Express.js Security Tips: How You Can Save and Secure Your App

Take 7 steps to make sure that your app is invincible

Is your phone locked? Do you have a pin-code, password, fingerprint, or FaceID? I am 99 percent sure that you do. And it is clear why – you care about your safety. Nowadays, keeping your phone protected is as important as brushing your teeth in the morning.

For diligent and mindful software developers, keeping their app secure is equally important to protecting their phones. If you are a developer and you choose to neglect it – please, reconsider your approach. If you are a project owner and your development team says that data safety can wait, please, reconsider your team.

In this article, I want to talk about how to make sure that your Express.js project is safe and invincible to malicious attacks.

There are 7 simple and not very simple measures to take for the purpose of data security:

1. **Use reliable versions of Express.js**
2. **Secure the connection and data**
3. **Protect your cookies**
4. **Secure your dependencies**
5. **Validate the input of your users**
6. **Protect your system against brute force**
7. **Control user access**

Let’s have a closer look at each.

1. Use reliable versions of Express.js

Deprecated or outdated versions of Express.js are a no go. The 2nd and 3rd versions of Express are no longer supported. In these, safety or performance issues are not fixed anymore.

As a developer, you absolutely have to migrate to Express 4. This version is a revolution! It is quite different in terms of the routing system, middleware, and other minor aspects.

2. Secure the connection and data

To secure HTTP headers, you can make use of Helmet.js – a helpful Node.js module. It is a collection of 13 middleware functions for setting HTTP response headers. In particular, there are functions for setting Content Security Policy, handling Certificate Transparency, preventing clickjacking, disabling client-side caching, or adding some small XSS protections.

```
npm install helmet --save
```

Even if you do not want to use all the functions of Helmet, the absolute minimum that you must do is to disable X-Powered-By header:

```
app.disable('x-powered-by')
```

This header can be used to detect that the application is powered by Express, which lets hackers conduct a precise attack. Surely, X-Powered-By header is not the only way to identify an Express-run application, but it is probably the most common and simple one.

To protect your system from HTTP parameter pollution attacks, you can use HPP. This middleware puts aside such parameters as req.query and req.body and selects the latest parameter value instead. The installation command looks as follows:

```
npm install hpp --save
```

To encrypt data which is being sent from the client to the server, use Transport Layer Security (TLS). TLS is a cryptographic protocol for securing the computer network, the descendant of the Secure Socket Layer (SSL) encryption. TLS can be handled with Nginx – a free but effective HTTP server – and Let's Encrypt – a free TLS certificate.

3. Protect your cookies

In Express.js 4, there are two cookie session modules:

- express-session (in Express.js 3, it was express.session)
- cookie-session (in Express.js 3, it was express.cookieSession)

The express-session module stores session ID in the cookie and session data on the server. The cookie-session stores all the session data to the cookie.

In general, cookie-session is more efficient. Yet, if the session data you need to store is complex and likely to exceed 4096 bytes per cookie, use express-session. Another reason to use express-session is when you need to keep the cookie data invisible to the client.

Besides, you should set cookie security options, namely:

- secure
- httpOnly
- domain
- path
- expires

If “secure” is set to “true”, the browser will send cookies only via HTTPS. If “httpOnly” is set to “true”, the cookie will be sent not via client JS but via HTTP(S). The value of “domain” indicates the domain of the cookie. If the cookie domain matches the server domain, “path” is used to indicate the cookie path. If the cookie path matches the request path, the cookie will be sent in the request. Finally, as the name itself suggests, the value of “expires” stands for the time when the cookies will expire.

