Preventing NoSQL Injection Attacks on Web Applications Built Using the MongoDB, Express and Node.js (MEN) Stack with the Static Analysis Tool "MEN Injection Guard"

Brian Strickland
Computer Science MS / Cyber Security and Privacy MS
University of Central Florida
Email: Brian.Strickland@knights.ucf.edu

1 Abstract

According to OWASP's Top 10, Injection attacks are the third highest threat that web applications need to defend against [OWA]. Injection attacks aren't a new security risk, but they are one that is commonly overlooked by inexperienced developers. There are various types of injection attacks, but this paper will specifically look at NoSQL attacks against a web application built using the MongoDB, Express and Node.js (MEN) stack.

It is important to help developers write code that will mitigate NoSQL injection attacks because of the potential risks that a vulnerable application can be exposed to. Inexperience developers can benefit from static analysis tools to help them identify where their code might be vulnerable and how to fix it.

According to a survey done by Veracode and DevOps.com [Zor] only 24% of undergraduate students are required to take a course on computer security. This means that the vast majority of junior developers entering the workforce will not be exposed to the topic of NoSQL injections during their academic careers. This gap in knowledge could lead to more vulnerable MEN applications being built, thus exposing data to unauthorized users.

This paper will present a new static analysis tool, MEN Injection Guard, that can help provide guidance to MEN stack developers on where their code may be susceptible to NoSQL injection attacks. MEN Injection Guard will provide developers a way to generate reports on where their code is vulnerable as well as auto generate tests that will fail if their application allows a NoSQL injection attack.

2 Introduction

Protecting developers from creating MEN applications that are vulnerable to NoSQL injection is an important subject to research. As mentioned, only 24% of undergraduate students are required to take courses on computer security [Zor], so it is important to provide junior MEN stack developers a tool that can help them mitigate the risk of their applications becoming victims to NoSQL injection attacks. The research presented in this paper is based around a new tool called MEN Injection Guard, which enables developers to specify a target application to scan and automatically generate test cases for.

MEN Injection Guard attempts to remove any NoSQL vulnerabilities from an application by first informing a developer that their application is vulnerable to NoSQL injection attacks. The tool does this by utilizing a preconfigured set of Fuzz tests, tests that injects invalid, malformed or unsuspected inputs into systems [Syn] to attempt Tautologies, Union Queries and JavaScript Injection attacks. The tool looks at identifying where in a defined route, for a targeted application, where vulnerabilities related to these attacks exists and lets the user know.

Identifying vulnerabilities are helpful, but MEN Injection Guard also looks at a way for developers to keep future NoSQL injection attacks out of their applications by providing automatic generated tests. These tests, which are generated based on current vulnerabilities found, can be used within a software development life cycle (SDLC) to mitigate the risks of NoSQL injection attacks occurring.

This paper will describe the MEN Injection Guard tool and how its components can help developers mitigate NoSQL vulnerabilities in their applications. It will also take a look at the background knowledge needed to understand what NoSQL injection is and the MongoDB, Express and Node.js (MEN) stack which it is built on.

3 Problem Statement

The software security problem that I'm solving is the oversight of allowing unwanted data from users via web applications to get processed by the application. NoSQL injections from web applications is not a new security risk, but it is one that is commonly overlooked by inexperienced developers. Without proper input sanitizing, bad actors can input

data that can trick the system into providing data that they are not authorized to access.

4 Background

MEN Injection Guard is a tool that is specifically designed to help developers build secure MEN applications by letting them know where their application may be vulnerable to NoSQL injection attacks. To fully understand the goals of MEN Injection Guard it is helpful to understand the topics of MEN applications and well as NoSQL injection.

4.1 MongoDB, Express and Node.js (MEN) Applications

MEN applications are made up of three main components; MongoDB, Express and Node.js. The tool is also built using the same web application stack so that users of the tool will be able to study it and make modifications to, if it becomes necessary. To help understand MEN, below is a description of each of the components.

