Comparing NoSQL and SQL Database Systems Based on Vulnerability to Injection and Adequacy of Countermeasures

Divyaa Kamalanathan, Rahul Gomes
Department of Computer Science and Mathematics
Minot State University
Minot 58701
divyaa.kamalanathan@ndus.edu, rahul.gomes@ndus.edu

Abstract

Databases are an integral part of the internet since storage of data is important for applications or websites. Hence, it is of the highest priority that these databases are kept as secure as possible.

SQL Injections (SQLi) [11] involve the permeation of SQL databases, such as MySQL, with the use of strings containing SQL keywords being injected into queries, usually through an online form. NoSQL Injections (NoSQLi) are similar, but they are used to permeate NoSQL databases such as MongoDB

To fully understand the threat that SQLi and NoSQLi pose to their respective databases, two identical websites were created, one using a MySQL database and the other using a MongoDB. Attacks were made using an online login form to these websites. Different injection techniques were used on both websites and relative ease at which the attack took place were documented. Without proper measures, i.e. the sanitization of strings, both were equally susceptible and that simple countermeasures could be used to prevent a wide array of attacks.

1 Introduction

We live in a world where Big Data has become a commonplace term, used to describe the vast amounts of data that is being mined and utilized in almost every aspect of our everyday lives. Hence, there has been increasing salience in securing this information to prevent serious ramifications to not only large corporations and institutions but also to individuals. This is why, according to OWASP (the *Open Web Application Security Project*), the biggest application security risk is Injection[1]. Injection is used to describe the act of sending untrusted data to an interpreter in the form of a command or query. The most common form of this is SQL Injection (SQLI) wherein "data provided by the user is included in a SQL query in such a way that part of the user's input is treated as SQL code" [2]. However, the advent of NoSQL systems such as MongoDB and Cassandra has also lead to the rising use of NoSQL Injection (NoSQLI) attacks.

In this paper, several SQL and NoSQL injection vulnerabilities, the methods used to exploit these vulnerabilities, and appropriate countermeasures for each are presented. MongoDB is used as the system to demonstrate NoSQL injection attacks and MySQL is used as the system to demonstrate SQL injection attacks. The databases used to examine both of these systems will be identical, in terms of the data that they contain. Then, this data will be compared and contrasted to provide an overview of the security of each database system in comparison with each other, in terms of ease of access and availability of countermeasures.

2 Background

2.1 SQL Injection Background

By definition, SQL (Structure Query Language) is "the standard means of manipulating and querying data in relational databases" [9]. It is the structure behind the majority of relational databases in use, i.e. databases that use tables which are connected to each other through their relationships. There are a variety of database management systems that utilize some variation of SQL, e.g. MySQL and MariaDB. SQL Databases are often selected due to the ability to link data to each other in a uniform way. SQL Injection attacks occur when input mechanisms present in the application or website are used malevolently. According to Halfond et. al. the main cause of SQL Injections is the insufficient validation of user input [2] which could be easily prevented using coding guidelines designed to reduce the risk of SQLIs.

2.2 NoSQL Injection Background

NoSQL (Not Only SQL) is another type of database management language that differs from SQL in that it is not based on a relational model but rather, it uses documents that do not follow a rigid structure (or schema). Each document contains data of any type and can be of any size, every document in a collection in a database could potentially be completely different from each other. NoSQL-based systems, including MongoDB and Apache CouchDB, are usually selected due to their flexibility and scalability [7]. NoSQL databases are said to be less susceptible to injections because the data in them are less

likely to be stored as traditional strings as they are commonly stored as objects. MongoDB builds BSON objects instead of strings during query assembly [6] . This does not mean it is unlikely that NoSQL Injections (NoSQLIs) would occur as on many occasions, the back-end uses Javascript or PHP models which accept strings as input.

