International Journal of Computer Applications (0975 - 8887)
Volume 183 - No.10, June 2021

A Survey on Detection and Prevention of SQL and NoSQL Injection Attack on Server-side Applications

Mehjabeen Shachi[1],Nurnaby Siddiqui Shourav[2],Abu Syeed Sajid Ahmed[3],
Afsana Afrin Brishty[4],Nazmus Sakib[5]
Department of Computer Science and Engineering
Ahsanullah University of Science and Technology, Dhaka-1208, Bangladesh

ABSTRACT

Attacks concerning data can be considered as an intense security threat. A couple of major cyberattacks on eminent database-driven web applications are SQL and NoSQL injection. Confidential data might be revealed to the hacker if the database is injected with malicious codes. Due to inadequate user input validation SQL injection brings a serious threat to the database by leaking proprietary information. Relational and non relational databases are very much vulnerable to these threats. NoSQL database shows higher performance than SQL database regarding efficient storage criteria and data retrieval time. It is flexible for handling big data and is considered to be more secure. Despite these facts and its growing popularity NoSQL databases are also vulnerable to injection attacks. Because of using a different query language, NoSQL injection is irrelevant to traditional SQL injection. Still, SQL and NoSQL injections are quite similar in this sense that both of the attacks rely on suspicious input execution on the server. So, it is a critical issue for non-relational databases as well. In this paper, numerous injection attacks are discussed along with detection and the countermeasures against SQL and NoSQL injection.

Keywords

SQL, NoSQL, injection attack, hacker, detection, prevention

1. INTRODUCTION

Data is so valuable in today's world that it is being called the "new oil" that is powering the modern world. Hackers have become more active since the demand for data has increased like never before. Vulnerabilities that endanger the personal data of users are regularly discovered. Though the technology for protecting valuable and sensitive data is improving constantly, the stealing of important data still exists and is catastrophic. Among all the cyber attacks, the injection attack got the highest rank in the top ten OWASP web applications security risks in 2017. This is a common attack on SQL (Structured Query Language), NoSQL (Not only SQL) databases, OS (Operating System), and LDAP (Lightweight Directory Access Protocol). Although these attacks differ in structure, the basics around them are relatively similar.

In 1970, Edgar F. Codd invented the relational database that arranges data into different rows and columns by associating a specific key for each row. Almost all relational database systems use SQL and are remarkably complex. They are traditionally more rigid and have a limited ability to handle complex data such as unstructured data. But SQL systems are still used extensively and are quite useful for maintaining accurate transactional records, legacy data sources, and numerous other use cases within both small and big organizations.

In 1998, Carlo Strozzi first used the acronym NoSQL while naming his lightweight, open-source "relational" database that did not use SQL. With NoSQL, an application with high performance and scalability can be developed quickly. NoSQL systems handle both structured and unstructured data, but they can also process unstructured Big data quickly. This is why it has become popular with large-scale cloud and web applications like Google, Facebook, Adobe, eBay, Cisco, etc.

Both SQL and NoSQL databases experience injection attacks till today. In 2012, hackers stole more than 450,000 login credentials from Yahoo by exploiting an SQL injection in their servers [1]. With the rise in popularity of NoSQL databases, security threats have also increased. There are some detection and prevention methods against these threats. The approaches usually follow a common pattern which has been shown in Fig-1.

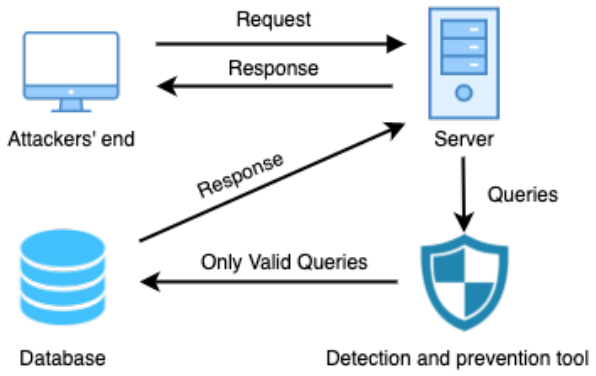

Fig. 1. General pattern of detection and prevention of SQL/NoSQL injection

First, a request is sent from the attackers' end to the server. Then the server sends queries to a database which is needed to serve the

response to that request. The detection and prevention model would stand before queries hit the database.
This study is carried out by analyzing 15 quality papers related to detection and prevention of SQL and NoSQL injection attacks. The discussion on SQL and NoSQL injection is organized in Section II and Section III respectively. In these two sections the types of injection attacks are described followed by some approaches to fix them using ML (Machine Learning) and non-ML approach. Finally, Section IV concludes the paper.