4.1.1 MongoDB

MongoDB is one of the various "Document-oriented" [Wik] NoSQL databases. Data stored in these document-oriented NoSQL databases differ from the traditional relational databases (RDB) as there is no predefined structure to the database. In an RDB, tables are created by a developer that define how data will be stored whereas in a document-oriented NoSQL database, the data is stored as objects that can differ from object to object. For instance, a single collection, user, within a MongoDB database could contain the following objects:

{username: "neo"}
{username: "smith", password: "1234"}

There are several ways to host MongoDB databases; locally, your own cloud or using products such as MongoDB Atlas which is built by the developers that built MongoDB and provides an out of the box solution to hosting and maintaining your MongoDB databases [Atl]. MEN Injection Guard can use either one of these options. During development, a locally ran MongoDB instance was used and the data was viewed using the free MongoDB management tool, Studio 3T Free [Sch].

4.1.2 Express

Express is a framework built on top of Node.js that allows you to make your application code more maintainable and modular. Node.js applications are focused around one single JavaScript function, but Express allows you to break this up into an array of functions by defining routes and middleware. It also provides features such as extensions to the request and response objects and views for dynamically rending HTML [Hah16a]

Middleware is stack that consist of an array of functions which are called from the single Node.js function. Each of these middleware functions are called in sequence from the main Node.js function. A special type of middleware called routing allows a developer to create functions for each POST and/or GET requests that they want to capture instead of using the single function to parse out URL parameters. For example, to load an "about" page and accept a "comment" post, the following code would be used:

Listing 1: Express GET and POST Middleware

```
var app = express();

// Listening for GET request to /about
app.get('about', function(request,
    response){
   response.send("Welcome to about");
});

// Listening POST requests to /comment
app.post('/comment', function(request,
    response){

});
```

Express extends the request and response capabilities of Node.js by adding functions such as redirect and sendFile.

Views provide an easy way to use your favorite view engine such as EJS, Pug, etc to dynamically generate HTML. An example EJS file could look like the following:

Listing 2: Express View Example

```
// From the server.js file
app.set("views", "/views");
app.set("view engine", "ejs");

app.get('about', function(request,
    response){
   response.render('about', {
      whatToSay: "hello there"
   };
});

// /views/about.ejs
<p><%=whatToSay%></p>
```

4.1.3 Node.js

Node.js allows developers to build a HTTP web server with little effort [Dav21] and allows them to create the web server using JavaScript. The JavaScript engine that Node.js uses is based off the V8 JavaScript engine that is used in Google Chrome; this means it is fast [Hah16b]

A simple web server built with Node.js can be created using the following example:

Listing 3: Simple Node.js Web Server

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

```
function serverRequests(request, response){
   response.end("Hello there");
}
http.createServer(serverRequests).listen(3007);
```

4.1.4 Example of a MongoDB, Express and Node.js (MEN) Application

Below is an example of a MEN application. This application first sets up a connection to a MongoDB server listening on localhost:3001. From there it will respond to any request to localhost:3000 and return a JavaScript object containing all users in the "users" collection that is located in the "sqli" MongoDB database.

Listing 4: MEN Applicaiton Example

```
var http = require('http');
var express = require('express');
var app = express();

(async () => {

   // Connect to the MongoDB Server
   var MongoClient =
       require('MongoDB').MongoClient;
   var db = await MongoClient
      .connect("MongoDB://localhost:3001");

   // Middleware to listen for GET requests
   // at http://localhost:3000
   app.get('/', async function(request,
       response){

      // Query a list of all users
      await
          db.db('sqli').collection('users')
         .find().toArray(function(err,
             result) {

         if (err) throw err;

         // Display them to the web page.
         response.send(result);

      });
   });

   // Create a web server
   http.createServer(app).listen(3000);

})();
```

4.2 NoSQL Injection

NoSQL injection is similar to SQL Injection as they both take advantage of using GET and POST requests (i.e. use input) to access information that they are not authorized to access. The main difference between them is the grammar and syntax of a query that they use [Pos]. Both types of these attacks are a subclass of what is called "Code Injection Attacks". Code Injection attacks provide attackers the means to bypass authentication, manipulate data, view data they are not authorized to access and execute commands on the remote machines [Eng13]

There are various classes of NoSQL injection attacks; Tautologies, Union queries, JavaScript injections, Piggy-backed queries, and Origin violation [Avi], but this MEN Injection Guard only looks at the following: Tautologies, Union queries and JavaScript injections on a MongoDB server.

