Analysis and Attenuation of NoSQL Injection Vulnerability attacks on Web Applications

Pritesh Patil[1], Rutuja Hirve[2], Ankita Urade[3], Rachana Badekar[4], Ashwini Gaikwad[5]
[1, 2, 3, 4, 5] AISSMS Institute of Information Technology

Abstract- *NoSQL data storage systems have become very popular due to their scalability and ease of use. SQL Injection is one of the most widely exploited web application vulnerability of the web 2.0 era. Flexibility and scalability of NoSQL databases are major cause for adoption and popularity. Today is the world of information era, where web applications interact with the back-end database to retrieve data as and when requested by the user. The global exposure of these applications makes them prone to the attacks because of presence of vulnerabilities. These security vulnerabilities continue to infect the web applications through injection attacks. The main mechanism of SQL attacks relevant in NoSQL can be divided into five classes Tautologies, Union queries, JavaScript injections, Piggybacked queries and Origin violation. In our proposed system by observations and providing detailed examples of NoSQL injection attacks we will take actions that needs to mitigate the risks of NoSQL attacks NoSQL JavaScript injection, PHP Tautology injection and NoSQL Union Query injection.*

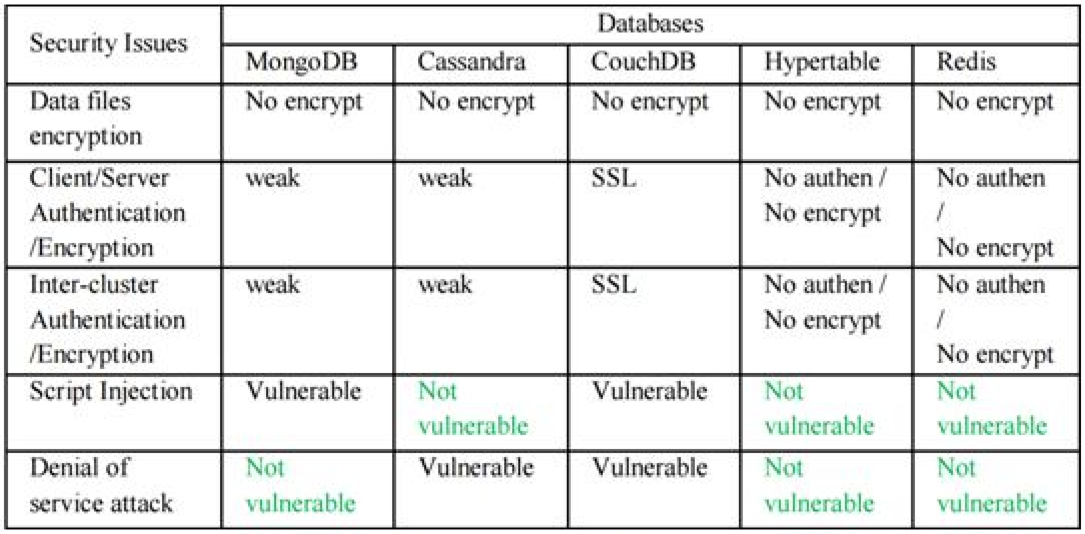
Keywords- NoSQL injection, Mitigation, Security, Web application

I. INTRODUCTION

The need to handle data in web-scale systems, in particular Big Data systems, has led to the creation of numerous NoSQL databases. Database security has been one of the most critical aspects of application security. Recently, the NoSQL databases have become more and more popular because NoSQL databases provide looser consistency restrictions than traditional SQL databases do. NoSQL databases often offer performance and scaling benefits by requiring fewer relational constraints and consistency checks[1]. Some examples are MongoDB [8], and Cassandra [9]. MongoDB can be adapted to all sizes of businesses and individuals since it is open source database. The data pattern can be updated flexibly with the development of the application while providing a secondary index and complete inquiry system. But does that mean NoSQL database systems are immune to injections? As an alternative to traditional relational database, NoSQL is a wide class of database management systems that are not traditional relational database management systems. They do not use SQL language as the primary query language, nor do they typically require fixed table schemas. NoSQL database system allows a user to change data attributes at any time, and data can be added anywhere. There is no need to specify a document design or even a collection up-font. NoSQL databases pay more attention to real-time data processing capability and they are good at directly operation on data access, which greatly promote the development of interactive software system. One of the biggest advantages is the ability to change attributes because of the weakening of structural, so modification process is very convenient.