2.3 Types of Attacks

2.3.1 SQLI Attacks

i. Tautology Attack

Used for: Bypassing authentication, extracting information
Description: Tautologies are statements that are always true. Statements such as 1=1 are tautologies and can be inserted into query input usually using the OR keyword to get past queries that require a condition to be fulfilled. This is the most commonly used method of SQLI attack [12].

ii. Union Attack

Used for: Bypassing authentication, extracting information
Description: Similar to tautology attacks, a UNION keyword is added to the original intended input along with an additional query to gain further information and to bypass conditions

iii. Illegal/Logically-Incorrect Queries
Used for: Identifying injectable parameters, performing database finger-printing, extracting data, performing denial of service
Description: This is usually used as a precursor to another attack because it allows an attacker to gain information about the structure of the database. A piece of information is input to be used as part of a query that causes an error to occur, this could be a syntax, type conversion, or logical error. The JavaScript console log, PHP alerts, and other error logging mechanisms are used to reverse-engineer the database structure so that a second more-knowledgeable attack can be conducted. It can also be used to perform denial of service attacks as it disrupts the connection.

2.3.2 NoSQLI Attacks

i. Union Attack

Used for: Bypassing authentication, extracting information
Description: Similar to tautology attacks, a UNION keyword is added to the original intended input along with an additional query to gain further information and to bypass conditions. Similar to the SQLi Union Attack.

ii. PHP Array Injections

Used For: Bypassing authentication, extracting data, illegal elevated access

Description: Before the query is converted into the JSON format by the MongoDB system, the PHP code used in the back-end of a webpage process the query in a different format allowing for mistranslations in which malicious input could be entered into the front end which is seen as innocuous by the PHP back-end but becomes malicious when translated into JSON.

2.4 Countermeasures

2.4.1 Prepared Statements

The primary cause of SQLI is input that has not been validated. In order to prevent malicious input from being processed, need to use a function like:
filter_input (type , variable name , filter function)
which takes a variable and applies a filter onto it, which could eliminate special characters or other undesired elements

2.4.1 Sanitized Input

Another method to prevent any undesired elements from being processed with the query would be to use:
mysqli_real_escape_string(connection,escapestring)
which removes all special characters from a string.

3 Methodology

3.1. Database Creation

In order to test the permeability of NoSQL and SQL database systems to injection attacks, four identical webpages were developed using the WAMP developer environment. For each type of database system, there were two pages: a login form where user input was used to assemble a query that was utilized in the second page which took the login information and displayed some information about the user (first name and last name). A MySQL-based database system is natively part of the WAMP was used as the database system for one of the webpages to test out SQLI attacks and countermeasures. MongoDB's PHP driver extension was used to host the database on WAMP for the other webpages to test out NoSQLI attacks and countermeasures. For the purposes of this experiment, the password was entered and stored in plaintext. The same data was stored in both databases identically. Figure 1 is a representation of the data.

login

username	password	fname	lname
hello	world	Walter	White

creditcards

cardnumber	pin	name
12345	123	Walter White

Figure 1: Representation of the data in each table

Figure 2(a) and 2(b) shows the commands used to store the data into the MySQL database and the MongoDB database, respectively:

```
1.  DROP TABLE IF EXISTS `login`;
2.  CREATE TABLE IF NOT EXISTS `login` (
3.  `username` varchar(10) NOT NULL,
4.  `password` varchar(10) NOT NULL,
5.  `fname` varchar(10) NOT NULL,
6.  `lname` varchar(10) NOT NULL
7.  ) ENGINE=MyISAM DEFAULT
    CHARSET=latin1;

8.  INSERT INTO `login` (`username`, `password`,
    `fname`, `lname`) VALUES
9.  ('hello', 'world', 'Walter', 'White');
10. COMMIT;

11. DROP TABLE IF EXISTS `creditcards`;
12. CREATE TABLE IF NOT EXISTS `creditcards` (
13. `cardnumber` varchar(20) NOT NULL,
14. `pin` varchar(20) NOT NULL,
15. `name` varchar(40) NOT NULL
16. ) ENGINE=MyISAM DEFAULT
    CHARSET=latin1;

17. INSERT INTO `creditcards` (`cardnumber`, `pin`,
    `name`) VALUES
18. ('12345', '123', 'Walter White');
```