2. SQL INJECTION

2.1 Types of SQL injection

2.1.1 Tautologies.
This is a common injection attack in which hackers manipulate the query in such a way that it always evaluates true after execution. They can log in as admin or even entirely fictional users. After injecting code into a conditional statement, hackers can get access to any information. For example:

```
SELECT * from item WHERE item_id = 123;
```

After injecting "1=1",

```
SELECT * from item WHERE item_id = 123 or
1=1;
```

Here, the hacker can easily access all information about the item with this specific item id.

2.1.2 Piggyback Queries.
In this injection attack one query is implanted into another query. The tailing part contains malicious queries. It is executed as part of the main query. For example:

```
SELECT * FROM item WHERE item_id = 123 or
1=1;
DROP TABLE item;
```

Here, the ';' character denotes the end of one query and the beginning of another.

2.1.3 Alternate Encodings.
In this method hackers combine the function "CHAR" with number coding to return the original character. For instance they use *char(47)* instead of character '/' to avoid filtering of this undesired character.

2.1.4 Illegal or Logically incorrect query.
Hackers write logically incorrect queries that force the system to generate errors with debugging information. This allows the attacker to extract injectable parameters. The query in the example below generates an error message that reveals information about the database [2].

```
SELECT accounts FROM users
WHERE login='' AND pass='' AND
pin= convert (int,(select top 1 name from
sysobjects WHERE xtype='u'))
```

Error Message:

"Microsoft OLE DB Provider for SQL Server (0x80040E07) Error converting nvarchar value 'Cred-itCards' to a column of data type int"

2.1.5 Union query.
Hackers use UNION operators with a typical query to inject malicious query. In the example below, the tailing query is malicious and it can acquire confidential information dodging the authentication process [3].

```
SELECT * from accounts WHERE id='212'
UNION select * from credit_card WHERE
user='admin'--' and pass='pass'
```

2.1.6 Stored Procedure.
The hacker attempts to perform stored procedures present in the database with malicious inputs. Usually, procedures that extend the functionality of the database are stored in the database management system and interaction with them are permitted. These stored procedures are a set of codes that perform some tasks without having to write them every time. They contain some dangerous codes that attackers can exploit to attack the system. For example:

```
CREATE PROCEDURE DBName .is Authenticated
@user Name varchar2, @pass varchar2, @pin
int AS EXEC("SELECT accounts FROM users
WHERE login='" +@user Name + If' and
pass='" +@password+ and pass=" +@pass);
```

The authorized or unauthorized use of stored procedure returns true or false. If the attackers give input as *SHUTDOWN; - -"* for username or password, the Stored Procedure generates the following query statement that shuts down the system [4].

```
SELECT Username FROM userTable WHERE
Username = user1 AND pass=' '; SHUTDOWN;
```

2.1.7 Other Types of Attack.
Besides these stated attacks there are other advanced methods such as Inference, Blind or Time-based, Fast Flux, and Compounded methods.

2.2 Detection and Prevention of SQL injection

2.2.1 ML approach.
The basis of this approach is to model SQLIA (SQL Injection Attack) detection as a data mining-based binary classification problem by constructing the SQLIA detection framework which exploits Support Vector Machine [5]. This framework proposed by Mi-Yeon Kim and Dong Hoon Lee has four phases.
In the data collecting phase, two sets of query trees for the normal class and malicious classes are created.
In the data preprocessing phase, n-dimensional feature vectors are converted from the query trees. From the query trees, syntactic and semantic features are extracted by the feature extractor module. Then it generates multi-dimensional sequences.
In the training phase, some SVM binary classification models are created. Then, through the evaluation of generated models, an optimal binary classification model is selected. With several measure-

ments, the evaluator model estimates the performance of the binary classification models and thus the best SVM model is chosen.

