Web Security Challenges in Node.js Applications

Shashank Kaul [1], Hrittik Bhattacharjee [2], Barry C. K. [3], Gaurav Verma [4], J. Gowthamy [5]
[1, 2, 3, 4, 5] SRM Institute of Science and Technology, Chennai, Tamil Nadu, India.

Abstract – Technology is growing at an exponential rate. New platforms and frameworks are being developed to enhance the capabilities of what can be developed and how. Node.js is a platform that provides an event-driven and asynchronous I/O platform for development using JavaScript. Now, Express is a framework developed on the Node.js platform for building flexible web applications and is very rapidly growing in popularity amongst developers. However, for the cost of availability and performance we sacrifice confidentiality thus leaving applications vulnerable to many security threats. Threats such as Cross-site Scripting (XSS), Injection, DOS attacks and many more are now things that most people can learn on the internet, making systems that do not take precautions very vulnerable. We explore the possible ways someone with malicious intent can carry out attacks when running frameworks such as Express and implement methods to make Node.js web applications secure for production.

Index Terms – Authentication, Data Validation, Injection, Security, Node.js.

1. INTRODUCTION

Cyber security is a rapidly growing area of study and has gained a large prominence over the Internet in the last decade. Not only has the risk of being targeted by a cyber-criminal increased but there has also been growth in educational resources to learn how to do such activities. Web applications are regularly exploited or attacked if correct measures are not taken in development stages. Research in security and exploits of frameworks has increased rigorously in the past few years and well-established organisations invest a lot into this field of study. Node.js [6] is a relatively new framework that is built on Google Chrome's V8 engine [9] to allow the traditionally client-sided JavaScript language to be used in creating servers and other functional modules. Node.js works in an event-driven loop which allows for tasks to be processed in a non-blocking or asynchronous paradigm. [1] Node shines in building fast, scalable network applications, as it's capable of handling a huge number of simultaneous connections with high throughput, which equates to high scalability. [1][2] In this paper we discuss the types of ways web applications can be targeted and exploited for vulnerable development. We also discuss the ways to implement solutions for these attacks. With the help of Express [2], a web framework for Node developers which allow us to create web applications with less code while still maintaining the main features of the parent platform, we implement a social media website similar to Reddit or Twitter which allows for basic tasks such as creating of user profiles, creating and editing blog posts, commenting and rating the posts.

In the following sections we discuss how the construction of this application is laid out. We apply the traditional methods of Node.js and Express development in the creation of this application in an attempt to provide a basis for assessment. We then research and implement various different aspects of securing a web application in the proposed system.

2. RELATED WORK

Our API follows the RESTful paradigm and thus we manage the CRUD operations of posts and comments as shown in the following table.

PATH	METHOD	USAGE
/posts	GET	Displays all posts.
/posts/new	GET	Show form to create new posts.
/posts	POST	Creates a new post.
/posts/:id	GET	Display a specific post.
/posts/:id/edit	GET	Show form to update a specific post.
/posts/:id	PUT	Update a specific post.
/posts/:id	DELETE	Delete a specific post.

Table 1 Applications API Routes

The above layout for our API routes makes it easier to understand the workflow of the user post and comment functionality.

We have taken into consideration the two most important areas of any web application where security must be implemented. In this section we highlight these 2 parts; authentication with session management, and data storage. Both modules discuss how we implement various *npm*

modules to create a working blog application to mimic real world tasks.

2.1. Authentication and Session Management

HTTP is a state-less protocol and thus browser cookies are used to maintain state. Most systems apply a session based authentication method where the client connects to the server to maintain a session-id in the browser's cookies once credential validation occurs. On the server-side, the application checks if the session has been added to a list of sessions. If so, further requests from the same client are allowed to go through.

PassportJS [10] is a very commonly used npm package for implementing various authentication strategies. We implement a cookie-based local authentication system using this package. The *passport-local-mongoose* package allows us to use both the functionalities of mongoose and passport together to carry out user authentication in local storage. The *passport-local-mongoose* package provides us with middleware attached to the schema it is added to. Various functions such as *register, authenticate,* and support for flash messages and provided in this middleware.

