NoSQL Injection

25.4k views

App Security | Attack Tools | Essentials | Threats

What Is NoSQL Injection?

NoSQL injection is a security weakness in a web application that uses a NoSQL database. NoSQL (Not Only SQL) refers to database systems that use more flexible data formats and do not support Structured Query Language (SQL). They typically store and manage data as key-value pairs, documents, or data graphs.

A NoSQL injection, similar to that of a SQL injection, can allow attackers to bypass authentication, exfiltrate sensitive data, tamper with data on the database, or even compromise the database and the underlying server. Most NoSQL injection vulnerabilities occur because developers accept and process user inputs without properly sanitizing them.

NoSQL databases do not support one standardized query language and therefore the exact queries allowed depend on:

- **Database engine**—for example, MongoDB, Cassandra, Redis, or Google Bigtable
- **Programming language**—for example, Python, PHP
- **Development framework**—for example, Angular, Node.js

A common denominator of most NoSQL databases is that they support the text-based JavaScript Object Notation (JSON) format, and typically allow user input via JSON files. If this input is not sanitized, it can be vulnerable to injection attacks.

NoSQL vs. SQL Injection

NoSQL databases have more relaxed consistency restrictions than standard SQL databases. The lower number of consistency checks and relational constraints provide advantages in terms of scaling and performance. However, NoSQL databases remain vulnerable to injection even if they don't use the SQL syntax.

Hackers can execute NoSQL injection attacks using procedural languages instead of SQL, a declarative language. These attacks can cause more damage than conventional SQL injection attacks.

The main differences between NoSQL and SQL injection attacks are the syntax and grammar of the queries. Attackers are unlikely to succeed if they attempt to execute a NoSQL injection attack using a malicious SQL injection string because NoSQL databases don't use standardized languages. However, NoSQL databases do use languages with a similar syntax to SQL, given that they have the same function.

Attackers can perform NoSQL injection in different application areas than an SQL injection attack. While SQL injection executes in the database engine, a NoSQL attack may execute at the database or application layer depending on the data model and NoSQL API. NoSQL injection attacks usually execute in the part of the application that parses, evaluates, or concatenates the attack string into an API call.

How Does NoSQL Injection Work?

NoSQL injection occurs when a query, most commonly delivered by an end-user, is not sanitized, allowing the attacker to include malicious input that executes an unwanted command on the database.

Traditional SQL injection techniques do not work on NoSQL databases, because they use a specific query language which does not support SQL. To attack NoSQL databases, attackers must adjust their techniques to product-specific query syntax, which might be written in the same language used to code the database engine.

Because some NoSQL databases use application code for their queries, attackers can not only perform unwanted actions on a NoSQL database, but also execute malicious code and unvalidated input within the application itself. This allows attackers to hijack servers and exploit vulnerabilities that go beyond the usual scope of SQL injection attacks—making NoSQL injections, in some cases, more severe than SQL injection.

In a typical NoSQL architecture, data access is managed by a software driver. Libraries in multiple languages are available to the database clients, allowing them to access the database. Even if these drivers are not vulnerable, they may have insecure APIs. This is another threat vector that can allow arbitrary code execution in the database and in the application itself, if not implemented safely by developers.

Example of NoSQL Injection in MongoDB

MongoDB is a common NoSQL database. Here are a couple of examples of how attackers can exploit the $where operator in MongoDB.

Example #1: Manipulating Input Data

If the attacker can manipulate the data that the $where operator receives, the attacker can inject malicious JavaScript that MongoDB will evaluate as part of the query. This vulnerability arises when user input passes

directly to the MongoDB query and avoids deletion. For example, the attacker could use the following script to exploit this vulnerability:

```
db.myCollection.find( { active: true, $where: function() { return obj.credits—obj.debits <
$userInput; } } );;
```

Injection testers can expose vulnerabilities without fully exploiting them. For example, they can insert special characters in the target API language to observe the result. This injection allows testers to determine if an application discards the input successfully. In MongoDB, the database will return an error if the string passed unfiltered contains a special character (i.e., ” ‘ ; \ { }).

Similar vulnerabilities allow attackers to execute arbitrary SQL code with regular SQL injection attacks. An attacker can freely manipulate or publish or manipulate your data and leverage the full-featured nature of JavaScript to force arbitrary commands. In addition to returning a database error during injection tests, the exploit relies on special characters to insert valid JavaScript.

