Express

Production Best Practices: Security

Overview

The term ***"production"*** refers to the stage in the software lifecycle when an application or API is generally available to its end-users or consumers. In contrast, in the ***"development"*** stage, you're still actively writing and testing code, and the application is not open to external access. The corresponding system environments are known as ***production*** and ***development*** environments, respectively.

Development and production environments are usually set up differently and have vastly different requirements. What's fine in development may not be acceptable in production. For example, in a development environment you may want verbose logging of errors for debugging, while the same behavior can become a security concern in a production environment. And in development, you don't need to worry about scalability, reliability, and performance, while those concerns become critical in production.

> **Note**: If you believe you have discovered a security vulnerability in Express, please see Security Policies and Procedures.

Security best practices for Express applications in production include:

- Don't use deprecated or vulnerable versions of Express
- Use TLS
- Use Helmet
- Use cookies securely
- Ensure your dependencies are secure
- Avoid other known vulnerabilities
- Additional considerations

Don't use deprecated or vulnerable versions of Express

Express 2.x and 3.x are no longer maintained. Security and performance issues in these versions won't be fixed. Do not use them! If you haven't moved to version 4, follow the migration guide.

Also ensure you are not using any of the vulnerable Express versions listed on the Security updates page. If you are, update to one of the stable releases, preferably the latest.

Use TLS

If your app deals with or transmits sensitive data, use Transport Layer Security (TLS) to secure the connection and the data. This technology encrypts data before it is sent from the client to the server, thus preventing some common (and easy) hacks. Although Ajax and POST requests might not be visibly obvious and seem "hidden" in browsers, their network traffic is vulnerable to packet sniffing and man-in-the-middle attacks.

You may be familiar with Secure Socket Layer (SSL) encryption. TLS is simply the next progression of SSL. In other words, if you were using SSL before, consider upgrading to TLS. In general, we recommend Nginx to

handle TLS. For a good reference to configure TLS on Nginx (and other servers), see Recommended Server Configurations (Mozilla Wiki).

Also, a handy tool to get a free TLS certificate is Let's Encrypt, a free, automated, and open certificate authority (CA) provided by the Internet Security Research Group (ISRG).

Use Helmet

Helmet can help protect your app from some well-known web vulnerabilities by setting HTTP headers appropriately.

Helmet is actually just a collection of nine smaller middleware functions that set security-related HTTP headers:

- csp sets the `Content-Security-Policy` header to help prevent cross-site scripting attacks and other cross-site injections.
- hidePoweredBy removes the `X-Powered-By` header.
- hpkp Adds Public Key Pinning headers to prevent man-in-the-middle attacks with forged certificates.
- hsts sets `Strict-Transport-Security` header that enforces secure (HTTP over SSL/TLS) connections to the server.
- ieNoOpen sets `X-Download-Options` for IE8+.
- noCache sets `Cache-Control` and Pragma headers to disable client-side caching.
- noSniff sets `X-Content-Type-Options` to prevent browsers from MIME-sniffing a response away from the declared content-type.
- frameguard sets the `X-Frame-Options` header to provide clickjacking protection.
- xssFilter sets `X-XSS-Protection` to enable the Cross-site scripting (XSS) filter in most recent web browsers.

Install Helmet like any other module:

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

Then to use it in your code:

```
// ...

var helmet = require('helmet')
app.use(helmet())

// ...
```

At a minimum, disable X-Powered-By header

If you don't want to use Helmet, then at least disable the `X-Powered-By` header. Attackers can use this header (which is enabled by default) to detect apps running Express and then launch specifically-targeted attacks.

So, best practice is to to turn off the header with the `app.disable()` method:

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

If you use `helmet.js`, it takes care of this for you.

> **Note**: Disabling the `X-Powered-By header` does not prevent a sophisticated attacker from determining that an app is running Express. It may discourage a casual exploit, but there are other ways to determine an app is running Express.

Use cookies securely

To ensure cookies don't open your app to exploits, don't use the default session cookie name and set cookie security options appropriately.

There are two main middleware cookie session modules:

- express-session that replaces `express.session` middleware built-in to Express 3.x.
- cookie-session that replaces `express.cookieSession` middleware built-in to Express 3.x.