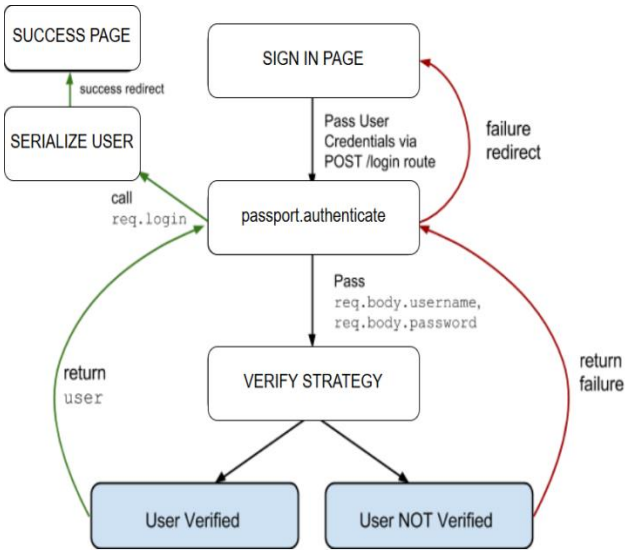

Fig 1.1 PassportJS Authentication Workflow

To maintain the user session we use an express module called *express-session* which allows us to store all the required user data within the browser cookies. This module allows for setting of various properties of browser cookies including validity and security.

In modern day browsers it is very easy to view one's cookies for individual websites. This allows attacks such as session fixation, session donation and session brute forcing which can be carried out to steal someone else's session and thus fully authenticate into the application.

2.2. Data Storage

A general purpose blog application allows registered users to perform CRUD operations on posts and comments. Most webservers implement storage of user accounts, user posts and comments using either a relational or non-relational data management system. In our case we avoided the use of the MongoDB package and implemented the *mongoose* package to create Schemas for user, posts and comments. This allows us to predefine the layout of collections in a database. It also gives us various functionalities including middleware, plugins for validation and population.

User data, post and comment data is stored in is *MongoDBs* BSON format by using *mongoose* middleware. We simultaneously added middleware to check for comment and post ownership by adding the middleware functions *checkPostOwnership* and *checkCommentOwnership*. This guarantees non-repudiation with content shared on the blog application.

The schema defined for User, Posts and Comments is shown in the following snippet.

```
var UserSchema = new mongoose.Schema(
  username: {
       type: String
  },
  password: {
       required: true,
       type: String
  }
       …
       …
       …
});
```

Similarly the schema for Posts and Comments is created. Mongoose offers references between schemas and thus we use object id (MongoDB's *_id*) references for each comment and post to its respective author (user).

```
author: {
 id: {
  type: mongoose.Schema.Types.ObjectId,
  ref: 'User'
 },
 username: String
}
```

By adding this to the Post and Comments schema we store the author's *_id* value created by MongoDB into the *author.id* property of the respective schema.

3. PORPOSED MODELLING

By introducing availability and usability we have disregarded some important security aspects and maybe even added some security threats. The proposed system discusses various

threats that could be used by exploiting our framework or lack of code. These include, attacks against the authentication system, session management and TLS, data validation, and error handling. [5][3]

3.1. Authentication

The most common attacks on authentication are brute-force attacks. A brute-force attack refers to trying all possibilities of the solution until one matches. If authentication is not correctly handled, once the attacker is aware of the server's route for login and a target username, he can send an infinite number of requests for authentication using the brute-force technique. This can be easily prevented by the use of a "rate-limiting" function [5]. By creating middleware using the *express-rate-limiter* package in node.js, we can assign a limit to the number of post requests from one client to a specific route.

```
var RateLimit = require('express-rate-limit');
var authLimiter = new RateLimit({
      // Maximum of 10 requests
      max: 10,
      delayMs: 0
      // disabled any delay between requests
}); //HTTP Code 429 on limit
app.use('/login', authLimiter);
```

The algorithm used for brute forcing will always produce a guaranteed result, however the time taken increases exponentially with more complex passwords. We also added flash messages to force use of numbers (0-9), both lower and upper-case alphabets and symbols when creating passwords. This functionality is provided by the passport middleware discussed in the existing system. Hence, we secured possible authentication problems with the system.