4.2.1 Tautologies

Tautologies utilizes NoSQL verbs such as $gt, $ne, $lt, etc. These are useful in bypassing a simple authentication page. For example, in the following code snippet, if the MEN application is sent a POST request with the following parameters username={"$gt": ""}&password={"$gt": ""}, a successful authentication will occur.

Listing 5: Bypassing Authentication

```
app.post('login', function(request,
    response){

   let query = {
      username = request.body.username,
      password = request.body.password
   }

   db.collection('users').findOne(query,
       function(error, result){
      if(result > 0){
         // Authenticate
      }
   });
});
```

4.2.2 Union queries

Union queries are a type of attack, that as the name suggest, uses the union (i.e. OR) operator to force a query to always be true. These types of attacks can be used to make a page provide more data than it is suppose to [RP]. It can also be used to bypass authentication as well.

An example Union query attack that the Attack Vector Tester tool within MEN Injection Guard showcases uses the following: ' || 'a'=='a An example of this in action can be seen below:

Listing 6: Example Union Query

```
let payload = "'" + request.body.payload +
    "'";

// NoSQL query to find user with the
// provided username for the user
DbConnector.db.collection('users')
   .find({$where: `this.username ==
       ${payload}`});

// The NoSQL query becomes the following
```

```
// with the Union query payload, thus
// returning all users
DbConnector.db.collection('users')
    .find({$where: `this.username == '' ||
        'a' == 'a'`});
```

4.3 JavaScript Injection Attacks

MongoDB databases are vulnerable to JavaScript Injection attacks if they are enabled to accept JavaScript. If input is not sanitized and allowed in a query to a MongoDB database, Denial of Service attacks could be sent to the server with request similar to: myParam=';while(1);' [Ran].

5 Related Work

There are two areas that the work relating to MEN Injection Guard focus on; 1) Identifying NoSQL vulnerabilities and 2) Automatically generating tests for developers to use in the software development life cycle (SDLC).

There already exists many contributions in the area of NoSQL injection vulnerabilities. Ahmed M. Eassa, Omar H. Al-Tarawneh, Hazem M. El-Bakry and Ahmed S. Salama developed a NoSQL Racket application which is a testing tool for detecting NoSQL injection Attacks in Web Applications [Eas+17]. This tool, was developed using PHP, and uses a table of reserved key words to attempt to attack various NoSQL databases. The tool seems to be limited to only the forms built into the to tool though (i.e. it can not attack other websites) MEN Injection Guard enables users to test any web application that they know the URL to.

In the paper, "NoSQL Injection Attack Detection in Web Applications Using RESTful Service", by Ahmed M. Eassa, Mohamed Elhoseny, Hazem M. El-Bakry and Ahmed S. Salama, the researchers developed a REST service that developers of web applications could use to check for NoSQL attacks as requests were made [aEE18]. This research created "DNIARS" which would accept a parameter and then send a response status code (200, 400 or 404) back to let the requester know whether or not the parameter was a NoSQL attacked. This functionality could in theory be used on every request on your web application to determine NoSQL attacks on the fly, however this would add additional requests for application users and add delayed responses. MEN Injection Guard enables developers to test their code for these vulnerabilities before deployment, thus negating the need for a service such as DNIARS.

Other such tools such as Null Sweep [Bel] provides users with a command line tool that they can use to test vulnerabilities against PHP GET Injections. This tool could be useful, however it hasn't been updated since 2020. Null Sweep also is not able to handle POST requests, which MEN Injection Guard is able to do.

Gartner, a service that provides expert guidance on tools, has a list of 20 reviewed, Security Information and Event Management (SIEM) as well [Gar]. However, these tools are not as narrowed focused as MEN Injection Guard so there will be more to learn to use the tool adequately.