Figure 2(a): Commands used to store data in MySQL Database

```
1.  use test
2.  switched to db test
3.  db.createCollection("login");
4.  { "ok" : 1 }
5.  >db.login.insert({username:"hello",password:"world",fname:"Walter",
    lname:"White"});
6.  WriteResult({ "nInserted" : 1 })
7.  db.login.find().pretty();
8.  {
9.  "_id" : ObjectId("5c0f2e1170794a2722428ea4"),
10. "username" : "hello",
11. "password" : "world",
12. "fname" : "Walter",
13. "lname" : "White"
14. }
15. db.createCollection("creditcards");
16. { "ok" : 1 }
17. db.creditcards.insert({cardnumber: "12345",pin:"123",name:"Walter
    White"});
18. WriteResult({ "nInserted" : 1 })
19. db.creditcards.find().pretty();
20. {
21. "_id" : ObjectId("5c0fb9f470794a2722428ea5"),
22. "cardnumber" : "12345",
23. "pin" : "123",
24. "name" : "Walter White"
25. }
```

Figure 2(b): Commands used to store data in MongoDB Database

We explain the significance of commands in the lines of figure 2(a):

1. If a table called “login” already exists, it is deleted
2. A new table is created called “login” with the attributes:
3. “username” which has 10 characters and cannot be left empty,
4. “password” which has 10 characters and cannot be left empty,
5. “fname” which has 10 characters and cannot be left empty
6. “lname” which has 10 characters and cannot be left empty
7. The engine used is MyISAM with the default character set being the latin1 character set
8. Values for the attributes in login are inserted

9. The value for “username” is “hello”, for “password” is “world”, for “fname” is “Walter” and for “lname” is “White”
10. These changes are committed to the database
11. If a table called “creditcards” already exists, it is deleted
12. A new table called “creditcards” is created with the attributes:
13. “cardnumber” which has 20 characters and cannot be left empty,
14. “pin” which has 20 characters and cannot be left empty,
15. and “name” which has 40 characters and cannot be left empty
16. The engine used is MyISAM with the default character set being the latin1 character set
17. Values for the attributes in “creditcards” are inserted
18. The value for “cardnumber” is “12345”, for “pin” is “123”, for “name” is “Walter White”

Similarly, the significance of lines in Figure 2(b) is discussed below.
1-2. The database currently in use is switched to the database called “test”
3. A table (or collection) called “login” is created
5. The “login” table is populated with the attribute username with the value “hello”, the attribute “password” with the value “world”, the attribute “fname” with the value “Walter”, and the attribute “lname” with the value “White”
15. Another collection called “creditcards” is created
17. The “creditcards” table is populated with the attribute “cardnumber” with the value “12345”, the attribute pin with the value “123”, and the attribute “name” with the value “Walter White”.

3.2. Login Form

Next, a page that serves as a login form is created, linked to the “login” table for the MySQL database and the “login” collection for the MongoDB database. Figure 3 shows the login form page code for both the SQL and NoSQL database types.

```
1.  <html>
2.  <head>
3.  <title>Test</title>
4.  </head>
5.  <body>
6.  <h2>Test Form</h2>
7.  <form method="post" action="test.php">
8.  Username:  //username input
9.  <input type="text" name="username" size="20">
10. <br>
11. Password: //password input
12. <input type="text" name="password" size="20">
13. <input type="submit" value="Submit">
14. </form>
15. </body>
16. </html>
```

Figure 3: The code for the form page

The form is a basic HTML form which uses the POST form method and sends the information collected in the form to the next page called “test.php”. The form contains two fields: a text field called “username” and a password field called “password”, as well

as a submit button which appeared as a login form on the website as shown in Figure 5 below.