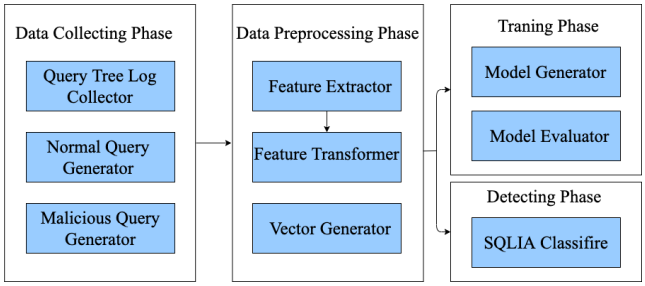

Fig. 2. SQLIA detection framework. [5]

In the detecting phase, testing data is enriched with new inputs. A class label is predicted for the data.
In the data preprocessing phase, an n-dimensional feature vector which is the testing data, is converted from the query tree. By using the optimized SVM binary classification mode, the classifier module classifies the new feature vector as malicious or normal.

2.2.2 non-ML approach.
Some non-ML approaches to detect and prevent SQL injection attack are described below,

2.2.2.1 Static Analysis .
Static analysis is a principle that finds the weaknesses and malicious parts in the system source code prior to reaching the execution stage [6]. This is a technique that is very frequently used by developers for detecting and preventing SQLIA or other vulnerabilities before building and executing the code.

2.2.2.2 Dynamic Analysis.
During runtime a model is generated to detect SQLIA. This detection mechanism executes a query before sending it to database server [7]. As a result SQLIA is detected and prevented before the query reaches the database. One disadvantage of this technique is the overhead involved in generating the model at runtime [8].

2.2.2.3 Combined Approach.
Combined Approach uses advantages of static analysis as well as dynamic analysis to detect and prevent SQLIA [9]. The hotspot is identified in the static phase and then a model is created indicating all the valid queries that can be made at that hotspot [4]. During runtime, the queries are compared with their model. The queries would not be sent to the database for execution if they don't match with their model.

2.2.2.4 Recommended Countermeasures.
There are some preventive measures that should be taken into consideration while building a web application,

—Disable unused features: Disabling every unnecessary feature or function is a way to prevent SQLIA as they might be potentially dangerous.

—Custom error message: malicious code or input might cause the server to generate error messages. These error messages contain information about the database that can be used for hacking. To prevent it from leaking too much Information a custom error message is usually set up.

—Escape functions: The escape functions quickly secures the server against most attacks.

—Prohibit certain keywords: Certain keywords like UNION, DROP and other malicious characters must be filtered because there is a high chance that they are from an injection attack. User input validation is a crucial part of SQLIA prevention.

—Limit the size of the data: Size of user input should be limited. Some injections require a certain number of characters. So limiting size of input data can prevent some SQLIA.

—Use the Prepare Statements: A prepared statement efficiently executes the similar SQL statements. This is the most effective solution for protection against SQL injections. It will consume a little time but will effectively secure the server.

—Query Parameterization: Separate the SQL statement from any kind of parameters that would be included in a call back to the database.

Many other prevention methods such as AMNESIA, SQL Check, SQL Guard, and CANDID have proven successful in SQL injection prevention, but SQL-DOM, SQLrand, AMNESIA, Tainting, SQLCheck, SQLGuard, CANDID are unsuccessful in preventing a Stored Procedure Attack [10].

3. NOSQL INJECTION

3.1 Types of NoSQL injection

The techniques that are used to inject SQL and NoSQL databases are quite relevant. Here, four types of NoSQL injections are demonstrated below.

3.1.1 PHP Tautologies injection.
Like SQL injection attacks, NoSQL also allows bypassing authentication by injecting code in conditional statements and produce expressions that are always true [11]. For example:

Database Type	Query	Injection
MongoDB	`db.logins.find({ username: { $ne: 1 }, password:{ $ne: 1 } })`	`{ $ne: 1 }`
CouchDB	`}POST /users/ find HTTP/1.1 Accept: application/json Content-Type: application/json Host: localhost:5984 { selector": { username: { $ne: null } } }`	`{ "$ne": null`

These queries exposes the entries where username and password are not null. Hackers can utilize the syntax of "$ne" (notequal) operator to login to the system without a proper username and password [12].

3.1.2 Union Queries.
Hacker uses a vulnerable parameter to change the data that was supposed to be returned from a given query. An OR condition has been used to bind an empty expression to the input. Since an empty expression is always valid, it renders the password check ineffective. For example:

Developer's input,

```
string query = username: ‘" + post_username
+ ‘", password: ‘" + post_password + ‘" }"
```

Constructed query,

```
{ username: ‘tolkien’, password: ‘hobbit’ }
```

Malicious input,

```
username=tolkien’,$or: [ {}, {
‘a’:’a&password=’} ], $comment:’successful
MongoDB injection’
```

Constructed query,

```
{ username: ‘tolkien’, $or: [ {}, { ‘a’
: ‘a’ , password: ‘’ } ], $comment:
‘successful MongoDB injection’ }
```

Here the empty query {} is always true [11].