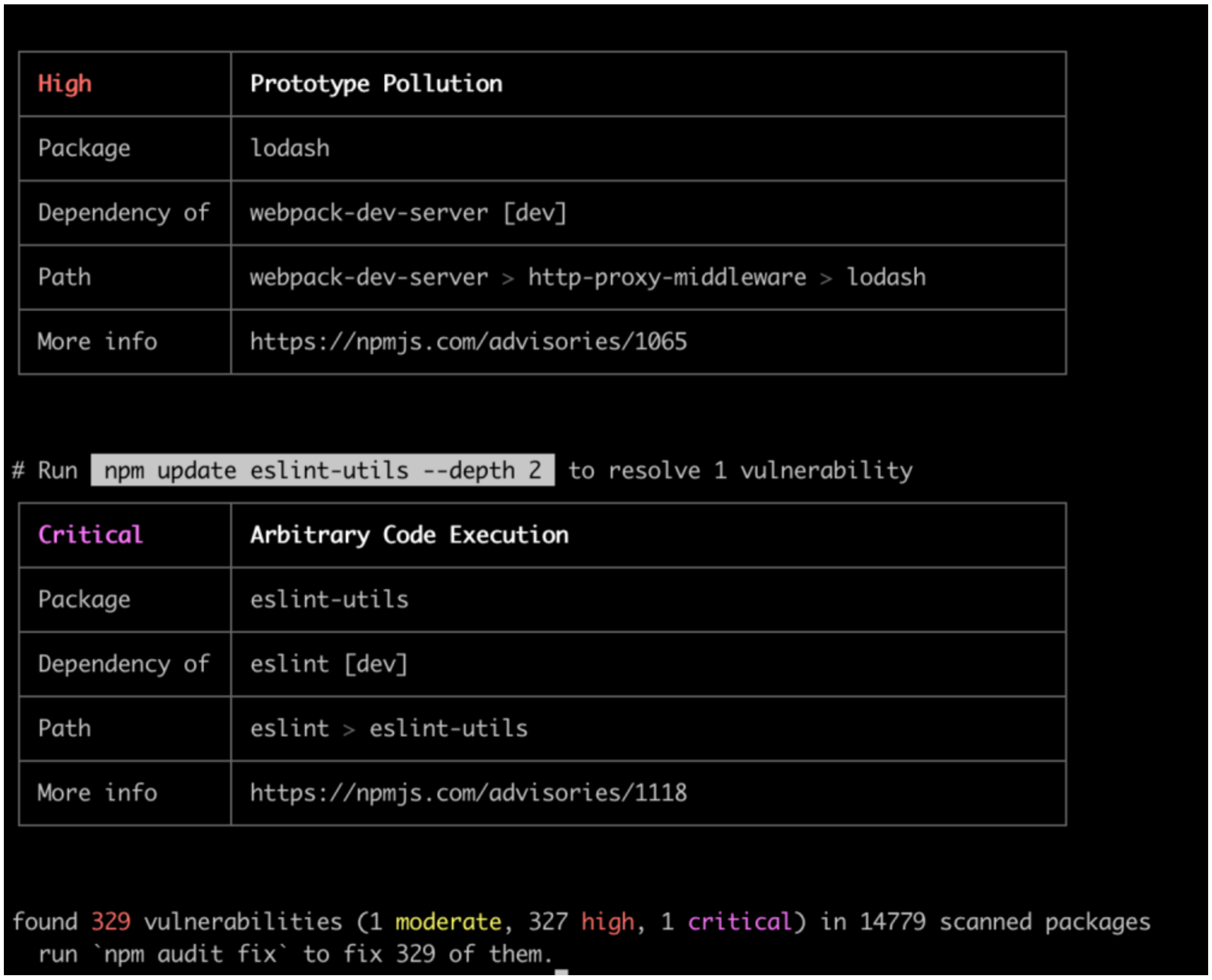
Another important recommendation is not to use the default session cookie name. It may enable hackers to detect the server and to run a targeted attack. Instead, use generic cookie names.

4. Secure your dependencies

No doubt, npm is a powerful web development tool. However, to ensure the highest level of security, consider using only the 6th version of it – npm@6. The older ones may contain some serious dependency safety vulnerabilities, which will endanger your entire app. Also, to analyze the tree of dependencies, use the following command:

```
npm audit
```