However, this big advantage also affects its safety where the highest incidence of attacks is injection attacks. Access to a database grants an attacker a dangerous amount of control over the most critical information. For example, SQL injection is a code injection technique that is used to attack data driven applications, in which malicious SQL statements are inserted into an entry field for execution. The malicious user can use SQL commands insert to the Web form submission or enter the domain name to achieve the purpose of tricking the server to execute malicious SQL commands. We all use internet in our daily routine but we are not aware of the fundamental principles of information security and privacy.

Attacks are often confused with vulnerabilities so we must have a clear idea between vulnerability and attack.. By knowing the design loopholes an attacker can submit anNoSQL query directly to the database to get unlimited and unauthorized access. An unauthorized access is the threat to the confidentiality, integrity and authority [14]. Databases are still potentially vulnerable to injection attacks, even if they do not use the traditional SQL syntax, because these NoSQL injection attacks may be executed within a procedural language, rather than in the declarative SQL language such as PHP injection attack and arbitrary JavaScript injection. In this paper we hope to raise the awareness of developers and information security owners to NoSQL security with our examples in both injection and protection.

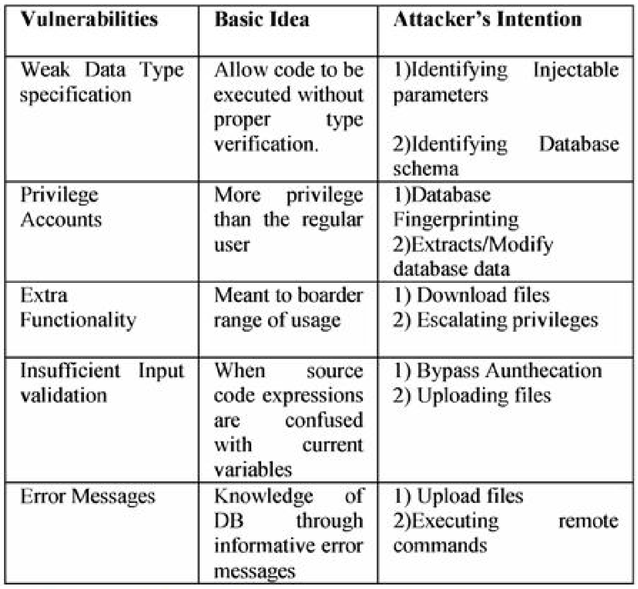


Security Issues	Databases				
	MongoDB	Cassandra	CouchDB	Hypertable	Redis
Data files encryption	No encrypt	No encrypt	No encrypt	No encrypt	No encrypt
Client/Server Authentication /Encryption	weak	weak	SSL	No authen / No encrypt	No authen / No encrypt
Inter-cluster Authentication /Encryption	weak	weak	SSL	No authen / No encrypt	No authen / No encrypt
Script Injection	Vulnerable	Not vulnerable	Vulnerable	Not vulnerable	Not vulnerable
Denial of service attack	Not vulnerable	Vulnerable	Vulnerable	Not vulnerable	Not vulnerable

Figure 1. The Security Comparison of the Top 5 Open Source NoSQL Databases are shown above in the table