3.1.3 Javascript injections.
NoSQL databases permit the execution of Javascript code and run complicated queries and transactions on the database engine. If the user input is not filtered or validated, there might be a risk of injection of random javascript code.

3.1.4 Piggy-backed Queries.
Here a hacker uses escape sequences and special characters like carriage return [CR], line feed [LF], closing braces, semicolons to end a query and then insert additional malicious queries to be executed. It can mess up the database immensely. For example,
Query:

```
db.doc.find({ username: ‘G. R. R.
Martin’});
db.dropDatabase();
db.insert({username: ‘dummy ’,password:
‘dummy ’})
```

Original query:

```
db.doc.find({ username: ‘G. R. R. Martin’})
```

Injection:

```
; db.dropDatabase();
db.insert({username: ‘dummy ’, password:
‘dummy’})
```

Here, after a semicolon, additional malicious query is injected by the hacker [12].

3.1.5 Origin violation.
HTTP REST API brings in a new class of vulnerabilities that allow the attacker to attack the NoSQL database even from another domain. In cross origin attacks an authorized user and its web browser are exploited to execute some unwanted action for the hacker. In the form of a Cross-Site Request Forgery (CSRF), this attack takes place when the trust that a site has in a user's browser is exploited to execute an illegal operation on a NoSQL database. By injecting an HTML form into a vulnerable website or deceiving a user into hacker's personal website, the hacker may execute a POST action on the database.
There are also Tor Browser Attack, Cross-site Injection Attack, EXE File Upload Attack [13].

3.2 MongoDB and NoSQL injection

Among all the 340 database systems, MongoDB is nominated as the most popular NoSQL database according to DB-Engines tracking [14]. MongoDB is an open-source NoSQL database that is written in C++. It is a document-based database. The document is named BSON(Binary JSON) which is similar to the JSON format. MongoDB stores these BSON formatted documents inside a collection. It uses the Map-reduce data processing paradigm to condense large amounts of data into useful aggregated results. Map-reduce is like a group by operation of SQL. Unlike relational databases, there is no statically typed schema in MongoDB which is why we call it schema-less. So each document in a collection can possess different attributes. Though MongoDB does not support traditional SQL queries the way MySQL does, it has document querying by which we can run SQL to find data less than or greater than a specific value or use regular expressions for pattern matching. MongoDB can be scaled within and across multiple distributed data centers with good performance, which makes it more preferable to other relational databases like MySQL.
MongoDB has some drawbacks in its design [15]. Some of the security issues of MongoDB are described below:

3.2.1 Hash Injections.
Though Hash injection attacks are not that severe as SQL injection attacks. But still, it can do authentication bypass, DOS attacks, and data leakage attacks.

3.2.2 Fast Password Hashing.
MongoDB uses a very fast MD5 password hashing algorithm but this fastness is not good for password hash calculation.

3.2.3 Authentication.
There is no authentication when MongoDB is used in shared mode. A pre-shared string works to authenticate in replica mode.

3.2.4 No Encryption.
Currently there is no data encryption in MongoDB.

3.2.5 Clear text data.
Since there is no encryption mechanism, MongoDB data files are stored as clear text. If any unauthorized user gets access to the database, they can gain access to the valuable information very easily.

3.2.6 Salt Reuse.
MongoDB uses the same salt which is "mongo" for all the users while calculating the hash of the password. This causes a problem of hash collisions and it can ease the hackers.

3.3 Detection and Prevention of NoSQL injection

3.3.1 ML approach.
An ML model was developed to detect NoSQL injection using feature-based supervised learning in this approach [12]. They created their own dataset of benign and malicious MongoDB queries by using all the available resources like OWASP, MongoDB manual, etc. Because there was no available labeled dataset for NoSQL injection. In their dataset, they worked on PHP array injection, NoSQL OR injection, JavaScript-based injection, and piggybacked queries. These types are already discussed in section 2.1.