npm audit can help to fix real problems in your project. It checks all your dependencies in dependencies, devDependencies, bundledDependencies, and optionalDependencies, but not your peerDependencies. Here you can read about all current vulnerabilities in any npm packages.



```
┌───────────────┬──────────────────────────────────────────────────────────────┐
│ High          │ Prototype Pollution                                          │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ Package       │ lodash                                                       │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ Dependency of │ webpack-dev-server [dev]                                     │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ Path          │ webpack-dev-server > http-proxy-middleware > lodash          │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ More info     │ https://npmjs.com/advisories/1065                            │
└───────────────┴──────────────────────────────────────────────────────────────┘


# Run  npm update eslint-utils --depth 2  to resolve 1 vulnerability
┌───────────────┬──────────────────────────────────────────────────────────────┐
│ Critical      │ Arbitrary Code Execution                                     │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ Package       │ eslint-utils                                                 │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ Dependency of │ eslint [dev]                                                 │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ Path          │ eslint > eslint-utils                                        │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ More info     │ https://npmjs.com/advisories/1118                            │
└───────────────┴──────────────────────────────────────────────────────────────┘


found 329 vulnerabilities (1 moderate, 327 high, 1 critical) in 14779 scanned packages
  run `npm audit fix` to fix 329 of them.
```

Another tool to ensure dependency safety is Snyk. Snyk runs the application check to identify whether it contains any vulnerability listed in Snyk's open-source database. To conduct the check, run three simple steps.

Step 1. Install Snyk

```
npm install -g snyk

cd your-app
```

Step 2. Run a test

```
snyk test
```

Step 3. Learn how to fix the issue

```
snyk wizard
```

Wizard is a Snyk method, which explains the nature of the dependency vulnerability and offers ways of fixing it.

5. Validate the input of your users

Controlling user input is an extremely important part for server-side development. This is a no less important problem than unauthorized requests, which will be described in the seventh part of this article.

First of all, wrong user input can break your server when some values are undefined and you do not have error handling for a specific endpoint. However, different ORM systems can have unpredictable behavior when you try to set undefined, null, or other data types in the database.

For example, destroyAll method in Loopback.js ORM (Node.js framework) can destroy all data in a table of the database: when it does not match any records it deletes everything as described here. Imagine that you can lose all data in a production table just because you have ignored input validation.

Use body/object validation for intermediate inspections

To start with, you can use body/object validation for intermediate inspections. For example, we use ajv validator which is the fastest JSON Schema validator for Node.js.

```
const Ajv = require('ajv');
const ajv = new Ajv({allErrors: true});
const speaker = {
  'type': 'object',
  'required': [
    'id',
    'name'
  ],
  'properties': {
    'id': {
      'type': 'integer',
      },
      'name': {
         'type': 'string',
      },
   },
};
const conversation = {
   type: 'object',
   required: [
      'duration',
      'monologues'
   ],
   properties: {
      duration: {
         type: 'integer',
      },
      monologues: {
         type: 'array',
         items: monolog,
      },
   },
};
const body = {
   type: 'object',
   required: [
      'speakers',
      'conversations'
   ],
   properties: {
      speakers: {
         type: 'array',
        items: speaker,
      },
      conversations: {
        type: 'array',
        items: conversation,
      },
    },
};
const validate = ajv.compile(body);
const isValidTranscriptBody = transcriptBody => {
    const isValid = validate(transcriptBody);
    if (!isValid) {
      console.error(validate.errors);
    }
    return isValid;

};
```

Handle errors

Now, imagine that you forgot to check a certain object and you do some operations with the undefined property. Or you use a certain library and you get an error. It can break your instance, and the server will crash. Then, the attacker can ping a specific endpoint where there is this vulnerability and can stop your server for a long time.

The simplest way to do an error handling is to use try-catch construction:

```
try {
   const data = body;
   if (data.length === 0) throw new Error('Client Error');
   const beacons = await
this.beaconLogService.filterBeacon(data);
   if (beacons.length > 0) {
      const max = beacons.reduce((prev, current) =>
(prev.rssi > current.rssi) ? prev : current);
      await this.beaconLogService.save({
         ...max,
         userId: headers['x-uuid']
      });
      return {
         data: {
            status: 'Saved',
            position: max
         },
      };
   }
   return {
      data: {
         status: 'Not valid object,
      },
   };
}
catch(err) {
   this.logger.error(err.message, err.stack);
   throw new HttpException('Server Error',
HttpStatus.INTERNAL_SERVER_ERROR);

}
```

Feel free to use a new Error(‘message’) constructor for error handling or even extend this class for your own purpose!