3.2. Session Management, Cookie Privacy and TLS

Since HTTP is stateless browsers maintain sessions with servers by storing unique identifiers assigned by the server within browser cookies. To prevent problems such as session hijacking and sniffing [3] we need to implement some sort of encryption/decryption infrastructure to hide data being sent in the HTTP headers and body from malicious users.

To do this, we implement Transport Layer Security [6][4] on a Node.js server. For development purposes we created a self-signed certificate however valid certificates are required for production environments. The snippet of code below shows how the *https* module can be used to save the key and certificate files to be used by the server. We add the certificate and key when using the *createServer(options, callback)* function.

```
var fs = require('fs');
var https = require('https');
var app = express();
var options = {
   key: fs.readFileSync(
    __dirname +'/certs/key.pem'),
   cert: fs.readFileSync(
    __dirname +'/certs/cert.pem')
};
// …EXPRESS CODE HERE
https.createServer(options,
   function(req, res) {
      res.writeHead(200);
      res.end("HTTP server running.\n");
}).listen(8000);
```

However this is not considered enough. There are methods for tricking the client into downgrading the connection to http or methods designed to break the SSL encryption called SSL-stripping. To avoid this we must add HTTP Strict transport security (HSTS)[6]. The mechanism of HSTS is to send a Strict-Transport-Security header to the client specifying when the SSL policy will expire. The browser will then default to HTTPS when communicating with the application until this header expires. This is done by adding the following piece of middleware [6].

```
app.use(function(req, res, next) {
      var aYear = 60 * 60 * 24 * 365;
      res.set(
      'Strict-Transport-Security',
       'max-age='
      + aYear +';includeSubdomains'
      );
      next();
});
```

Here we add the Strict-Transport-Security header with the expiry within a year. Also we specify to include subdomains of the host within the policy. Node.js does not set HTTPS by default and so to prevent this from creating problems we must also add a redirect for any request to the *http* application. This is easier to write using Express as follows:

```
app = express();
app.get('*',function(req, res){
   res.redirect(
    'https://'+req.headers.host+req.url
   );
}).listen(80);
```

Our application now runs TLS over HTTP correctly and our application is free from session hijacking attacks and sniffing.

3.3. Data Validation

3.3.1. Cross-site Scripting (XSS)

A social media web application consists of many modules which require user input. This input is generally sent from client to server by the use of HTML forms to send an HTTP post request. If the inputs of these forms are not checked and filtered, an attacker can execute code on the server side by formatting the input. This is called Cross-Site Scripting (XSS). XSS is of two types; reflected XSS and stored XSS. Reflected XSS occurs when the attacker injects executable JavaScript code into the HTML response with specially crafted links while Stored XSS occurs when the application stores user input which is not correctly filtered. It runs within the user's browser under the privileges of the web application [10].

To add the first layer of protection against XSS we set the HTTP headers *'X-XSS-Protection'* and *'Content-Security-Policy'* to activate protection present within modern browsers. Node.js provides us with the *helmet-csp* which provides an easier way to add a Content Security Policy. [5]

```
var csp = require('helmet-csp');
app.use(csp({
 // Directives
 defaultSrc: ["'self'"],
 scriptSrc: ["'self'", "'unsafe-inline'"],
 styleSrc: ['bootstrap.link'],
 imgSrc: ['img.com', 'data:'],
 sandbox: ['allow-forms', 'allow-scripts'],
 reportUri: '/report-violation',
 // Set to an empty array
 // to allow nothing else through
 objectSrc: [],

 // All browsers report errors
 reportOnly: false,
}));
```

The npm module helmet provides us with a variety of properties to adjust the content security policy easily. As shown above, we can use properties like *defaultSrc*, *scriptSrc*, *styleSrc*, *imgSrc*, *objectSrc* and many more to adjust the policy. [6]

To further the sanitization of inputs, we implemented the *express-sanitizer* library which is a Node.js library for filtering of requests from the client. This library provides us with middleware to escape all request/response values. Thus any user input containing executable code will be sanitized into a regular string.