In terms of automating generating test cases, the work "A Static Analysis Framework for Detecting SQL" by X. Fu, X. Lu, B. Peltsverger, S. Chen, K. Qian and L. Tao [Fu+07] inspired MEN Injection Guards test generator functionality. This work proposed the creation of a static analysis framework called SAFELI. This framework was built to automatically generate test cases for ASP.NET. However, all the components for SAFELI have not been implemented and since it is only a framework, there isn't a direct way to use it to test web application vulnerabilities or actually generate the test cases like MEN Injection Guard does.

According to the related works, there is nothing that has been created that concentrates prevention of NoSQL injection attacks on the MEN stack. Some of the unique advantages that MEN Injection Guard has over the already existing tools is that it is written in the web application stack, MEN, that it checks for vulnerabilities with. This allows developers to extended and understand the makings of the tool without learning a new language. It also provides an option to auto generate Jasmine tests that a developer can add into their software development life cycle (SDLC). MEN Injection Guard also provides users a way to configure their own fuzz tests to use as well as test their injection attacks on a built in vulnerable MEN application as well as any other publicly hosted application. This allows the user to do more targeted testing.

6 Evaluation

In order to provide developers with vulnerability reports and test code that they could use to mitigate threats from NoSQL injection attacks, a prototype tool MEN Injection Guard was designed and developed. MEN Injection Guard is designed to have three main components; Report Generator, Test Case Generator and the Attack Vector Tool. Before any of these components were built however, Robo3T [Sch] was used to manually craft NoSQL injection attacks so that they could be verified to work. Below is an example of these manually constructed attacks:

Listing 7: Using Robo3T to Build Injection Query

```
// A typical username lookup when browser
// sends request
db.getCollection('users')
    .findOne({"username": 'brian'})

// A valid NoSQL injection attempt by
// the user submitting { "$gt": "" }
// but will fail
db.getCollection('users')
    .findOne({"username": '{ "$gt": "" }'})

// A successful NoSQL injection attempt
// by he user submitting
// ' || { "$gt": "" } || '
db.getCollection('users')
    .findOne({
        "username": '' || { "$gt": "" } || ''
    })
```

6.1 Attack Vector Tester

The Attack Vector Tester is a small utility that enables developers to test various NoSQL injections against vulnerable code. The developer is able to craft their own injection attacks that take advantage of NoSQL vulnerabilities and get immediate feedback on whether or not the attack will work. After each attack submission, the tool will respond with whether or not the attack was successful as well as provide the user with the generated query that was used for the attack.

The tool has the following three attacks available to test using the Attack Vector Tester: Tautologies, Union Queries and JavaScript Injections. Listings 8, 9 and 10 show outputs for each of these types of attacks.

The following Tatutology attack would find all usernames in the users collection:

Listing 8: Tautologies Output

```
User Input: {"$gt": ""}
Tool Output: Success
Query:
dbo.collection('users')
   .find({"username": {"$gt":""}"});
```

This Union Query attack would find all usersnames in the users collection:

Listing 9: Union Query Output

```
User Input: '' || 'a'=='a'
Tool Output: Success
Query:
dbo.collection('users')
   .find({$where: `this.username == '' ||
      'a'=='a'`});
```

The JavaScript Injection attack below would delay the MongoDB database for 10 seconds. This would be useful in a DoS attack:

Listing 10: JavaScript Injection Output

```
User Input: '' || (function(){
   var date=new Date();
   do{curDate = new Date();}
   while(curDate-date<10000);})() || ''

Tool Output: Success

Query:
dbo.collection('users').find({$where:
    `this.username == '' || (function(){ var
    date=new Date(); do{curDate = new
    Date();}while(curDate-date<10000);})()
    || ''`})
```

6.2 Report Generator

The Report Generated essentially looks at a user defined web application route and then iterates over a number of configurable NoSQL injection attacks on the site and finally reports the results to the user. The idea of iterating over various NoSQL injection attacks comes from an idea found in the book 19 Deadly Sins of Software Security [MV05] where a randomSQL subroutine was used to provide random SQL reserved words to inject into a program. However, since MEN Injection Guard is only interested in NoSQL injections, a list of NoSQL fuzz tests, provided by SecLists, was used [Dg].