Use JOI

The main lesson here is that you should always validate user input so you don't fall victim to man-in-the-middle attacks. Another way to do it is with the help of @hapi/joi – a part of the hapi ecosystem and a powerful JS data validation library.

Pay attention here that the module joi has been deprecated. For this reason, the following command is a no go:

```
npm install joi
```

Instead, use this one:

```
npm install @hapi/joi
```

Use express-validator

One more way to validate user input is to use express-validator – a set of express.js middlewares, which comprises validator.js and function sanitizer. To install it, run the following command:

```
npm install --save express-validator
```

Sanitize user input

Also, an important measure to take is to sanitize user input to protect the system from a MongoDB operator injection. For this, you should install and use express-mongo-sanitize:

```
npm install express-mongo-sanitize
```

Protect your app against CSRF

Besides, you should protect your app against cross-site request forgery (CSRF). CSRF is when unauthorized commands are sent from a trusted user. You can do this with the help of csurf.

Prior to that, you need to make sure that session middleware for cookies is configured as described earlier in this article. To install this Node.js module, run the command:

```
npm install csurf
```

6. Protect your system against brute force

A brute force attack is the simplest and most common way to get access to a website or a server. The hacker (in most cases automatically, rarely manually) tries various usernames and passwords repeatedly to break into the system.

These attacks can be prevented with the help of rate-limiter-flexible package. This package is fast, flexible, and suitable for any Node framework.

To install, run the following command:

```
npm i --save rate-limiter-flexible
```

```
yarn add rate-limiter-flexible
```

This method has a simpler but more primitive alternative: express-rate-limit. The only thing it does is limiting repeated requests to public APIs or to password reset.

```
npm install --save express-rate-limit
```

7. Control user access

Among the authentication methods, there are tokens, Auth0, and JTW. Let's focus on the third one! JTW (JSON Web Tokens) are used to transfer authentication data in client-server applications. Tokens are created by the server, signed with a secret key, and transferred to a client. Then, the client uses these tokens to confirm identity.

Express-jwt-permissions is a tool used together with express-jwt to check permissions of a certain token. These permissions are an array of strings inside the token:

```
"permissions": [
   "status",
   "user:read",
   "user:write"

]
```