The main difference between these two modules is how they save cookie session data. The express-session middleware stores session data on the server; it only saves the session ID in the cookie itself, not session data. By default, it uses in-memory storage and is not designed for a production environment. In production, you'll need to set up a scalable session-store; see the list of compatible session stores.

In contrast, cookie-session middleware implements cookie-backed storage: it serializes the entire session to the cookie, rather than just a session key. Only use it when session data is relatively small and easily encoded as primitive values (rather than objects). Although browsers are supposed to support at least 4096 bytes per cookie, to ensure you don't exceed the limit, don't exceed a size of 4093 bytes per domain. Also, be aware that the cookie data will be visible to the client, so if there is any reason to keep it secure or obscure, then express-session may be a better choice.

Don't use the default session cookie name

Using the default session cookie name can open your app to attacks. The security issue posed is similar to `X-Powered-By`: a potential attacker can use it to fingerprint the server and target attacks accordingly.

To avoid this problem, use generic cookie names; for example using express-session middleware:

```
var session = require('express-session')
app.set('trust proxy', 1) // trust first proxy
app.use(session({
  secret: 's3Cur3',
  name: 'sessionId'
}))
```

Set cookie security options

Set the following cookie options to enhance security:

- `secure` - Ensures the browser only sends the cookie over HTTPS.
- `httpOnly` - Ensures the cookie is sent only over HTTP(S), not client JavaScript, helping to protect against cross-site scripting attacks.
- `domain` - indicates the domain of the cookie; use it to compare against the domain of the server in which the URL is being requested. If they match, then check the path attribute next.

- `path` - indicates the path of the cookie; use it to compare against the request path. If this and domain match, then send the cookie in the request.
- `expires` - use to set expiration date for persistent cookies.

Here is an example using cookie-session middleware:

```
var session = require('cookie-session')
var express = require('express')
var app = express()

var expiryDate = new Date(Date.now() + 60 * 60 * 1000) // 1 hour
app.use(session({
  name: 'session',
  keys: ['key1', 'key2'],
  cookie: {
    secure: true,
    httpOnly: true,
    domain: 'example.com',
    path: 'foo/bar',
    expires: expiryDate
  }
}))
```

Ensure your dependencies are secure

Using npm to manage your application's dependencies is powerful and convenient. But the packages that you use may contain critical security vulnerabilities that could also affect your application. The security of your app is only as strong as the "weakest link" in your dependencies.

Since npm@6, npm automatically reviews every install request. Also you can use 'npm audit' to analyze your dependency tree.

```
$ npm audit
```

If you want to stay more secure, consider Snyk.

Snyk offers both a command-line tool and a Github integration that checks your application against Snyk's open source vulnerability database for any known vulnerabilities in your dependencies. Install the CLI as follows:

```
$ npm install -g snyk
$ cd your-app
```

Use this command to test your application for vulnerabilities:

```
$ snyk test
```

Use this command to open a wizard that walks you through the process of applying updates or patches to fix the vulnerabilities that were found:

```
$ snyk wizard
```

Avoid other known vulnerabilities

Keep an eye out for Node Security Project or Snyk advisories that may affect Express or other modules that your app uses. In general, these databases are excellent resources for knowledge and tools about Node security.

Finally, Express apps - like any other web apps - can be vulnerable to a variety of web-based attacks. Familiarize yourself with known web vulnerabilities and take precautions to avoid them.

Additional considerations

Here are some further recommendations from the excellent Node.js Security Checklist. Refer to that blog post for all the details on these recommendations:

- Implement rate-limiting to prevent brute-force attacks against authentication. One way to do this is to use StrongLoop API Gateway to enforce a rate-limiting policy. Alternatively, you can use middleware such as express-limiter, but doing so will require you to modify your code somewhat.
- Use csurf middleware to protect against cross-site request forgery (CSRF).
- Always filter and sanitize user input to protect against cross-site scripting (XSS) and command injection attacks.
- Defend against SQL injection attacks by using parameterized queries or prepared statements.
- Use the open-source sqlmap tool to detect SQL injection vulnerabilities in your app.
- Use the nmap and sslyze tools to test the configuration of your SSL ciphers, keys, and renegotiation as well as the validity of your certificate.
- Use safe-regex to ensure your regular expressions are not susceptible to regular expression denial of service attacks.