3.3.2. NOSQL Injection

Many developers have shifted from using relational databases such as PostgreSQL and MySQL to non-relational databases such as MongoDB and Redis. These NoSQL databases allow for rapid scalability and quicker processing of data. However, just like their relational-db system counter parts, they are vulnerable to injection attacks. An example of an injection attack for MongoDB is given below.

```
User.findOne({email:email, password:password})
  .then((user) => {
    if(!user) {
     return Promise.reject(
        'Incorrect email or password'
     );
    } else {
     return Promise.resolve(user);
    }
}).catch((e) => {
  return Promise.reject(e);
});
```

The above snippet shows a mongoose query for finding data in the User schema. This code can be exploited by passing `{"&ne":""}` as the password. This will cause the condition for finding the user to be validated as true and the function will return the user with the matching username.[12]

Practically,

https://.../login?user=Jack&**password[%24ne]=**

This can be avoided by removing the password object from the *findOne* params and by adding *bcryptjs* hashing in the login/registration handlers. Thus we will be able to generate a hash for the password input and compare it to the original value.

3.3.3. Code Injection

Node.js provides many functions called *eval*, *exec*, *spawn*, *fork*, and *Function* that allows the developer to create functions out of string input and execute them [5][6]. Code injection vulnerability might be exploited if the developer has assumed that the user will always put in valid input and has not restricted string input to only valid cases.

In practice, if you have a POST request like:

https://example.com/download?file=user1.txt

In the above example, the request parameter *file* is running the function *exec* (shown below) to interpret what file to return. A user could abuse this and execute arbitrary commands on the

server machine by changing the value of the *file* parameter into something similar to:

https://.../download?file=**user1.txt%3Bcat%20/etc/passwd**

In this example %3B becomes the semicolon, so multiple OS commands can be run.

In Node.js,

```
child_process.exec('ls',
  function(err, data) {
      console.log(data);
  });
```

According to the documentation, *child_process.exec* makes a call to execute /bin/sh, so it is a shell interpreter and does not run individual programs. This is problematic when user input is passed to this method as the attacker can manipulate this string. To overcome this issue we use *child_process.execFile*. The *execFile* function executes the specified script/program and stores the output in the variable *stdout*. We also match the input from the user to unwanted expressions. This is done by using the in-built JavaScript function *match* which allows us to match regular expressions to a string. If any value matches to the specified input then we terminate the request from the user and respond with status 400. An example of the above to solutions is given below:

```
app.post('/download', function (req, res) {
  var file = req.body.file || " ";

  // Test for everything besides
  // alphanumeric, fullstop and -
  if(host.match(/^[-\.]|[^a-zA-Z0-9\-\.]/)) {
   res.status(400).send('Invalid input');
   return;
  }
  execFile('/usr/bin/script', [host],
    function (err, stdout, stderr) {
      if(err || stderr) {
          console.error(err || stderr);
          res.sendStatus(500);
          return;
      }
      res.send(
          '<h3>Downloading link for '+file+':</h3>'+
          '<pre>' + stdout + '</pre>'
      );
   });
});
```

3.4. Error Handling

Due to the single threaded event loop of Node.js, a single unhandled error could cause a crash on the server. Handling of errors in Node.js has been left completely in the user's control. Traditionally this problem would be tackled by implementing a *try-catch* statement wherever fragility with exceptions might occur. But no matter how hard we try, there will still be some errors that are not handled by these *try-catch* statements.

Memory leaks that may accumulate and cause a crash in the process are bound to arise when only catching errors. We study the impact of *forking* of the main process into multiple child processes. When one child process encounters an error that process would be immediately stopped and a new fork would be created. This prevents the resources that have been leaked from accumulating. We look into this by using the *cluster* module in Node.js. [6]

