AppLaunch

1.Security practices for Express Js

Express.js — a minimalist web framework built to create Node.js web applications. It comes with HTTP utilities for rapidly developing APIs

But when it comes to security it's not perfect.

Helmet js securing HTTP headers that are returned by your Express apps.

Install Helmet js

npm install helmet --save

usage

```
var helmet =require('helmet');

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

AppLaunch

list of HTTP headers supported by Helmet.js

1.1 Content security policy

lets you set the Content-Security-Policy which allows you to mitigate cross-site scripting attacks.

Function : helmet.contentSecurityPolicy(options)

1.2 Expect - CT

sets the Expect-CT header. This prevents mis-issued SSL certificates.

Function: helmet.expectCt(options)

1.3 X-DNS-Prefetch-Control

lets you set the X-DNS-Prefetch-Control header in Express. This helps control DNS prefetching and improves user privacy.

Function:helmet.dnsPrefetchControl

1.4 X-Frame-Options

helmet.frameguard sets the X-Frame-Options in the header to prevent clickjacking attacks.

Function: helmet.frameguard(options)

1.5 X-Powered-By

helmet.hidePoweredBy removes the X-Powered-By browser, which can give valuable information to malicious users to exploit. In Express, this information is sent to the public by default.

Function: helmet.hidePoweredBy(options)

1.6 Strict-Transport-Security

helmet.hsts sets the Strict-Transport-Security header. This tells the browser to prefer HTTPS over HTTP.
Function: helmet.hsts(options)

1.7 X-Download-Options

helmet.isNoOpen sets the X-Download-Options header. This is specific to the vulnerabilities in IE 8 and forces potentially unsafe downloads to be saved and prevents the execution of HTML in your site's context.

Function: helmet.ieNoOpen(options)

1.8 X-Content-Type-Options

helmet.noSniff sets the X-Content-Type-Options header to nosniff. This prevents MIME type sniffing.

Function: helmet.noSniff(options)

1.9 Referrer-Policy

helmet.referrerPolicy sets the Referrer-Policy header. This controls the information inside the Referer header.

Function: helmet.referrerPolicy(options)

1.10 X-XSS-Protection

helmet.xssFilter prevents cross-site scripting. While browsers come with a filter that prevents this by default, it is not evenly applied and bugginess can range depending on if the end-user is using Chrome, IE, Firefox, Safari, or something else.
Using helmet.xssFilter puts another layer of security on your API. Here is how to use it:

Function: helmet.xssFilter(options)

2.NodeJs - Owasp security cheat sheet

There are several different recommendations to enhance security of Node.js applications. These are categorized as:

1. Application Security
2. Error and Exception handling
3. Server Security
4. Platform Security

2.1Application Security

2.1.1 Use flat Promise chains:

Asynchronous callback functions are the strongest function in node js but increasing nesting within the callback functions is so difficult to handle the errors in the node js .

Promises provide top down execution and asynchronous while delivering the errors and results to next function

2.1.2 Set request size limits:

Buffering and parsing of request bodies can be a resource intensive task. If there is no limit on the size of requests, attackers can send requests with large request bodies that can exhaust server memory and/or fill disk space.

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

2.1.3 Perform the input validation:

Input validation is a crucial part of application security. Input validation failures can result in many different types of application attacks. These include SQL Injection, Cross-Site Scripting, Command Injection, Local/Remote File Inclusion, Denial of

Service, Directory Traversal, LDAP Injection and many other injection attacks. In order to avoid these attacks, input to your application should be sanitized first

```
mongoSanitize = require('mongo-express-sanitize');

app.use(mongoSanitize());
```

2.1.4 Perform output escaping:

you should escape all HTML and JavaScript content shown to users via application in order to prevent cross-site scripting (XSS) attacks.

2.1.5 Take precaution against brute forcing:

Attackers can use brute-forcing as a password guessing attack to obtain account passwords. Therefore, application developers should take precautions against brute-force attacks especially in login pages

```
const limiter = new RateLimiter();

limiter.addLimit('/login', 'GET', 5, 500); // login page can
be requested 5 times at max within 500 seconds
```

2.1.6 Remove unnecessary routes:

A web application should not contain any page that is not used by users, as it may increase the attack surface of the application. Therefore, all unused API routes should be disabled in Node.js applications.

2.1.7 Prevent HTTP Parameter Pollution:

HTTP Parameter Pollution(HPP) is an attack in which attackers send multiple HTTP parameters with the same name and this causes your application to interpret them in an unpredictable way.

```
const hpp = require('hpp');
app.use(hpp());
```

2.1.8 Return what is necessary:

Information about the users of an application is among the most critical information about the application. User tables generally include fields like id, username, full name, email address, birth date, password and in some cases social security numbers. Therefore, when querying and using user objects, you need to return only needed fields as it may be vulnerable to personal information disclosure.

```
exports.sanitizeUser = function(user) {
  return {
    id: user.id,
    username: user.username,
    fullName: user.fullName
  };
};
```

2.2 Error & Exception Handling

2.2.1 Handle uncaughtException:

```
process.on("uncaughtException", function(err) {
    console.log("uncaughtException");
});
```

2.2.1 Listen to event using EventEmitter:

```
const events = require('events');
const myEventEmitter = function(){
    events.EventEmitter.call(this);
}
require('util').inherits(myEventEmitter, events.EventEmitter);
myEventEmitter.prototype.someFunction = function(param1, param2) {
    //in case of an error
    this.emit('error', err);
}
const emitter = new myEventEmitter();
emitter.on('error', function(err){
    //Perform necessary error handling here
});
```