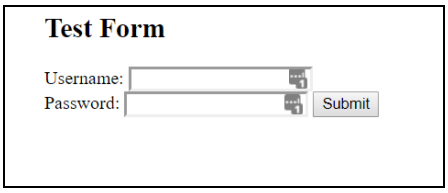

Figure 4: The webpage with a login form

Figure 5(a) and 5(b) shows "test.php", the page that processes the user input from the login page and runs the query to the database, for the MySQL database and the MongoDB database respectively:

```
1.  <?php
2.  //Database information
3.  $dbServer = 'localhost'; //server name
4.  $dbUser = 'health'; //database username
5.  $dbPass = 'secret'; //database password
6.  $db = 'test'; //database name
7.  //Start the MySQL connection
8.  $mysqli = new
    mysqli($dbServer,$dbUser,$dbPass,$db);
9.  if (mysqli_connect_errno()) //if there is a connection
    error
10. {
11. echo "Connect failed: " . mysqli_connect_error();
12. exit();
13. }
14. //store the passed in username and password from login
15. $username = $_POST['username'];
16. $password = $_POST['password'];
17. //The SQL Query
18. $sql= "SELECT fname,lname FROM login WHERE
    username= '$username' AND password= '$password'
    ";
19. //Run the query and store in array 'result'
20. $result= $mysqli->query($sql);
21. ?>
22. <html>
23. <head>
24. <title>Test</title>
25. </head>
26. <body>
27. <h2>Test Page</h2>
28. <?php
29. //if there were any matches for the result
30. if($result-> num_rows > 0)
31. {
32. $row = $result->fetch_assoc(); //convert to array
33. foreach($row as $key => $item)
34. {  //display fname and lname
35. echo $key . ":" . $item . "<br>";
36. }
37. }
38. ?>
39. </body>
40. </html>
```

Figure 5(a): "test.php" using MySQL

```
1.  <?php
2.  require 'vendor/autoload.php';
3.  // connect to the database
4.  $m = new
    MongoDB\Client("mongodb://localhost:27017");
5.  // select the database
6.  $db = $m->test;
7.  //select the collection
8.  $collection = $db->login;
9.  //put the login info into variables
10. $username = $_POST['username'];
11. $password = $_POST['password'];

12. //run the query and return the results to array
    'cursor'
13. $cursor = $collection->find(array("username"=>
    $username,"password"=> $password));
14. ?>
15. <html>
16. <head>
17. <title>Test</title>
18. </head>
19. <body>
20. <h2>Test Page</h2>
21. <?php
22. if($cursor != null) //if there are any results
23. {
24. foreach ($cursor as $document)
25. {
26. ?>
27. <p>fname : <?=$document["fname"] ?> </p>
28. <p>lname : <?=$document["lname"] ?> </p>
29. <?php
30. }
31. }
32. ?>
33. </body>
34. </html>
```

Figure 5(b): "test.php" using MongoDB

In Figure 5(a) the commands corresponding to the line numbers has been discussed below.

- 3-6. The necessary information for the database is stored, i.e. the server name, the database username, the database password, and the database name.
- 8-13. Then the sql connection is started.
- 18-21. The query to get “fname” and “lname” from the login table using the username and password entered is run.
- 30-35. If the query was successful, the attributes and the values for each attribute would be displayed.

Similarly, in Figure 5(b) we discuss the following in lines.

- 2-4. First, the connection to the database server is made,
- 6. The “test” database is selected.
- 13. The query to use the username and password to get “fname” and “lname” is run. 24-28. If the query is successful, those attributes are listed along with the values inside the table that correspond to them.

Figure 6 below displays the results of a successful query for both webpages.

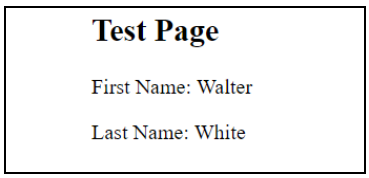

Figure 6: Successful login attempt

The attack types stated in 2.3 for each of the database types were conducted and the results were recorded. Then, all of the countermeasures for each of the database types were applied and the attacks conducted once again, and the results were recorded.

4 Attack Implementation

4.1 SQLI Attacks

4.1.1 Tautology Attack

In the form page for the SQL site:
the statement “ 150 OR 1=1—” was entered into both the username and password fields and then entered.
This would result in the query:
SELECT fname,lname FROM login WHERE username= 150 OR 1=1-- AND password= 150 OR 1=1—; as shown in Figure 7.