II. NOSQL VULNERABILITIES

Vulnerabilities are the weakness, bugs, loopholes, fault or flaw in the existing application[10]. NoSQL is vulnerable the same way SQL databases are vulnerable. Some attacks which are relevant in SQL databases become obsolete in NoSQL databases. However, NoSQL databases does not mean zero risk. NoSQL databases suffer from CSRF(Cross-site Request Forgery)and other vulnerabilities.The web programming languages have vulnerabilities due to some syntax constraints. The poor programming/coding practice leads to vulnerabilities like improper sanitization of inputs, type checking, over privilege accounts and detailed error messages. These loopholes attract the attacker to customize attacks. The attacker can plan a particular attack according to the specific vulnerability present in the application. Besides these vulnerabilities the attacker uses the diversity of SQL language to implement the attacks [3]. The SQLIV are placed according to the Attack intention. The table 1 shows the list of known vulnerabilities, their basic ideas and their corresponding SQL Injection Attacks. Exploitation of these vulnerabilities can be done at input provided by user or at the address bar of the web applications.



Vulnerabilities	Basic Idea	Attacker's Intention
Weak Data Type specification	Allow code to be executed without proper type verification.	1)Identifying Injectable parameters 2)Identifying Database schema
Privilege Accounts	More privilege than the regular user	1)Database Fingerprinting 2)Extracts/Modify database data
Extra Functionality	Meant to boarder range of usage	1) Download files 2) Escalating privileges
Insufficient Input validation	When source code expressions are confused with current variables	1) Bypass Aunthecation 2) Uploading files
Error Messages	Knowledge of DB through informative error messages	1) Upload files 2)Executing remote commands

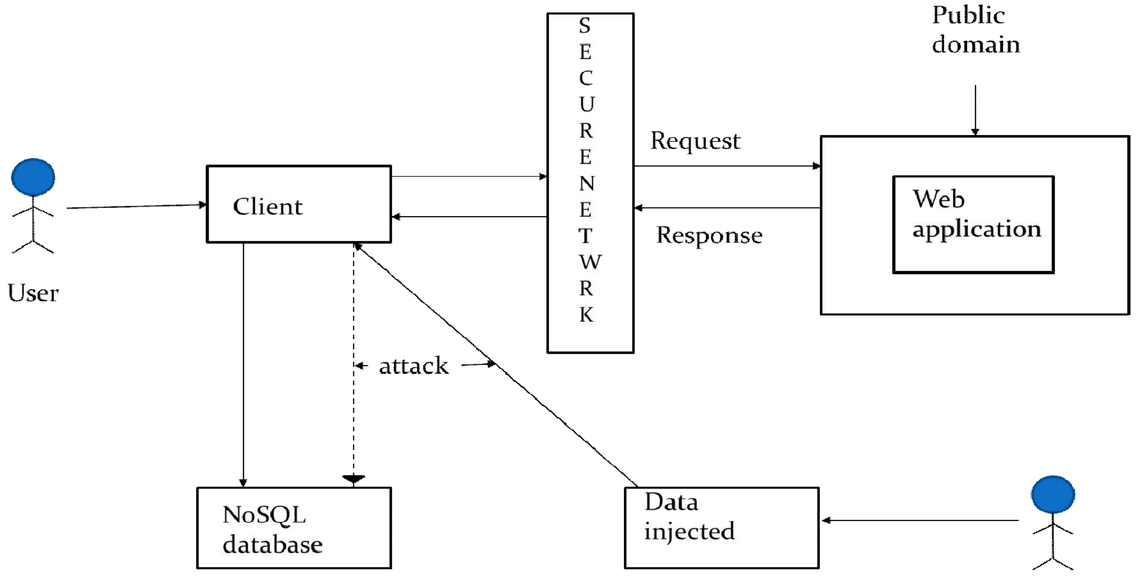
Figure2: This table shows Improper Programing Vulnerabilities[10]

2.1 NoSQL Invasion Course

Many database security threats are caused by database vulnerability. As for injection, there are many methods to learn. Database type, version, and other information are used to identify the motivation for this type of attack. One purpose is to collect the type and structure of the database to prepare for other types of attacks, which can be described as a preparatory step attack. The main mechanisms of SQL attacks relevant in NoSQL can be divided into five classes[1].