6.2.1 Fuzz Tests Configuration

The set of fuzz tests that a user wants to use is completely configurable by modifying the config.js file. There is a constant variable called FUZZ_TESTS that the program uses which is listed in Listing 11.

Listing 11: Default Fuzz Tests

```
const FUZZ_TESTS = [
   `{ $ne: 1 }`,
   `', $or: [ {}, { 'a':'a`,
   `' } ], $comment:'successful MongoDB
       injection'`,
   `db.injection.insert({success:1});`,
   `db.injection.insert({success:1});return
       1;db.stores.mapReduce(function() { {
       emit(1,1`,
   `true, $where: '1 == 1'`,
   `, $where: '1 == 1'`,
   `$where: '1 == 1'`,
   `', $where: '1 == 1'`,
   `1, $where: '1 == 1'`,
   `|| 1==1`,
   `' || 'a'=='a`,
   `' && this.password.match(/.*/)//+%00`,
   `' && this.passwordzz.match(/.*/)//+%00`,
   `'%20%26%26%20this.password.match(/.*/)//+%00`,
   `'%20%26%26%20this.passwordzz.match(/.*/)//+%00`,
   `{$gt: ''}`,
   `{"$gt": ""}`,
   `[$ne]=1`,
   `';sleep(5000);`,
   `';it=new%20Date();do{pt=new%20Date();}
      while(pt-it<5000);`,
   `{$nin: [""]}}`
];
```

6.2.2 Attack Target Route Configuration

In order to scan a targeted website, users of the tool define a URL to the target application as well as a route (the path) to attack and what the expected outcome of the route should be. For instance, if a successful login happens, what content would the user see. Refer to Listing 12 for an example TEST_APP_CONFIG setting:

Listing 12: Target App Configuration

```
const TEST_APP_CONFIG = {
   'app-url': 'localhost:3006',
   'routes': {
      'login': {
         'url': 'login',
         'successContent': 'Welcome to your
            profile'
```

```
        }
    }
}
```

MEN Injection Guard provides users a way to set the attack URL, target input types and the success content before a scan is ran. If the configuration isn't changed, the unsecured demo application will be used.

Attack URL: This is the URL that all NoSQL injection tests will run against

Target Inputs: These are a comma separated list of input fields that NoSQL injection data will be sent to. For instance if you want to send NoSQL injection attacks to both <input name="password" /> and <input name="username" /> this value would be set to username,password.

Success Content: This is the content that you would expect to see if an attack was successful.

6.2.3 NoSQL Injection Attack Results

Once all attacks in the configuration file have been iterated through, a response containing the number of failed and successful attacks, along with what the attack was, will be sent back to the users browser window. Below is a mobile display example of a successful scan using the Scanner:

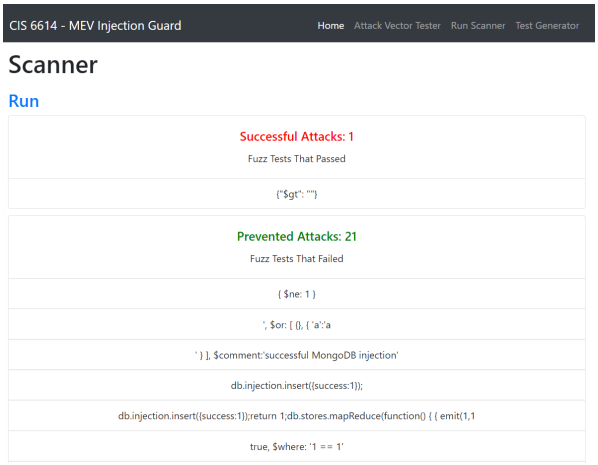

Fig. 1: MEN Injection Guard - Scanner

6.3 Test Case Generator

MEN Injection Guards Test Case Generator provides developers a way to copy and paste Jasmine tests into their project to protect against any future NoSQL injection attack on their vulnerable routes. Jasmine tests [Jas] are built by iterating through each successful attack that occurs on the defined target route and generates a templated Jasmine test for the user. These tests are displayed to the user via the browser. Below is an example test that is auto generated for the 'login' route, which failed to handle a NoSQL attack using {"gt":""}.