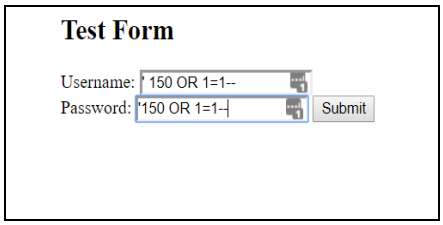

Figure 7: SQLi Tautology Attack

4.1.2 UNION Attack

In the form page for the SQL site:
the statement " *' UNION SELECT cardnumber,pin FROM creditcardsWHERE fname=". "Walter White"–* " was inserted into the username field.
This would result in the query:
SELECT fname,lname FROM login WHERE username= " UNION SELECT cardnumber,pin FROM creditcards WHERE name= "Walter White" -- AND password="
This is shown in Figure 10.

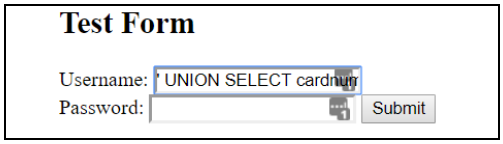

Figure 8: SQLi UNION Attack

4.1.3 Illegal Query Attack

Because the maximum-sized integer that PHP can support is **9,223,372,036,854,775,807**
In the form page for the SQL site: the statement " ' AND 1= **92233720368547759 00**"
was inserted into the username field. This would result in the query:
SELECT * FROM login WHERE username='' AND 1= **92233720368547759 00** AND password= "
As shown in Figure 12, below:

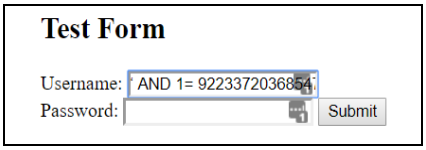

Figure 10: SQLi Illegal Query Attack

4.2 NoSQLI Attacks

4.2.1 PHP Array Injection Attack

In the login form page for the NoSQL site:
The statement "*[$ne]=1*" was entered for both the username and password field
This results in the PHP query:
$collection->find(array("username"=> array("$ne" => 1),"password"=> array("$ne" => 1)));

Which results in the MongoDB query, *db.login.find({username: {$ne:1} , password: {$ne:1}});* as shown in Figure 13 below.

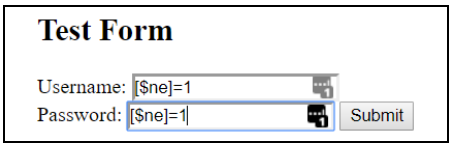

Figure 11: NoSQLi PHP Array Injection Attack

4.2.2 UNION Attack

In the login form page for the NoSQL site. The statement " *', $or: [{}, {'a': 'a " was entered for the username and " ' }]* " for the password
Which results in the MongoDB query:
*db.login.find{ username: 'tolkien', $or: [{}, { 'a': 'a', password '' }]);*as shown in Figure 14 below:

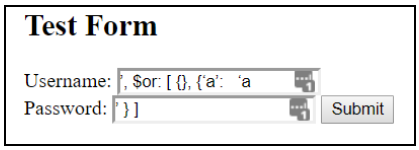

Figure 14: NoSQLi UNION Attack

5 Attack Results

5.1 SQLI Attacks

5.1.1 Tautology Attack

In the tautology attack conducted on the MySQL website, the "fname" and "lname" values from the "login" table were listed despite no username or password being input. This is because when the query was being run, the username and password being returned were true statements so the query:
SELECT fname,lname FROM login WHERE username= '$username' AND password= '$password'
Would return "fname" and "lname" values for every item in the table.

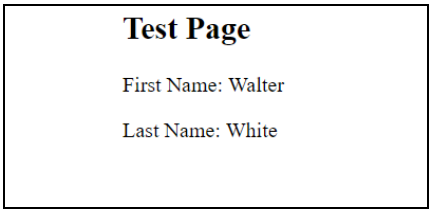

Figure 15: Results of the attacks from 2.3.1(i)

5.1.2 UNION Attack

In the second attack 2.3.1(ii), the UNION attack results in the "fname" and "lname" being replaced by the values for cardnumber and pin as shown in Figure 16 below. The reason why this occurs is because an additional query to get the cardnumber and pin for

the name “Walter White” was attached to the original query resulting in the cardnumber and pin being the results of the query rather than “fname” and “lname” from the “login” table.

