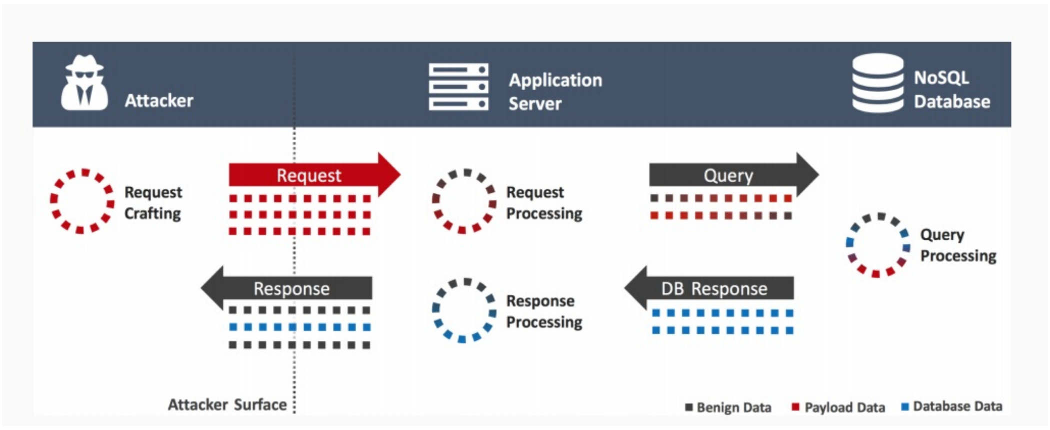
How to pull off a successful NoSQL Injection attack

From NoSQL Injection to Serverside Javascript Injection

Fiddly Cookie Follow
Feb 4, 2019 · 6 min read ★



Source

NoSQL (not only SQL) has disrupted the usage of traditional data stores. It has introduced a new concept of data storage which is non-relational unlike the previous storage mechanisms and thus, provides looser restrictions in consistency. It has a document store, key-value store, and graph. Due to the new requirements of modern-day applications, there has been wide adoption of NoSQL databases which could conveniently facilitate the distribution of data across numerous servers. Nosql databases provide an avenue for wide scalability and they require a single database node to execute all operations of the same transaction

NoSQL models offer a new data model and query formats making the old SQL injection attacks irrelevant. Yet, they give attackers new ways to insert malicious code.

Let's understand the NoSQL data models in MongoDB:

Source

Source

The following diagram illustrates a query that selects and orders the matching documents using an index:

Read operations retrieves **documents** from a **collection** i.e. queries a collection for documents. This is how you can read the documents from a collection in MongoDB:

> *db.items.find(queryObject)*
>
> *db — current database object*
>
> *Items — collection names ‘items’ in the current database*
>
> *find — method to execute on the collection*
>
> *queryObject — an object used to select data*

```
queryObject = {amount:0}; //items with fixed value ‘amount’ is 0
```

Before we go any further, let’s quickly analyze the attack mechanism in traditional SQL databases.

SQL Injection

Consider a simple SQL statement used to authenticate a user with a username and password.

```
SELECT * FROM accounts WHERE username = ‘$username’ AND password =
‘$password’
```

If the developer has not incorporated prepared statement to construct the SQL queries, an attacker can supply admin’ — in the username field to access the admin user’s account bypassing the condition that checks for the password. The tampered query may look like this:

```
SELECT * FROM accounts WHERE username = ‘admin’ — AND password = ‘’
```

What is a Second-Order SQL Injection and how can you exploit it successfully?

What is SQL Injection

medium.com

NoSQL Injection

NoSQL database is accessed using a driver which exposes a wrapper that provides libraries in multiple languages for the DB client. These drivers themselves might not be vulnerable, sometimes they present unsafe APIs that, when implemented insecurely by an application developer, could open up avenues for vulnerabilities in the application that allow arbitrary operations on the database. Yet these databases are still potentially vulnerable to injection attacks, even if they aren’t using the traditional SQL syntax.

MongoDB expects input in JSON array format. The equivalent of the previously illustrated query for NoSQL MongoDB database is

```
db.accounts.find({username: username, password: password});
```

While here we are no longer dealing with a query language, an attacker can still achieve the same results as SQL injection by supplying a JSON input object as below:

```
POST /login HTTP/1.1
Host: example.org
Content-Type: application/json
Content-Length: 38

{
“username”: “admin”,
“Password”: {'$gt': “”}
}
```

If it is not a JSON request

```
POST /login HTTP/1.1
Host: example.org
Content-Type: application/x-www-form-urlencoded
Content-Length: 29

user=admin&password[%24ne]=
```

MongoDB Operators

Here was an example of an equivalent attack in both cases, where the attacker manages to retrieve the admin user's record without knowing a password. Both SQL and NoSQL databases are vulnerable to attack.