6.4 Bad URLs

Both the Report Generator and Test Case Generator rely on real URLs. If a user submits an URL that cannot be resolved, they are informed simply with a "Scanner - Bad URL Request" or "Test Generator - Bad URL Request" afterward.

6.5 Azure and MongoDB Hosting

To enable testing from outside of a local environment, MEN Injection Guard was ported to Azure as an Azure App Service. This required an Azure App Plan Service (compute power) as well as a Azure App Service capable of supporting Node 16 LTS to be created. The Web App was configured to be continually deployed to via GitHub Actions (i.e. whenever a new commit is made to the repository, a new version of the application is deployed to the Azure App Service. Azure App Services were used to host MEN Injection Guard because it supports multi frameworks, such as Node.js, and doesn't require any server setup or maintenance [Mica]

Since MEN Injection Guard no longer lived in a local environment, a cloud based version of MongoDB was also required. A database could of been spun up within Azure, but MongoDB Atlas provides a free out of the box service [Atl]. To make it easy for the Azure App Service running MEN Injection Guard to access the MongoDB Atlas instance, the network access security restrictions were set to 0.0.0.0 so that anything could access it. This decision was made because there are numerous IP addresses associated with an Azure App Service which can change. [Micb]

An additional Azure App Service was also created to host the Unsecured Application that MEN Injection Guard uses as a default attack target.

7 Results

MEN Injection Guard provides an additional layer of protection to developers that may be unaware of how to protect against NoSQL injection attacks. The findings that this tools produced was that it was able to find all vulnerable attacks available within the unsecured application that was built for testing.

When testing with the NoSQL fuzz test data provided by Daniel Miessler, 0% of the "findOne" queries were vulnerable unless the developer implicitly converts the raw request data to a JSON object using JSON.parse(). However, if the developer decides to convert the query with JSON.parse(), the application becomes more vulnerable to NoSQL injection attacks. It was also discovered that while attempting a Tautology attack, the attack must send a NoSQL injection code to each of the inputs required for the query. In the "Bypassing Authentication" example from Listing 5, if only one of the username or password fields doesn't use a successful NoSQL injection grammar, than the attack will fail. This is why MEN Injection Guard enables users to define all target inputs to attack on.

While publishing MEN Injection Guard to Azure, a cloud offering of MongoDB, provided by MongoDB Atlas, was used so that a dedicated server didn't need to be spun up to access a MongoDB server. However, while doing this, it was discovered that MongoDB Atlas adds additional protection against NoSQL injection attacks. Using the same scanning on a locally hosted MongoDB database and an MongoDB Atlas hosted one produced different results. An attack using {"gt":""} was successful when running against the local database, but not successful when running against the cloud hosted version.

8 Conclusion

The development of the prototype tool MEN Injection Guard set out to provide developers a way to test their applications for NoSQL vulnerabilities and to keep them safe while doing additional development. It did this in two ways; 1) by providing developers a way to scan their applications and 2) provide them with automated tests that they could use as part of their software development life cycle.

As it stands, MEN Injection Guard does provides users a way to test their applications, but the tool leaves much to be desired. The application could use improvements around the "Target Application Configuration" settings. For MEN Injection Guard to successfully let you know if a test passes, the user already needs to know what they would see from the results. This may be doable, but perhaps there would be a way to query for normal responses and if the response doesn't fit within normal parameters, count that as a successful attack. Another item that would improve the usability of the publicly available hosted MEN Injection Guard instance would be to allow users to configure which fuzz tests that they want to use. Right now this is only configurable for users that install the tool and run it locally.

However, the tool still fulfills the problem of protecting developers from writing applications that are vulnerable to NoSQL injection. Since they will be testing their own applications, they will know what responses to expect and will get the correct injection vulnerabilities notices.

