What is NoSQL Injection and How Can You Prevent It?

Zbigniew Banach - Fri, 29 May 2020 -

NoSQL injection vulnerabilities allow attackers to inject code into commands for databases that don't use SQL queries, such as MongoDB. Let's see how NoSQL injection differs from traditional SQL injection and what you can do to prevent it.

A Quick Introduction to NoSQL

Even if you don't work with databases, you've probably heard of NoSQL among the cloud-related buzzwords of the past few years. NoSQL (a.k.a. "non-SQL" or "not only SQL") is a general term covering databases that don't use the SQL query language. In practice, it's used to refer to non-relational databases that are growing in popularity as the back-end for distributed cloud platforms and web applications. Instead of storing data in tables, as with relational databases, NoSQL data stores use other data models that are better suited for specific purposes, such as documents, graphs, objects, and many others.

MongoDB, currently one of the most popular NoSQL database products, stores data as documents using a syntax similar to JSON (JavaScript Object Notation). Among other benefits, this allows developers to build full-stack applications using only JavaScript. We will use MongoDB queries to demonstrate how attackers can exploit unvalidated user input to inject code into a web application backed by a NoSQL database.

Why MongoDB Injection Is Possible

With traditional SQL injection, the attacker exploits unsafe user input processing to modify or replace SQL queries (or other SQL statements) that the application sends to a database engine. In other words, an SQL injection allows the attacker to execute commands in the database. Unlike relational databases, NoSQL databases don't use a common query language. NoSQL query syntax is product-specific and queries are written in the programming language of the application: PHP, JavaScript, Python, Java, and so on. This means that a successful injection lets the attacker execute commands not only in the database, but also in the application itself, which can be far more dangerous.

MongoDB uses the Binary JSON (BSON) data format and comes with a secure BSON query assembly tool. Queries are also represented as BSON objects (i.e. binary data), so direct string injection is not possible. However, MongoDB deliberately allows applications to run JavaScript on the server within the `$where` and `mapReduce` operations. The documentation is very clear that if this functionality is used, the developer should take care to prevent users from submitting malicious JavaScript. In other words, MongoDB deliberately includes a potential injection vector. So what can a malicious user do if the developer is *not* careful?

Simple MongoDB Injection in PHP

For a basic authentication bypass, the attacker can try to enter MongoDB operators in field values, for example `$eq` (equals), `$ne` (not equal to) or `$gt` (greater than). Here's an unsafe way to build a database query in a PHP application, with the parameter values taken directly from a form:

```
$query = array("user" => $_POST["username"], "password" =>
    $_POST["password"]);
```

If this query is then used to check login credentials, the attacker can abuse PHP's built-in associative array processing to inject a MongoDB query that always returns true and bypass the authentication process. This may be as simple as sending the following POST request:

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

PHP will translate this into an array of arrays:

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

When sent as a MongoDB query to a user store, this will find all users where the user name and password are not equal to 1, which is highly likely to be true and may allow the attacker to bypass authentication.

JavaScript Injection in MongoDB

Let's say we have a vulnerable application where the developer uses MongoDB's `$where` query operator with unvalidated user inputs. This allows an attacker to inject malicious input containing JavaScript code. While the attacker can't inject completely arbitrary JavaScript, only code that uses a limited set of functions, this is quite enough for a useful attack.

To query a MongoDB data store with the `$where` operator, you would normally use the `find()` function, for example:

```
db.collection.find( { $where: function() {
    return (this.name == 'Invicti') } } );
```

This would match records with name Invicti. A vulnerable PHP application might directly insert unsanitized user input when building the query, for example from the variable `$userData`:

```
db.collection.find( { $where: function() {
    return (this.name == $userData) } } );
```

The attacker might then inject an exploit string like `'a'; sleep(5000)` into `$userData` to have the server pause for 5 seconds if the injection was successful. The query executed by the server would be:

```
db.collection.find( { $where: function() {
    return (this.name == 'a'; sleep(5000) ) } } );
```

Because queries are written in the application language, this is just one of the many types of injection possible. For example, if Node.js is used for server-side scripting, as in the popular MEAN stack (MongoDB, ExpressJS, AngularJS, and Node.js), server-side JavaScript injection into Node.js may be possible.

Preventing NoSQL Injection Attacks

The consequences of a successful MongoDB injection or other NoSQL injection attack can be even more serious than with traditional SQL injection. Attackers can not only extract data from the database, but also execute code in the context of the application, for example to perform denial of service attacks or even compromise admin user accounts and take control of the server. Such attacks are especially dangerous since NoSQL data stores are often a novelty to developers familiar only with relational database products, which increases the risk of insecure code.

Many popular NoSQL products are still young and under intense development, so it's always a good idea to use the most recent version. For example, earlier versions of MongoDB were notoriously insecure on many levels and had serious injection vulnerabilities, while in more recent versions, security is taken much more seriously.

As is often the case in web application security, the best way to prevent NoSQL injection attacks is to avoid using unsanitized user inputs in application code, especially when building database queries. MongoDB, for example, has built-in features for secure query building without JavaScript. If you do need to use JavaScript in queries, follow the usual best practices: validate and encode all user inputs, apply the rule of least privilege, and know your language to avoid using vulnerable constructs. For more advice, see the OWASP page on testing for NoSQL injection.

Retrieved from https://www.invicti.com/blog/web-security/what-is-nosql-injection/ on 3 Oct 2022