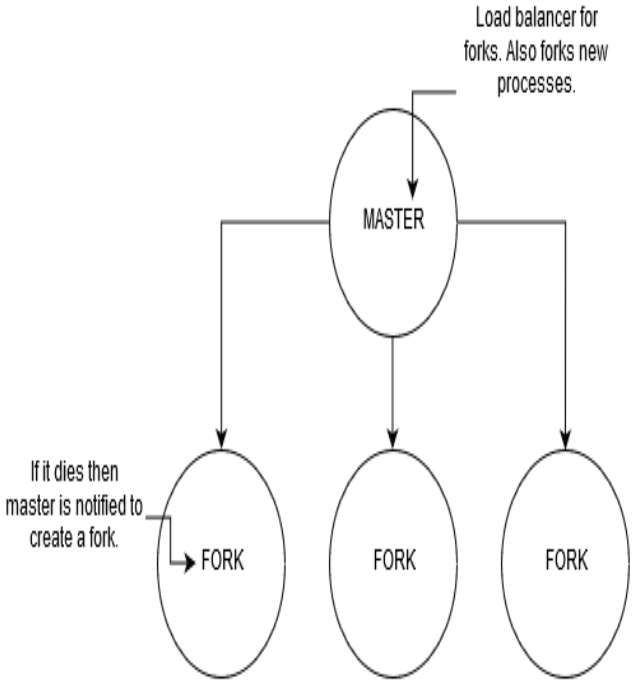

Fig 1.2 Creating forks on the master process

By using this module we are able to create forks depending on the number of CPUs available at the time of execution. Each fork is called a worker. The Node.js Domains API provides a handler for all error instances. Any error that occurs on the server will be sent to the domain *on('error', ...)* handler. However to use it without any loopholes we must combine the use of domains and clusters. Each worker will create its own domain so that we it can catch errors and re-fork correctly.

```
// the worker
var domain = require('domain');
var server = require('https')
.createServer(
 options, // Cert + Key
 function(req, res) {
  var d = domain.create();
  d.on('error', function(er) {
     console.error('error', er.stack);
     try {
        // Close down the cluster
        // in 30 seconds no matter what.
        var killtimer = setTimeout(
        function() {
         process.exit(1);
        }, 30000);
      killtimer.unref();
      server.close();
      cluster.worker.disconnect();
      res.statusCode = 500;
      res.setHeader(
       'content-type',
       'text/plain'
      );
      res.end(An Error occurred! \n');
    } catch (er2) {
      console.error(
       'Error sending 500',
        er2.stack
      );
    }
});
// we need to explicitly add
// req and res to the domain.
d.add(req);
d.add(res);
// handler function in the domain.
d.run(function() {
    handleRequest(req, res);
});
});
```

The above solution provides an overall improvement to the application up-time and security.

4. CONCLUSION

By following a modular approach for eliminating security threats, we have immensely narrowed down the vulnerabilities in our web application. This study shows that Node.js can be used correctly and securely in production environments with regular maintenance and updation of npm modules. We further require to investigate vulnerabilities present in ExpressJS template engine to assess problems with XSS that might occur on the front-end.

REFERENCES

[1] Andres Ojamaa, Karl Duuna. “Assessing the Security of the Node.js Platform”. Proc. of the International Conference for Internet Technology and Secured Transactions. Dec 2012.
[2] Mr. Ninaad Nirgudkar, Ms. Pooja Singh. “The MEAN Stack”. International Research Journal of Engineering and Technology. May 2017..
[3] Arunima Jaiswal, Gaurav Raj, Dheerendra Singh. "Security Testing of Web Applications: Issues and Challenges”. International Journal of Computer Applications. Feb 2014.
[4] L. C. Paulson.“Inductive analysis of the internet protocol TLS”. ACM Trans. Computer Systems Security vol. 2, no. 3, pp. 332–351, 1999.
[5] Gergely Nemeth. “Node.js Security Checklist”. RisingStack Online Blog. Oct 2015.
[6] Karl Duuna. “Secure Your Node.js Web Application.” Dec 2015.
[7] Joyent, Inc. Node.js homepage. [Online]. Available: http://nodejs.org/
[8] J. Wegner. Why Node.JS? Security. [Online]. Available: http://www.wegnerdesign.com/blog/why-node-js-security/
[9] Google, Inc. V8 JavaScript Engine. [Online]. Available: http://code.google.com/p/v8/
[10] PassportJS. [Online] Documentation: http://www.passportjs.org/docs/
[11] Open Web Application Security Project. [Online] https://www.owasp.org/
[12] OWASP: Testing for NOSQL Injection. [Online] https://www.owasp.org/index.php/Testing_for_NoSQL_injection