References

[aEE18] A.M. assa, M. Elhoseny, and H.M. El-Bakry. "NoSQL Injection Attack Detection in Web Applications Using RESTful Service". In: *Program Comput Soft* 44 (2018).

[Atl] MongoDB Atlas. *What is MongoDB Atlas*. URL: https://www.mongodb.com/docs/atlas/.

[Avi] Anton Puzanov Aviv Ron. *Analysis and Mitigation of NoSQL Injections*. URL: https://www.infoq.com/articles/nosql-injections-analysis/.

[Bel] Charlie Belmer. *NoSQLi 0.5.1*. URL: https://nullsweep.com/nosqli-0-5-1-released/.

[Dav21] Ashley Davis. *Bootstrapping Microservices with Docker, Kubernetes and Terraform*. Shelter Island, NY: Manning, 2021. Chap. 1.

[Dg] Jason Haddix Daniel Miessler and g0tmi1k. *SecLists The Pentester's Companion*. URL: https://github.com/danielmiessler/SecLists/blob/master/Fuzzing/Databases/NoSQL.txt.

[Eas+17] Ahmed M. Eassa et al. "NoSQL Racket: A Testing Tool for Detecting NoSQL Injection Attacks in Web Applications". In: *International Journal of Advanced Computer Science and Applications* 8 (2017).

[Eng13] Patrick Engebretson. *The Basics of Hacking and Penetration Testing*. 2nd ed. Waltham, MA: Syngress, 2013. Chap. 6.

[Fu+07] Xiang Fu et al. "A Static Analysis Framework For Detecting SQL Injection Vulnerabilities". In: 1 (2007), pp. 87–96. DOI: 10.1109/COMPSAC.2007.43.

[Gar] Gartner. *Products In Security Information and Event Management (SIEM) Category*. URL: https://www.gartner.com/reviews/market/security-information-event-management.

[Hah16a] Evan M. Hahn. *Express in action*. Shelter Island, NY: Manning, 2016. Chap. 2.

[Hah16b] Evan M. Hahn. *Express in action*. Shelter Island, NY: Manning, 2016. Chap. 1.

[Jas] Jasmine. *Jasmine Behavior-Driven JavaScript*. URL: https://jasmine.github.io/.

[Mica] Microsoft. *App Service Overview*. URL: https://learn.microsoft.com/en-us/azure/app-service/overview.

[Micb] Microsoft. *Inbound and outbound IP addresses in Azure App Service*. URL: https://learn.microsoft.com/en-us/azure/app-service/overview-inbound-outbound-ips#when-outbound-ips-change.

[MV05] David LeBlanc Michael Howard and John Viega. *19 Deadly Sins of Software Security*. Emeryville, California: McGraw-Hll / Osborne, 2005. Chap. 4.

[OWA] OWASP. *A03:2021 - Injection*. URL: https://owasp.org/Top10/A03_2021-Injection/.

[Pos] Howard Poston. *What is NoSQL injection?* URL: https://resources.infosecinstitute.com/topic/what-is-nosql-injection/.

[Ran] RangeForce. *NoSQL Injection*. URL: https://medium.com/rangeforce/nosql-injection-6514a8db29e3.

[RP] Aviv Ron and Anton Puzanov. *Analysis and Mitigation of NoSQL Injections*. URL: https://

www.infoq.com/articles/nosql-injections-analysis/.

[Sch] Dmitry Schetnikovich. *Robo 3T is now Studio 3T Free*. URL: https://robomongo.org/.

[Syn] Synopsys. *Fuzz Testing*. URL: https://www.synopsys.com/glossary/what-is-fuzz-testing.html.

[Wik] Wikipedia. *Document-oriented database*. URL: https://en.wikipedia.org/wiki/Document-oriented_database.

[Zor] John Zorabedian. *Veracode Survey Research Identifies Cybersecurity Skills Gap Causes and Cures*. URL: https://www.veracode.com/blog/security-news/veracode-survey-research-identifies-cybersecurity-skills-gap-causes-and-cures.