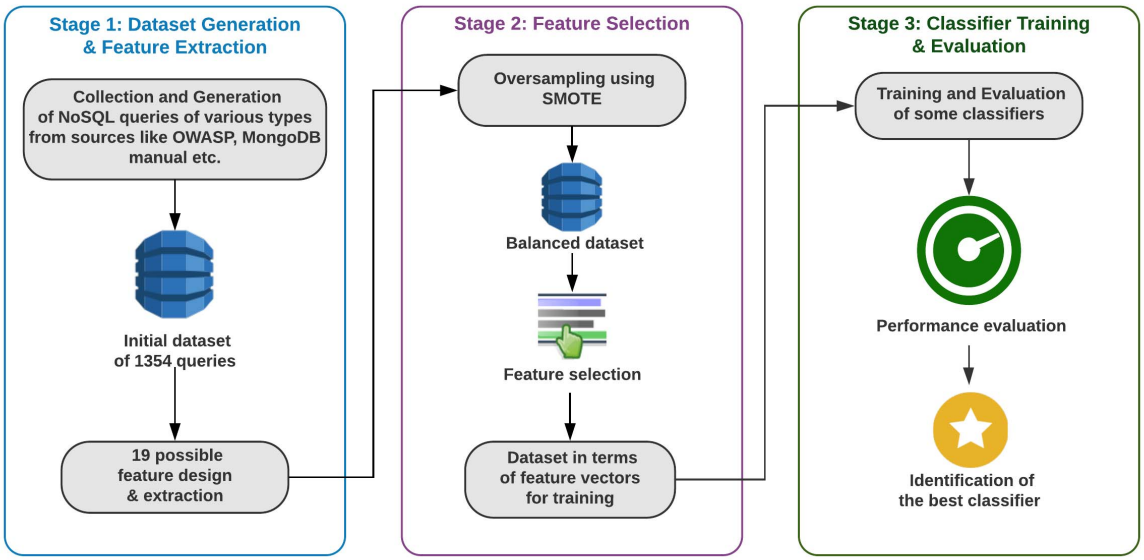

Fig. 3. Conceptual model of ML based detection model[12]

They used WEKA's ClassifiifierSubsetEval with J48(decision tree), IBK(k nearest neighbor) classifiers, and greedy stepwise search with backward elimination to chalk out the 10 highest ranked features for their model considering information gain and correlation [16][17][18]. Those selected features are mentioned in Table 1.

Table 1. Features of NoSQL injection dataset by information gain and correlation [19].

Rank	Features
1	Contains Comparison
2	New Query
3	Contains Empty String
4	Contains Not Equal
5	Contains Payload
6	Presence of Return
7	Always True Expression
8	Evaluation Query Operation
9	Contains Logical Operator
10	Element Query Operation

Using the 10 features mentioned in Table 1 they worked on Binary classification where the two classes are Benign and Injection. They used learning classifiers like decision tree-based ID3 algorithm, artificial neural network, with backpropagation, random forest AdaBoost, k nearest neighbor (IBk), support vector machines (SVM) and XGBoost etc [19][20][21][22][23][24]. They used 10-fold cross-validation to evaluate the performance of the classifiers [25]. After that, they used a NoSQL injection generation tool named NoSQLMap to create a test dataset [26]. NoSQLMap is not used to generate their original dataset. Using this test dataset filled with NoSQL injection they tested their model. NoSQLMap provides 4 vulnerable web applications which were tested by both Sqreen, the only available NoSQL injection detection tool and their proposed method [27]. Their method outperforms Sqreen by 36.25% which means the detection rate of their method was 36.25% higher on average than Sqreen.

3.3.2 non-ML approach.

Some non-ML approaches to detect and prevent NoSQL injection attack are described below,

3.3.2.1 Automata.

In this approach an Automata-based detection model is created for NoSQL injection [28]. Mainly time-based and blind-based boolean injection attacks are focused here.
In time-based injection, the hacker tries to append a javascript function along with a valid NoSQL token. By doing so, the attacker puts the database on hold. The below query would put the MongoDB database on hold for 5 seconds if it gets an entry as "John".

```
John',$where: 'function(){ sleep(5000);
return this.name ==\John'" }
```

In Blind based boolean injection attackers make use of MongoDB functionalities to access a list of collections, number of collections etc.

```
return (db.getCollectionNames().length ==
1);
return(tojsononeline
(db.collectionname.find()[0]).length == 1);
return(db.getCollectionNames() [0][0] ==
'a');
```

To detect and prevent NoSQL injection, first, they identify the points in the source code from where database calls are made. Normally in MongoDB find(), insert(), remove() and update() are those

points. Then they build some NFA models for valid or safe queries that can be generated from those identified hotspots using the JSA (Java String Analyzer) library [29].