1. Tautology: In this type of attack, the injection is by using conditional OR operator so that the will evaluates to TRUE. In this type of attacks normally the user will authenticate and get the data by using WHERE clause. This query transforms all the user information and so the database tables are open to an unauthorized users[4].
2. Piggy Backed Query: This attack is different from others because the attacker will inject some additional queries into the original query or normal query, and so the database will receive multiple queries. Now the first query is executed normally and it is valid. The additional queries are injected queries, which are executed additional to the first. Hence the system is vulnerable to this attack and allows several multiple statements in a single query[4].
3. Union queries: This attack can be done by injecting a UNION query into a vulnerable parameter which will return a dataset.This dataset is the union of the result of original query and the injected query. The SQL UNION operator will combines the result of two or more queries and produce the result set in the UNION from the participating queries.
4. Javascript injections: This is a new class of vulnerabilities introduced by NoSQL databases which allow execution of Javascript in the database context. Javascript allows running complicated transactions and queries on the database engine. Passing unsanitized user input to these queries may allow injecting arbitrary javascript code by an attacker that may result in illegal data extraction or data alteration[4].
5. Origin violation HTTP REST API's are a popular module in NoSQL databases. This introduces a new class of vulnerabilities that allow the attacker to attack the database even from another domain. In cross origin attacks a legitimate user and its web browser are exploited to perform some unwanted action on behalf of the attacker. Here, we show such violations in the form of a Cross-Site Request Forgery (CSRF) attack in which the trust that a site has in a user's browser is exploited to perform an illegal operation on a NoSQL database. By injecting an HTML form into a vulnerable website or tricking a user into a website of his own, an attacker may perform a POST action on the target database, compromising the database[1].

III. SYSTEM DESIGN



3.1 Methodology:

- A customer will register into the online shopping website by entering all his details along with his username and password.
- The intimate details of the client are stored in the backend NoSQL database.
- An attacker will attack the website by using any of the known NoSQL injection attacks like javascript union queries, php tautology etc.
- This will thus compromise the intimate details about the customer which he fed into the website while registering.
- Our web application will thus aim at mitigating these attacks by the use of best practices of code and running the application through a Dynamic Application Security Testing (DAST) or Static Application Security Testing (SAST)[6].

This study presents an experimental evaluation of NoSQL engines dependability based on fault injection. Dependability is an integrating concept that includes the following attributes:

- availability – readiness for correct service.
- reliability – continuity of correct service.
- safety – absence of catastrophic consequences on the user(s) and the environment.
- integrity – absence of improper system alterations.
- maintainability – ability to undergo modifications and repairs. In the context of NoSQL engines, availability and integrity assume a special importance.

The goal of this study is to understand how NoSQL engines perform in the presence of faults, in particular how these faults impact the integrity of the data and the availability of the engine.

IV. NOSQL INJECTION ATTACKS

4.1 NoSQL Union Query Injection

JavaScript also has troubles with NoSQL databases. As compared to php, breaking the query structure, as has been done in SQL injection, is more difficult with a JSON structured query. A typical insert statement in MongoDB could be the following:

db.books.insert({ title: 'Harry Potter', author: 'J. K. Rowling' })

This inserts a new document into the books collection with a title and author field. A typical query could be

db.books.find({ title: 'Harry Potter' })

This is an example of the login form of our web application which will send the username and password via HTTP post to the backend which constructs the query by concatenating strings.

string query = "{ username: '" + post_ username + "', password: '" + post_passport + ' " }"

With (J.K Rowling + Harry Potter), this would build the query:

{ username: 'J.K Rowling', password: 'Harry Potter' }

An example of the malicious input in which the password field turns out worthless is:

username=J.K Rowling', $or: [{}, {'a': 'a&password=' }], $comment: 'successful MongoDB injection'

This input constructes into the query

{ username: 'J.K Rowling', $or: [{}, { 'a': 'a', password '' }], $comment: 'successful MongoDB injection' }