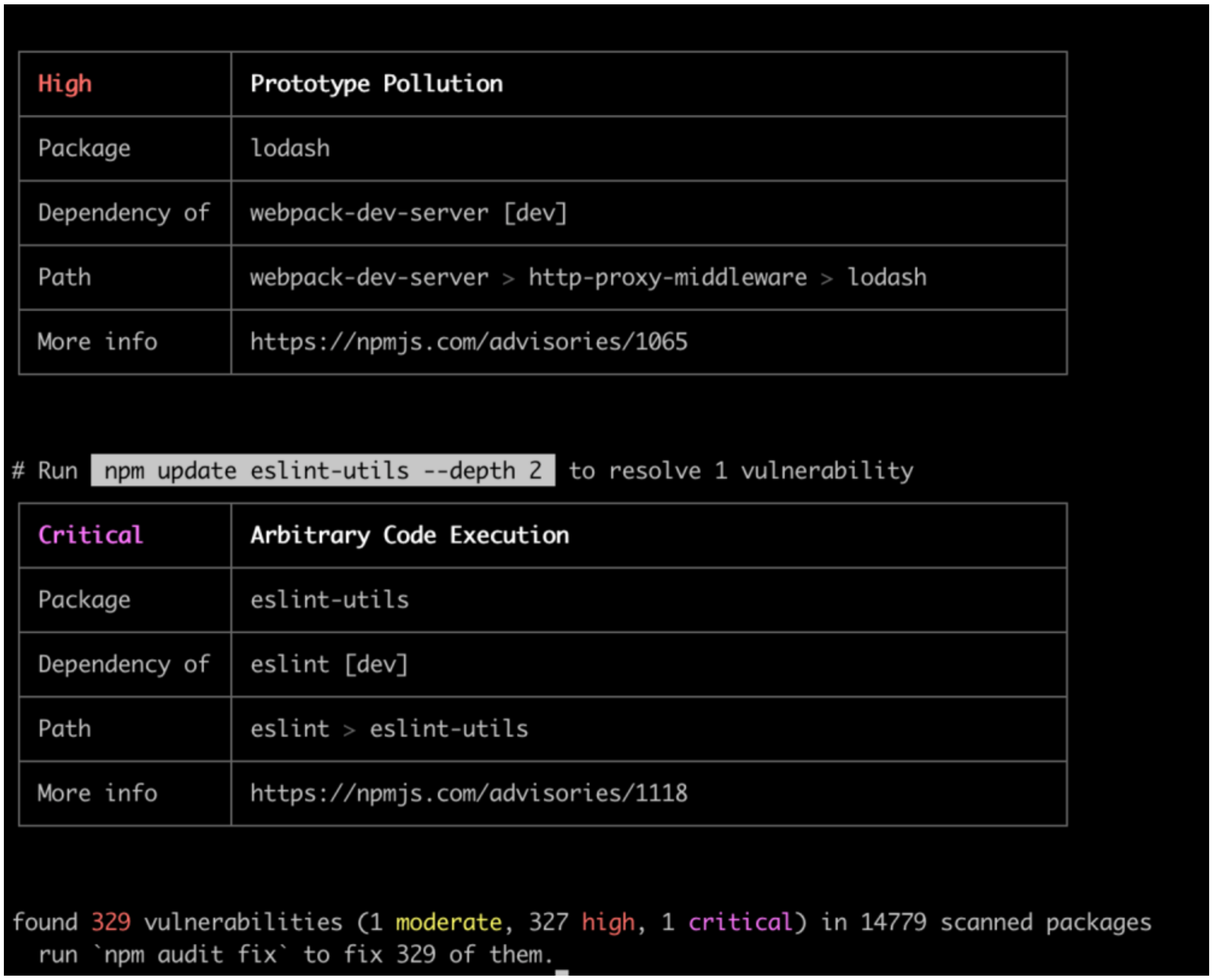
To install the tool, run the following command:

```
npm install express-jwt-permissions --save
```

To Wrap Up

Here, I have listed the essential Express.js security best practices and some tools that can be used along the way.

Just to review:



```
┌───────────────┬──────────────────────────────────────────────────────────────┐
│ High          │ Prototype Pollution                                          │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ Package       │ lodash                                                       │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ Dependency of │ webpack-dev-server [dev]                                     │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ Path          │ webpack-dev-server > http-proxy-middleware > lodash          │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ More info     │ https://npmjs.com/advisories/1065                            │
└───────────────┴──────────────────────────────────────────────────────────────┘


# Run  npm update eslint-utils --depth 2  to resolve 1 vulnerability
┌───────────────┬──────────────────────────────────────────────────────────────┐
│ Critical      │ Arbitrary Code Execution                                     │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ Package       │ eslint-utils                                                 │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ Dependency of │ eslint [dev]                                                 │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ Path          │ eslint > eslint-utils                                        │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ More info     │ https://npmjs.com/advisories/1118                            │
└───────────────┴──────────────────────────────────────────────────────────────┘


found 329 vulnerabilities (1 moderate, 327 high, 1 critical) in 14779 scanned packages
  run `npm audit fix` to fix 329 of them.
```

I strongly recommend that you make sure that your application is resistant to malicious attacks. Otherwise, your business and your users may suffer significant losses.

Do you have an idea for Express.js project?

My company KeenEthics is experienced in express js development. In case you need a free estimate of a similar project, feel free to get in touch.

If you have enjoyed the article, you should definitely continue with a piece on data safety in outsourcing to Ukraine: KeenEthics Team on Guard: Your Data is Safe in Ukraine. The original article posted on KeenEthics blog can be found here: express js security tips.

P.S.

A huge shout-out to Volodia Andrushchak, Full-Stack Software Developer @KeenEthics for helping me with the article.

The original article posted on KeenEthics blog can be found here: Express.js Security Tips: Save Your App!