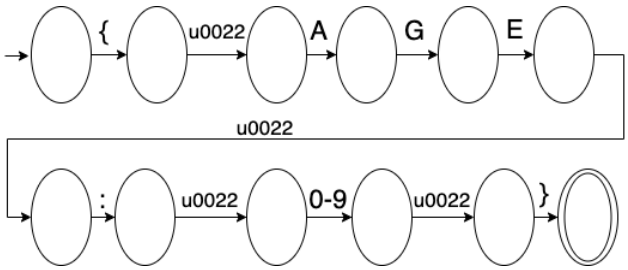

Fig. 4. Sample model for MongoDB query "AGE": "", where "" is any number[24]

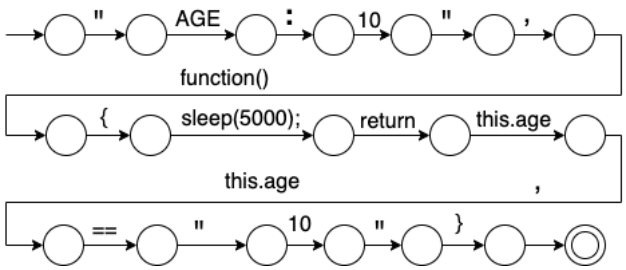

Fig. 5. Sample model for an attack intended user input[24]

After that, they create some dynamic queries when the user gives an input. If those queries match the pre-prepared NFA models, the query is recognized as valid or safe for their system. Only valid queries are allowed to hit the database. In fig 5 we can see that an invalid input has been passed. Now if we compare fig 4 and fig 5, we will notice that the models are not matching. So the query of fig 4 would not be allowed to hit the database. So this is how their automata-based NoSQL injection detection and prevention solution works.

3.3.2.2 User input validation.

Developers take several precautions while building the system to detect and prevent SQLIA. For example, in MongoDB input boxes are limited by adding the following code[23],

```
onkeypress = return"event.keyCode>=
48&&event.keyCode<=57"
```

This only accepts numbers. Notations, space or some specific characters are also checked and filtered to avoid malicious code injection.

3.3.2.3 Parameterization.

The user input variables must not be inserted straight into the condition statement and should be filtered. In parameterization, parameterized statements are used to pass input variables. Rather than using embedding user input variables into the condition statement it uses parameters. The code is shown below,

```
if(is_numeric($usearchtwo)=="true"){} else
echo "Incorrect.";
```

This piece of code checks if the query contains any number and passes the value if it does [23].

3.3.2.4 Malicious feature detection.

Malicious feature detection is to detect if the system or software

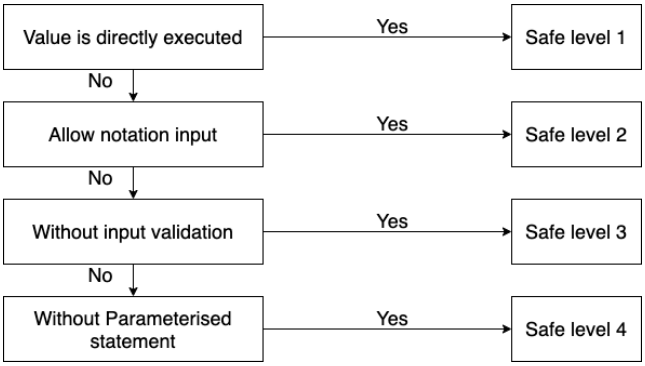

Fig. 6. Flowchart of JavaScript code for limiting input [23]

has some features which are dangerous for security. Based on some malicious code and features. This detection can help developers detect their safety level in their work. The higher the safe level number is, the more secure the NoSQL database is.

4. CONCLUSION

Although there exist many technologies to tackle software security threats, SQL and NoSQL injection attack is still a severe security threat for web applications. This study has shown that NoSQL databases also face similar kinds of vulnerability attacks like SQL does and also shown different techniques to detect and prevent them. Since hackers are becoming more innovative over time, detecting and preventing SQL and NoSQL injection attacks more efficiently and accurately has become an urgent need. It is expected that web applications will be free from injection attacks if the developer follows the approaches discussed in this study.