Exploiting MongoDB query operators:

MongoDB has a lot of query operators

> *$ne — not equal*
>
> *$gt — greater than*
>
> *$regex — regular expression*
>
> *$where — clause lets you specify a script to filter results*

$gt & $ne query operator

db.items.find(queryObject)

queryObject — an object used to select data, for instance:

```
queryObject = {amount:0}; //items with fixed value ‘amount’ is 0
```

If we inject $gt:0, we will obtain all the items with field amount >0. The queryObject will be as follows:

```
queryObject = {amount: {$gt:0}}; //items with field amount >0
```

Let's consider that the following JSON request is sent to the application:

```
{desiredType: ‘shirt’}
{desiredType: {$ge:0}}
```

Application Controller

```
app.post(‘/products/find’,(req,res) => {
const query = {type: req.body.desiredType}
Document.find(query).exec().then((r)=>res.json(r));});
```

The final query becomes as follows:

```
{type: {$gt:0}}
```

In MongoDB, ***$gt*** selects those documents where the value of the field is greater than (i.e. >) the specified value. Thus the above statement compares the type in the database with type greater than 0, which returns ***true***.

The same results can be achieved using another comparison operator such as ***$ne***.

$where query operator

The ***$where*** clause lets you specify a script to filter results. **This operator in MongoDB is a feature that is best avoided.** Now, why should we avoid some feature which allows us to specify scripts? Let’s understand this with a simple example. Let’s say we have a blog. Our blog has lots of articles that can be read publicly, but we also have some private articles which are for our internal use and not supposed to be published. So we have a field hidden in our documents which can be ***true*** or ***false*** depending on whether or not our visitors are supposed to see the article. Our MongoDB query to get a list of all articles in a certain category for displaying it to the website visitor looks like this:

```
db.articles.find({“$where”: “this.hidden == false && this.category
== ‘“+category+”’” });
```

That should make sure nobody looks at our hidden articles. Or does it? When the user controls the category-variable, they can set it to this string to *‘; return ‘’ == ‘*

The resulting Javascript snippet which gets sent to the database is this:

```
this.hidden == false && this.category == ‘’; return ‘’ == ‘’
```

But why can’t we simply pass a ‘; ‘’=’

When you have a javascript snippet with multiple commands separated by a semicolon ‘;’, they will be executed as a function and a return statement is needed to determine which value will be passed back to the caller. This function will always return true. That means the user will also see all articles in our collection, including those which are supposed to be hidden.

Apparently, MongoDB and many of the NoSQL databases are nothing more than JavaScript processing engines

Source

Its performance is abysmal, and not just because it doesn't benefit from indexes. Almost every common use-case can be solved much more efficiently with a common find-query or aggregation, especially one as trivial as this.

How do I prevent it?

Here are some measures to prevent SQL / NoSQL injection attacks, or minimize the impact if it happens:

1. Prepared Statements: For SQL calls, use prepared statements instead of building dynamic queries using string concatenation.
2. Input Validation: Validate inputs to detect malicious values. For NoSQL databases, also validate input types against expected types
3. Least Privilege: To minimize the potential damage of a successful injection attack, do not assign DBA or admin type access rights to your application accounts. Similarly, minimize the privileges of the operating system account that the database process runs under.
4. HAPI and EXPRESS

. . .

Thanks for reading. Is there is anything I missed? Feel free to let me know.

Fiddly Cookie

The latest Tweets from Fiddly Cookie (@fiddlycookie). I build to break. Pentester | Researcher | DevSecOps Engineer...

twitter.com

Handling threats from plagued 'White Collar Robots' — A case-study on RPA security

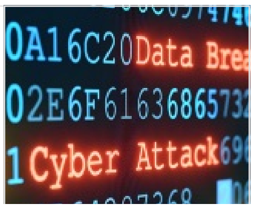
Robotic Process Automation (RPA) is a new wave of future technologies. Evolving over time, automation started with...

medium.com

Things most cyber-security professionals are not aware of

Real-world data on security incidents and data breaches from Verizon Data Breach Investigations Report

medium.com



medium.com

NoSQL Vulerability Web Security Bug Bounty