As long as the username is correct, this query suceeds. In SQL terminology, this query is similar to

SELECT * FROM logins WHERE username = 'J.K Rowling' AND (TRUE OR ('a'='a' AND password = ''))
{ username: 'J.K Rowling', $or: [{}, { 'a': 'a', password '' }], $comment: 'successful MongoDB injection' }

This query succeeds because the password part becomes redundant. This attack will succeed in any case in which the username is correct

4.2 PHP Tautology Injections

In this example, a web application is built with a php backend. Php encodes the requests to the JSON format like the array below:

array('title' => 'Harry Potter', 'author' => 'J. K. Rowling');
would be encoded by PHP to the following JSON:
{"title": "Harry Potter", "author": "J. K. Rowling"}

The typical url encoded by php looks like the following:
username=Rowling&password=Harry

The back-end PHP code to process it and query Mongo DB for the user would look like the following:
db->logins->find(array("username"=>$_ POST["username"], "password"=>$_POST["password"]));

This is what the developers intends to do with the query of:
db.logins.find({ username: 'Rowling', password: 'Harry'})

Php has a build-in mechanism which lets attackers send malicious code such as the following:
username[$ne]=1&password[$ne]=1
This input in translated by php into:
array("username" => array("$[ne] " => 1), "password" => array("$ne" => 1));,
which is encoded into the MongoDB query
db.logins.find({ username: {$ne:1 }, password {$ne: 1 })

Because $ne is MongoDB's not equals condition, it queries all entries in the logins collection for which the

username is not equal to 1 and the password is not equal to 1. Thus, this query will return all users in the logins collection. In SQL terminology, this is equivalent to:
SELECT * FROM logins WHERE username <> 1 AND password <> 1

In this scenario, the vulnerability gives attackers a way to log in to the application without valid credentials. In other variants, the vulnerability might lead to illegal data access or privileged actions performed by an unprivileged user. To mitigate this issue, we need to cast the parameters received from the request to the proper type, in this case, using the string
db->logins->find(array(“username”=>(string) $_POST[“username”],“password”=>(string)$_POST[“password”]));

4.3 Cross-Origin Violations:

Another common feature of NoSQL databases is that they can often expose an HTTP REST API that enables database query from client applications. Databases that expose a REST API include MongoDB, CouchDB, and HBase. The exposure of a REST API enables simple exposure of the database to applications—even HTML5 only–based applications—because it terminates the need for a mediate driver and lets any programming language perform HTTP queries on the database. The advantages are clear, but does this feature come with a security risk? We answer this in the affirmative: the REST API exposes the database to CSRF attacks, letting attackers bypass firewalls and other perimeter defenses. As long as a database is deployed in a secure network behind security measures such as firewalls, to compromise the database, attackers must either find a vulnerability that will let them into the secure network or perform an injection that will let them execute arbitrary queries. When a database exposes a REST API inside the secured network, anyone with access to the secured network can perform queries on the database using HTTP only, thus allowing such queries to be initiated from the browser. If attackers can inject an HTML form into a website or trick users into the attackers’ own website, they can perform any post action on the database by submitting the form. Post actions include adding documents. In our research, we inspected Sleepy Mongoose, a full-featured HTTP interface for MongoDB. The Sleepy Mongoose API is defined by the URL as http:// {host name}/{db name}/{collection name}/{action}. Parameters for finding a document can be included as query parameters, and new documents can be added as request data. For example, if we want to add the new document { username: ‘attacker’ } to the collection admins in the database called hr on the safe.internal.db host, we would send a post HTTP request to http://safe .internal.db/hr/admins/_insert with the URL encoded data username=attacker. Now let’s see how a CSRF attack uses this functionality to add a new document to the admins collection, thus adding a new admin user to the hr database (which is located in the supposedly safe internal network), as Figure 5 depicts. For the attack to succeed, a few conditions must be met. First, attackers must have control over a website either of their own or from exploiting a benign, unsecured website. Attackers place an HTML form in the website and a JavaScript that will submit the form automatically, such as
<form action=” http://safe.internal. db/hr/admins/_insert” method=”POST” name=”csrf”><input type=”text” name=”docs” value=” [{&quot;username&quot;:attacker}]” /></form>
<script>document.forms[0].submit(); </script>