5. REFERENCES

[1] G. Keizer, "Yahoo fifixes password-pilfering bug, explains who's at risk,"2012

[2] W. G.J. Halfond, Jeremy Viegas, and Alessandro Orso. A Classification of SQL Injection Attacks and Countermeasures. College of Computing, Georgia Institute of Technology, 2006.

[3] A. Alazab, Moutaz Alazab, Jemal Abawajy, Michael Hobbs. Web Application Protection against SQL Injection Attack. The 7th International Conference on Information Technology and Applications, pp. 1-7, ICITA 2011

[4] Ghafarian, A. A hybrid method for detection and prevention of SQL injection attacks. , 2017 Computing Conference. (2017)

[5] Kim, M.-Y. and Lee, D.H. Data-mining based SQL injection attack detection using internal query trees. , Expert Systems with Applications, 41. (2014) , 5416–5430

[6] Lashkaripour, Z. and Ghaemi Bafghi, A. A security analysis tool for web application reinforcement against SQL injection attacks (SQLIAs). , 2013 10th International ISC Conference on Information Security and Cryptology (ISCISC). (2013)

[7] Shrivastava, G. and Pathak, K. SQL Injection Attacks: Technique and Prevention Mechanism. , International Journal of Computer Applications, 69. (2013) , 35–39

[8] Website. . [Online]. Available: D. Box and A. Hejlsberg. LINQ: .NET Language-Integrated Query. https://msdn.microsoft.com/en-us/library/bb308959.aspx. [Accessed: 04-May-2021]

[9] Lee, I. et al. A novel method for SQL injection attack detection based on removing SQL query attribute values. , Mathematical and Computer Modelling, 55. (2012) , 58–68

[10] Alsobhi, H. and Alshareef, R. SQL Injection Countermeasures Methods. , 2020 International Conference on Computing and Information Technology (ICCIT-1441). (2020)

[11] Ron, A. et al. Analysis and Mitigation of NoSQL Injections. , IEEE Security Privacy, 14. (2016) , 30–39

[12] Islam, M.R.U. et al. Automatic Detection of NoSQL Injection Using Supervised Learning. , 2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC). (2019)

[13] Abdalla, H.B. et al. NoSQL Injection: Data Security on Web Vulnerability. , International Journal of Security and Its Applications, 10. (2016) , 55–64

[14] Singh, S. Security Analysis of MongoDB. .

[15] Palvi Aggarwa and Rinkle Rani, Security Issues and User Authentication in MongoDB,Emerging Research in Computing, INformation, Communication and Applications (2014)

[16] Moore, A.W. and Lee, M.S. Efficient Algorithms for Minimizing Cross Validation Error. , Machine Learning Proceedings 1994. (1994) , 190–198

[17] Ross Quinlan, J. (1993) C4.5: Programs for Machine Learning, Morgan Kaufmann.

[18] Aha, D.W. et al. Instance-based learning algorithms. , Machine Learning, 6. (1991) , 37–66

[19] Jin, C. et al. An improved ID3 decision tree algorithm. , 2009 4th International Conference on Computer Science & Education. (2009)

[20] Hopfield, J.J. Artificial neural networks. , IEEE Circuits and Devices Magazine, 4. (1988) , 3–10

[21] Ho, T.K. Random decision forests. , Proceedings of 3rd International Conference on Document Analysis and Recognition.

[22] Freund, Y. and Schapire, R.E. A desicion-theoretic generalization of on-line learning and an application to boosting. , Lecture Notes in Computer Science. (1995) , 23–37

[23] M., A. et al. NoSQL Racket: A Testing Tool for Detecting NoSQL Injection Attacks in Web Applications. , International Journal of Advanced Computer Science and Applications, 8. (2017)

[24] Chen, T. and Guestrin, C. XGBoost. , Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. (2016)

[25] https://www.openml.org/a/estimation-procedures/7

[26] https://github.com/codingo/NoSQLMap

[27] Sqreen, "Web application and user protection," https://www.sqreen.io.

[28] Joseph, S. and Jevitha, K.P. An Automata Based Approach for the Prevention of NoSQL Injections. , Communications in Computer and Information Science. (2015) , 538–546

[29] Feldthaus, A., Miller, A.: Java String Analyzer. http://www.brics.dk/JSA/

[30] Hou, B. et al. MongoDB NoSQL Injection Analysis and Detection. , 2016 IEEE 3rd International Conference on Cyber Security and Cloud Computing (CSCloud). (2016)