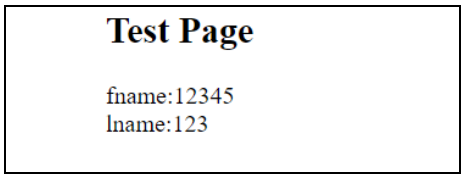

Figure 16: Results of 2.3.1(ii)

5.1.3 Illegal Query Attack

In the third attack 2.3.1(iii), the Illegal Query Attack resulted in an error which caused the page to crash because the PHP interpreter was unable to process the query.

5.2 NoSQLI Attacks

5.2.1 PHP Array Injection Attack

The PHP Array Injection Attack described in 2.3.2(i) is very similar to the tautology attack described in 2.3.1(i) in that they both used always true statements. In this attack, the username and password used in the query are just true statements so the requested “fname” and “lname” do not have to belong to a specific “username” and “password” so all the “fname” and “lname” fields in the “login” table are returned.

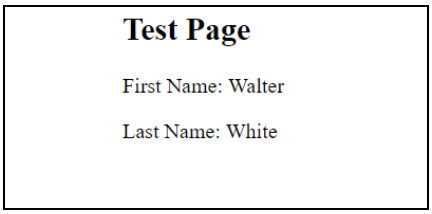

Figure 17: Results of the attacks from 2.3.2(i)

5.1.2 UNION Attack

Because, in this union attack, the query would not require both the password and username to match. The results for “fname” and “lname” where the username was “hello” was returned and listed on the “test.php” page.

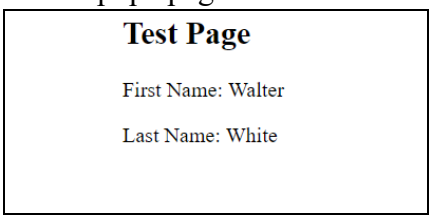

Figure 18: Results of the attacks from 2.3.2(ii)

6 Countermeasures

6.1 Implementation

The countermeasures described in 2.4 were applied to the MongoDB and MySQL websites. Figure 19(a) shows the code for the implementation of countermeasures on the MySQL "test.php" page, and figure 19(b) shows the code for the implementation of countermeasures on the MongoDB "test.php" page.

6.1.1 SQL Page Countermeasures

```
1.  <?php
2.  $dbServer = 'localhost';
3.  $dbUser = 'health';
4.  $dbPass = 'secret';
5.  $db = 'test';

6.  $mysqli = new
        mysqli($dbServer,$dbUser,$dbPass,$db);

7.  if (mysqli_connect_errno())
8.  {
9.  echo "Connect failed: " . mysqli_connect_error();
10. exit();
11. }

12. $username = filter_input
        (INPUT_POST,'username',
        FILTER_SANITIZE_STRING);
13. $password = filter_input (INPUT_POST,'password',
        FILTER_SANITIZE_STRING);
14. mysqli_real_escape_string($mysqli ,$username)
15. mysqli_real_escape_string($mysqli ,$password)

16. $sql= "SELECT fname, lname FROM login
        WHERE username= '$username' AND
        password= '$password' ";


17. $result= $mysqli->query($sql);
18. ?>

19. <html>
20. <head>
21. <title>Test</title>
22. </head>
23. <body>
24. <h2>Test Page</h2>
25. <?php
26. if($result-> num_rows > 0)
27. {
28. $row = $result->fetch_assoc();
29. foreach($row as $key => $item)
30. {
31. echo $key . ":" . $item . "<br>";
32. }
33. }
34. ?>

35. </body>
36. </html>
```

Figure 19(a): Implementation of countermeasures on MySQL "test.php" page

The code for the MySQL “test.php” remains unchanged except the input for the username and password values are made into prepared statements (line 12-13) and then sanitized (line 14-15)

