University of Calabria
A.A. 2025/2026

CYBER OFFENSE AND DEFENSE

FINAL PROJECT

NoSQL injection

Presented by
Frandina Ilaria 264232 - Siciliano Samuele 263633 - Villella Luigi 269404

Date
December 2025

Database Overview on NoSQL

NOSQL DATABASES

NoSQL databases (MongoDB, Redis, CouchDB, Cassandra) represent a different approach to data management, created to meet the needs of the **modern web** and **distributed applications.**

SQL VS NOSQL

Unlike relational databases, NoSQL uses **flexible schemas** and stores data in formats such as **JSON**. Queries are no longer SQL strings but ***structured objects*** that manipulate documents, key-values, or graphs.

USE AND SPREAD

Their strength lies in their **horizontal scalability** and **high performance**, making them ideal for big data, real-time applications, and cloud-native architectures. Today we find them everywhere: e-commerce, social networks, IoT, analytics systems.

NoSQL injection

HOW NOSQL INJECTION WORKS

Instead of manipulating SQL strings, the attacker exploits **special query language operators**. In MongoDB, some of the most common variations for injecting are as follows:

$ne (Not Equal) - Bypass authentication

Search for admin user with password other than null, if admin exists it will always return a result.

```
{"username": "admin", "password": {"$ne": null}}
```

$where - JavaScript code execution

Executes JavaScript arbiter inside the database. With || 1==1 “where” it loses meaning.

```
{"$where": "this.password == 'x' || '1'=='1'"}
```

$regex - Regex Execution

Look for passwords that start with “a”. It is useful for rebuilding the password, char by char.

```
{"password": {"$regex": "^a"}}
```

When user input **is NOT properly validated**, an attacker **can authenticate without valid credentials**, **extract sensitive data** from the database, or, in the worst cases, **execute arbitrary code on the server**. However, input validation is not **just** a SQL (or NoSQL) issue; it's a general security issue. You **must never trust** user input, whatever database you use.

1° - Exploiting NoSQL operator injection to bypass authentication

By script solution DEMO

INITIAL TESTS We tested if the application was vulnerable to NoSQL injection by trying to bypass authentication through these two steps:

01 Attempting to log in without knowing the username using the $gt operator
{"username":{"$gt": ""},"password":"peter"}

02 Attempting to log in without knowing your password using the $gt operator
{"username":"wiener","password":{"$gt": ""}}

SOLUTION After checking the application vulnerability, we built the final payload to conclude the exploit. Using the $gt (greater than) operator in the password field to bypass the authentication check (since any password in the database will be greater than an empty string, making the condition always true) and the $regex operator to search for an account whose name begins with "admin"

03 Attempting to enter by searching for any username starting with “admin” using the $regex operator and bypassing password checking using the $gt operation
{"username":{ "$regex": "admin.*"},"password":{"$gt": ""}}

2° - Exploiting NoSQL injection to extract data

By script Solution DEMO

INITIAL TESTS We tested whether the application was vulnerable by checking whether it was running injected JavaScript code at **$where** by adding a always **true OR condition** to the GET of lookup:
administrator' || '1'=='1

SOLUTION After verifying the lookup API vulnerability, we performed the following steps to rebuild the administrator account password.

01 Search for length range by testing in blocks of 10 (1-10, 10-20 etc.) until the correct range is found
administrator' & this.password.length < 10 || ‘a’==’b

02 Determining the exact length by testing each value in the range found
administrator' & this.password.length == 8 || ‘a’==’b

03 Extract password char by char using array access operator and iterating over the entire alphabet for each position
administrator' & this.password[0] == 'a' || 'a'=='b

The final condition **|| 'a'=='b** is added for each request,used to balance the superscripts of the original query
ORIGINAL: db.users.find({ $where: "this.username == 'USER_INPUT'" })
WITH INJECTION: db.users.find({ $where: "this.username == 'administrator' & CONDITION || 'a'=='b'" })

3° - Exploiting NoSQL operator injection to extract unknown fields

By script solution DEMO

INITIAL TESTS We verified the vulnerability by injecting JavaScript ($where) into the login. Analyzing traffic with ZAP, we noticed distinct behaviors:

1. On active users (e.g. wiener), password bypass ensures immediate access.
2. On the victim user Carlos, since the fact that his account has been disabled, the bypass returns the error "Account locked". This message becomes our Boolean Oracle.

SOLUTION The script automates the attack on Carlos. To force the $where clause to execute in the database, we first bypass the password check using the MongoDB operator $ne (not equal):

{ "username": "carlos", "password": {"$ne": "invalid"}, "$where": "..." };

With **"$where": "1"** we have gained the response “Account locked”

01

The script iterates over Object.keys(this). For each index, it reconstructs the char-by-char field name of the user document, checking each attempt via regex against the oracle and discovering the existence of hidden parameters such as “resetToken”.

"$where": “Object.keys(this)[i].match('^partial.*')”

02

Once the target field is identified, the script extracts its value (the token). Each character is guessed by testing whether the ^partialtoken.* regex generates the "Account locked" error.

"$where": this.nameFieldFound.match('^partialToken.*')

Prevention

GENERAL GUIDELINES

Countermeasures vary depending on the specific NoSQL technology used, but some general guidelines can be followed to mitigate the risk of injection.

01 VALIDATION AND SANITIZATION OF INPUT

Implement a whitelist of allowed characters and reject any input that contains unexpected characters or patterns, always validating the format and data type.

02 PARAMETERIZED QUERIES

Avoid **direct concatenation** of user input into queries, instead using the parameter binding capabilities offered by the database to separate query logic from user-supplied data.

03 OPERATORS CHECK

Whitelist allowed **operators** and block dangerous operators like $where or $regex when not strictly necessary, validating the structure of JSON objects before processing them.

University of Calabria
A.A. 2025/2026

THANK YOU FOR LISTENING

Presented by
Frandina Ilaria 264232 - Siciliano Samuele 263633 - Villella Luigi 269404

Date
December 2025