = new Error('An error occurred');
    // Emit the error event
    this.emit('error', err);
};

// Create an instance of the custom EventEmitter class
const emitter = new MyEventEmitter();

// Listen for error events and handle them
emitter.on('error', function(err) {
    // Perform necessary error handling here
    console.error('Error:', err.message);
});

// Trigger the function that may emit an error
emitter.someFunction();
```

In the example above:

- We define a custom EventEmitter class `MyEventEmitter` that inherits from `events.EventEmitter`.
- Inside the custom class, we define a function `someFunction` that may emit an error event.
- We create an instance of the custom EventEmitter class `emitter`.

- We listen for the err[illegible]handle it appropriately.
- Finally, we trigger the function `someFunction` to potentially emit an error event.

By following this pattern, you ensure that errors occurring within EventEmitter objects are properly handled, reducing the risk of uncaught exceptions and improving the stability of your Node.js application.

Set cookie flags appropriately

When managing session information using cookies in web applications, it's crucial to set appropriate flags for each cookie to mitigate session management vulnerabilities. Flags like `httpOnly`, `Secure`, and `SameSite` play a vital role in enhancing the security of session cookies.

COPY

```
# Install the express-session package
npm install express-session
```

COPY

```
const express = require('express');
const session = require('express-session');

const app = express();

// Set up session middleware with appropriate cookie flags
app.use(session({
    secret: 'your-secret-key', // Secret key for session encryption
    name: 'cookieName', // Name of the cookie
    cookie: {
        secure: true, // Send cookie only over HTTPS
        httpOnly: true, // Prevent client-side JavaScript from
accessing the cookie
        path: '/user', // Limit the cookie to a specific path
        sameSite: true // Prevent cross-site request forgery (CSRF)
attacks
    }
}));

// Define your routes and other middleware...
```

In the above example:

- We use the `express-session` package to manage sessions in our Express application.
- Within the `session` middleware configuration, we set various flags for the session cookie:
  - `secure`: Ensures that the cookie is sent only over HTTPS, enhancing security.
  - `httpOnly`: Prevents client-side JavaScript from accessing the cookie, mitigating XSS attacks.
  - `path`: Limits the scope of the cookie to a specific path (`/user` in this case).
  - `sameSite`: Helps prevent CSRF attacks by restricting the cookie from being sent in cross-site requests.
- Other options, such as `secret` (for session encryption), `name` (name of the cookie), and additional flags like `domain`, `expires`, etc., can also be configured as needed.

By setting these cookie flags appropriately, you bolster the security of your application's session management and mitigate various common web security threats.

Avoid eval(), setTimeout(), and setInterval()

avoiding functions like `eval()`, `setTimeout()`, and `setInterval()` is crucial for maintaining security in your applications. Let's explore how to avoid using these functions along with real-world examples and best practices.

First, let's look at how t

```
// Instead of using eval() to dynamically access DOM elements:
const getElementForm = getElementType == 'id' ? 'getElementById' :
'getElementByName';
const priceTagValue = document[getElementForm](elementId).value;

// Instead of using eval() to dynamically require a file in Node.js:
const db = './db.json';
const dataPoints = require(db);
```

In the examples above, we're using safer alternatives to `eval()` to achieve the same functionality. By directly accessing DOM elements or requiring files without using `eval()`, we mitigate the risk of code injection vulnerabilities.

Now, let's discuss why it's important to avoid using `setTimeout()` and `setInterval()` with code strings:

```
// Instead of passing a code string to setTimeout():
setTimeout(() => console.log(1 + 1), 1000);

// Instead of passing a code string to setInterval():
setInterval(() => console.log(1 + 1), 1000);
```

Using arrow functions or function references instead of code strings ensures that the code is securely executed. Passing code strings to `setTimeout()` or `setInterval()` can lead to potential security vulnerabilities and is generally discouraged.

By following these best practices and avoiding the use of `eval()`, `setTimeout()`, and `setInterval()` with code strings, you can enhance the security of your applications and minimize the risk of code injection attacks and other security vulnerabilities.

Avoid new Functic

avoiding the use of the `Function` constructor is paramount for maintaining the security of your applications. Let's delve into why it's important to avoid `new Function()` and explore safer alternatives:

```
// Instead of using the Function constructor:
const addition = new Function('a', 'b', 'return a + b');
addition(1, 1); // This executes the function with parameters 1 and 1
```

When using `new Function()`, you're essentially creating a new function object from a string, which poses similar security risks as `eval()`. This means that if the string passed to `new Function()` is derived from untrusted user input, it can lead to code injection vulnerabilities.

To mitigate these risks, it's better to define functions using traditional function expressions or arrow functions:

```
// Define a function using a function expression:
const addition = function(a, b) {
  return a + b;
};
addition(1, 1);

// Define a function using an arrow function:
const addition = (a, b) => a + b;
addition(1, 1);
```

By defining functions using function expressions or arrow functions, you eliminate the risk of code injection vulnerabilities associated with using the `Function` constructor.

This approach ensures th [illegible] ven when dealing with untrusted user input.

Remember, security should always be a top priority in software development, and avoiding potentially risky constructs like `new Function()` is a step in the right direction.

Avoid code serialization in JavaScript

Serialization and deserialization are powerful techniques used to convert complex data structures into a format that can be easily stored or transmitted. However, when not implemented securely, serialization and deserialization can introduce serious security vulnerabilities into your JavaScript applications. Let's explore why it's crucial to avoid code serialization and some best practices to mitigate these risks.

Why Avoid Code Serialization?

Code serialization involves converting executable code into a string format that can be stored or transmitted and then deserialized back into executable code. While this may seem convenient, it poses significant security risks, especially when dealing with untrusted or user-controlled input. Malicious actors can exploit vulnerabilities in the deserialization process to execute arbitrary code on your server, leading to severe security breaches.

Example Vulnerability in js-yaml

One popular JavaScript library that encountered security vulnerabilities due to code serialization is js-yaml. In prior versions, js-yaml was found vulnerable to code execution due to insecure deserialization practices. The vulnerability stemmed from the use of the `new Function()` constructor to deserialize JavaScript functions encoded as YAML.

```
function resolveJa[illegible]t*/) {
  /*jslint evil:true*/  var func;

  try {
    func = new Function('return ' + object);
    return func();
  } catch (error) {
    return NIL;
  }
}
```

Malicious actors could craft YAML payloads containing JavaScript functions and exploit this vulnerability to execute arbitrary code on the server.

Use a Node.js security linter

To enhance the security of your Node.js applications, consider using a Node.js security linter. This tool can help you identify insecure coding practices and potential vulnerabilities in your codebase. One such tool is eslint-plugin-security, which integrates with ESLint to provide security-focused rules for your JavaScript and Node.js projects.

How to Use eslint-plugin-security

1. **Install ESLint and eslint-plugin-security**: If you haven't already, install ESLint and eslint-plugin-security as dev dependencies in your Node.js project.

```
npm install eslint eslint-plugin-security --save-dev
```

2. **Configure ESLint**: Create or modify your ESLint configuration file (e.g., .eslintrc.js) to include the eslint-plugin-security plugin and enable the recommended

configuration.

```
module.exports = {
  plugins: ['security'],
  extends: ['plugin:security/recommended']
};
```

3. **Run ESLint**: Now, run ESLint in your project directory to analyze your code and identify potential security issues.

```
npx eslint .
```

How eslint-plugin-security Helps

- **Detect Insecure Coding Conventions**: eslint-plugin-security includes rules to detect insecure coding practices, such as the use of `eval()` with expressions or string literals, or the use of insecure Node.js APIs like `child_process.exec()`.
- **Enhanced Security Awareness**: By integrating security-focused linting into your development workflow, you and your team will become more aware of potential security vulnerabilities and adopt best practices to mitigate them.

However, it's essential to note that eslint-plugin-security may have limitations, such as false positives or rigid rule enforcement. Therefore, while it serves as a valuable tool for identifying security issues, it should be used in conjunction with other security measures, such as manual code reviews, security testing, and regular dependency updates.

References

- https://cheatsheets[illegible]Security_Cheat_Sheet.html
- https://snyk.io/blog/5-ways-to-prevent-code-injection-in-javascript-and-node-js/
- https://www.nodejs-security.com/blog

Node.js Devops DevSecOps appsec secure coding

Written by

RR **Reza Rashidi** Add your bio

Published on

DevSecOpsGuides Add blog description

RR **Reza Rashidi**

Attacking Kubernetes

In the ever-evolving landscape of cybersecurity, Kubernetes has emerged as a dominant force in manag...

RR **Reza Rashidi**

Attacking Azure

Microsoft Azure, a leading cloud computing platform, offers a myriad of services and features to fac...

RR **Reza Rashidi**

Attacking Supply Chain

In today's interconnected and rapidly evolving technological landscape, DevOps practices have revolu...