6.1.2 NoSQL Page Countermeasures

```
1.  <?php
2.  require ‘vendor/autoload.php’;
3.  // connect
4.  $m = new MongoDB\Client(“mongodb://localhost:27017”);
5.
6.  // select a database
7.  $db = $m->test;
8.  $collection = $db->login;
9.
10. $username =
        filter_input(INPUT_POST,’username’,FILTER_SANITIZE
        _STRING);
11. $password =
        filter_input(INPUT_POST,’password’,FILTER_SANITIZE
        _STRING);
12.
13. $cursor = $collection->find(array(“username”=>
        $username,”password”=> $password));
14.
15.
16. ?>
17.
18. <html>
19. <head>
20.     <title>Test</title>
21. </head>
22. <body>
23.     <h2>Test Page</h2>
24. <?php
25.     if($cursor != null)
26.     {
27.         foreach ($cursor as $document)
28.         {
29.
30.
31. ?>
32.      <p>fname: <?=$document[“fname”] ?> </p>
33.       <p>lname: <?=$document[“lname”] ?> </p>
34. <?php
35.     }
36.         }
37. ?>
38.
39. </body>
40. </html>
```

Figure 19(b): Implementation of countermeasures on MongoDB “test.php” page

The code for the MongoDB “test.php” remains unchanged except the input for the username and password values are made into prepared statements (line 10-11)

6.2 Countermeasure Results.

6.2.1 SQL Page Countermeasures

When the countermeasures were applied, the next page (“test.php”) was left blank. Hence, the countermeasures were able to prevent all the attempted attacks.

6.2.2 NoSQL Page Countermeasures

When the countermeasures were applied, the next page (“test.php”) was left blank. Hence, the countermeasures were able to prevent all the attempted attacks.

7 Conclusion

While NoSQL Injection attacks tend to occur at a lower rate due to the fact that the queries processed in PHP from traditional strings are converted to objects prior to running. It is more difficult to successfully attempt a NoSQL injection and there are fewer known methods for attacking NoSQL databases. However, they are both equally susceptible to similar attacks if the input that they receive is not sanitized or validated appropriately. The high number of injection attacks that occur are due mainly to the practice of manually formatting and concatenating user input directly to queries.

References

[1]]"Top 10-2017 Top 10 - OWASP", *Owasp.org*, 2017. [Online]. Available: https://www.owasp.org/index.php/Top_10-2017_Top_10.

[2] W. Halfond, A. Orso and P. Manolios, "Using positive tainting and syntax-aware evaluation to counter SQL injection attacks", *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering - SIGSOFT '06/FSE-14*, 2006.

[3] A. Ron, A. Shulman-Peleg and E. Bronshtein, "No SQL, No Injection? Examining NoSQL Security", *Proceedings of the 9th Workshop on Web 2.0 Security and Privacy (W2SP) 2015*, 2015.

[4] A. Ron, A. Shulman-Peleg and A. Puzanov, "Analysis and Mitigation of NoSQL Injections", IEEE Security & Privacy, vol. 14, no. 2, pp. 30-39, 2016.

[5]"Testing for NoSQL injection - OWASP", Owasp.org, 2017. [Online]. Available: https://www.owasp.org/index.php/Testing_for_NoSQL_injection.

[6]"FAQ: MongoDB Fundamentals — MongoDB Manual", Docs.mongodb.com, 2018. [Online]. Available: https://docs.mongodb.com/manual/faq/fundamentals/#how-does-mongodb-address-sql-or-query-injection.

[7]A. Ron and E. Broshtein, "Does NoSQL Equal No Injection?", Security Intelligence, 2015. [Online]. Available: https://securityintelligence.com/does-nosql-equal-no-injection/.

[8]P. Corey, "What is NoSQL Injection?", Pete Corey, 2017. [Online]. Available: http://www.petecorey.com/blog/2017/07/03/what-is-nosql-injection/.

[9]M. Heller, "What is SQL? Structured Query Language explained", InfoWorld, 2017. [Online]. Available: https://www.infoworld.com/article/3219795/sql/what-is-sql-structured-query-language-explained.html.

[10]"The Hitchhiker's Guide to SQL Injection prevention", Treating PHP delusions, 2018. [Online]. Available: https://phpdelusions.net/sql_injection.

[11] Clarke, Justin, and SQL Injection Attacks. "Defense.", 2009.

[12]R. Shettar, A. Ghosh, "SQL Injection Attacks and Defensive Techniques", Int.j.Computer Technology and Applications, vol. 5, no. 2, pp. 699-703, 2014.