For example, an attacker could execute a JavaScript function by inserting the following input into the $userInput element in the code:

```
0;var date=new Date(); do{curDate = new Date();}while(curDate-date<10000)
```

This input will result in the execution of the following attack string, inducing the target MongoDB instance to execute at full CPU usage for 10 seconds:

```
function() { return obj.credits—obj.debits < 0;var date=new Date(); do{curDate = new
Date();}while(curDate-date<10000); }
```

Example #2: Replacing the $where Operator

If MongoDB parameterizes or discards the query input altogether, attackers can use alternative techniques to perform a NoSQL injection attack. NoSQL instances often have reserved variable names independent of the application's programming language.

For example, the $where component in MongoDB is a reserved query operator that must be passed to queries unchanged—changing the $where construct can induce database errors. However, MongoDB also accepts $where as a valid name for the PHP variable, allowing attackers to create PHP variables called $where, which

they use to inject malicious code into the database query. The PHP documentation in MongoDB provides explicit warnings about this vulnerability.

It is important to enclose any special query operator starting with $ using single quotation marks ‘ ’ to prevent PHP from attempting to replace the $exists element with the variable $exists value.

Attackers can exploit MongoDB by inserting malicious code in place of the operator. This technique works even for queries that don’t require user input. For example:

```
db.myCollection.find( { $where: function() { return obj.credits–obj.debits < 0; } } );
```

An attacker can assign data to a PHP variable by contaminating HTTP parameters. The contaminated parameters can trigger MongoDB errors by creating $where variables—the parameter contamination indicates that the query is invalid. The $where value is enough to expose a vulnerability even without the actual $where string. The attacker can fully exploit this vulnerability by inserting the following script:

```
$where: function() { //inserted arbitrary JavaScript }
```

Preventing NoSQL Injection Attacks

Improve Developer Skills

NoSQL attacks can be more difficult to prevent than traditional SQL injection because many NoSQL databases include unsafe or non-standard code and functionality, which is unfamiliar to developers. The first step is to read the documentation and security guidelines for your specific NoSQL database.

Beyond the basics, teams should invest in training to ensure developers are well-versed in the database engine being used, and know how to correctly implement security best practices.

Use the Latest Version

Many popular NoSQL products are in active development, so it is important to use the latest version and upgrade frequently. Vulnerabilities are discovered in NoSQL databases on a daily basis. For example, older versions of MongoDB were less secure and suffered from serious injection vulnerabilities, but newer versions are more secure.

Avoid Accepting Raw User Input in Application Code

The best way to prevent NoSQL injection attacks is to avoid using raw user input in your application code, especially when writing database queries. For example, MongoDB has built-in functionality to build secure queries without using JavaScript.

If you do need to use JavaScript for your queries, carefully validate and encode all user inputs, enforce least-privilege rules, and ensure you use secure coding practices in the relevant programming language to avoid using vulnerable constructs.

NoSQL Injection Protection with Imperva

Imperva Web Application Firewall can prevent application layer attacks, including NoSQL, SQL, and other code injections, with world-class analysis of web traffic to your applications.

Beyond injection attack prevention, Imperva provides comprehensive protection for applications, APIs, and microservices:

Runtime Application Self-Protection (RASP) – Real-time attack detection and prevention from your application runtime environment goes wherever your applications go. Stop external attacks and injections and reduce your vulnerability backlog.

API Security – Continuous protection for all APIs using deep discovery and classification of sensitive data to detect all public, private and shadow APIs to empower security teams to implement a positive security model.

Advanced Bot Protection – Prevent business logic attacks from all access points – websites, mobile apps and APIs. Gain seamless visibility and control over bot traffic to stop online fraud through account takeover or competitive price scraping.

DDoS Protection – Block attack traffic at the edge to ensure business continuity with guaranteed uptime and no performance impact. Secure your on premises or cloud-based assets – whether you're hosted in AWS, Microsoft Azure, or Google Public Cloud.

Attack Analytics – Ensures complete visibility with machine learning and domain expertise across the application security stack to reveal patterns in the noise and detect application attacks, enabling you to isolate and prevent attack campaigns.

Client-Side Protection – Gain visibility and control over third-party JavaScript code to reduce the risk of supply chain fraud, prevent data breaches, and client-side attacks.