Second, attackers must trick users into entering the infected site by means of phishing or inject an infection into a site that users visit regularly. Finally, users must have permissions and access to the Mongoose HTTP interface. In this manner, attackers can perform actions—in this case, inserting new data into the database located in the internal network—without having access to the internal network. This attack is simple to execute but demands that attackers perform reconnaissance to identify the names of the host, database, and so on.

V. MITIGATION

Mitigating security risks in NoSQL deployments is important in light of the attack vectors we presented in this paper.

Let’s examine a few recommendations for each of the threats:

- Prepared Statements: Use prepared statements instead of building dynamic queries using string concatenation.
- Input Validation: Validate inputs to detect malicious values. For NoSQL databases, also validate input types against expected types
- Least Privilege: To minimize the potential damage of a successful injection attack, do not assign DBA or admin type access rights to your application accounts. Similarly minimize the privileges of the operating system account that the database process runs under.
- Strong JSON structure queries.
- Awareness about the use of sanitized input statements.

The mitigation proposed by us will include two phases:

1. Development and Testing:

In this, we consider the threats involved in the software development lifecycle of our online shopping website. The various attacked modules will be mitigated by using the following techniques

i. Using best practices of code like strong JSON structure, proper validation, prepared statement etc.
ii. Looking closely through the design aspects such as what need to be protected and how will this occur.
iii. Spreading awareness among the developers so that they are less likely to portray weaknesses in their code
iv. Running dynamic and static security testing so as to detect the vulnerabilities in code for injection attacks. We will run various test cases to check the performance of the tester.

2. Monitoring and Attack Detection

A look at the importance of adopting intrusion detection systems will be shown.

VI. CONCLUSION

In general, developers should apply defense methods to their NoSQL database systems to prevent injections or any other attacks from happening. In addition, security layers are recommended to be built to keep malicious code away from the systems. We reviewed different attacks which are vulnerable to the database.

In the future work, we plan to examine new injection possibilities and other vulnerabilities particularly on NoSQL databases, platforms and languages, as well as to study how to defend and mitigate attacks so that NoSQL databases are safe to use.

REFERENCES

[1] Aviv Ron, Alexandra Shulman-Peleg, and Anton Puzanov “Analysis and Mitigation of NoSQL Injections”

[2] A Lane, “No SQL and No Security,” blog, 9 Aug. 2011; www.securosis.com/blog/nosql-and-no-security.4.

[3] L. Okman et al. “Security Issues in NoSQL Databases,” Proc. IEEE 10th Int’l Conf. Trust, Security and Privacy in Computing and Communications (TrustCom), 2011, pp. 541–547.

[4] A. Ron, A. Shulman - Peleg and E. Bronshtein, “No SQL, No Injection? Examining NoSQL Security”

[5] Haldar, Vivek, Deepak Chandra, and Michael Franz. "Dynamic taint propagation for Java." Computer Security Applications Conference, 21st Annual. IEEE, 2005.

[6] 9 Advantages of Interactive Application Security Testing (IAST) over Static (SAST) and Dynamic (DAST) Testing http://www1.contrastsecurity.com/blog/9-reasons-why-interactive-toolsare-better-than-static-or-dynamic-tools-regarding-application-security

[7] Least Privilege mitigation to SQL injection https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet#Least_Privilege

[8] MongoDB web site www.mongodb.org

[9] http://cassandra.apache.org

[10] Chandershekhar Sharma, Dr.S.C.Jain “Analysis and Classification of SQL Injection Vulnerabilities and Attacks on Web Applications”