Introduction to NoSQL Injection

Introduction to NoSQL

Background

Many applications rely on databases to store data, such as passwords, email addresses, or comments. The most popular database engines are `relational` (e.g. Oracle and MySQL). However, over the past decade, `non-relational` databases, also known as `NoSQL` databases, have become increasingly more common, with `MongoDB` now being the 5th most used database engine (as of November 2022).

There are four main types of `NoSQL` databases, and unlike `relational` databases, which all store data similarly in `tables`, `rows`, and `columns`, the way `NoSQL` databases store data varies significantly across the different categories and implementations.

Type	Description	Top 3 Engines (as of November 2022)
Document-Oriented Database	Stores data in `documents` which contain pairs of `fields` and `values`. These documents are typically encoded in formats such as `JSON` or `XML`.	MongoDB, Amazon DynamoDB, Google Firebase - Cloud Firestore
Key-Value Database	A data structure that stores data in `key:value` pairs, also known as a `dictionary`.	Redis, Amazon DynamoDB, Azure Cosmos DB
Wide-Column Store	Used for storing enormous amounts of data in `tables`, `rows`, and `columns` like a relational database, but with the ability to handle more ambiguous data types.	Apache Cassandra, Apache HBase, Azure Cosmos DB
Graph Database	Stores data in `nodes` and uses `edges` to define relationships.	Neo4j, Azure Cosmos DB, Virtuoso

In this module, we will focus solely on `MongoDB`, as it is the most popular `NoSQL` database.

Introduction to MongoDB

`MongoDB` is a `document-oriented` database, which means data is stored in `collections` of `documents` composed of `fields` and `values`. In `MongoDB`, these `documents` are encoded in BSON (Binary JSON). An example of a `document` that may be stored in a `MongoDB` database is:

Code: javascript

```javascript
{
  _id: ObjectId("63651456d18bf6c01b8eeae9"),
  type: 'Granny Smith',
  price: 0.65
}
```

Here we can see the document's `fields` (type, price) and their respective `values` ('Granny Smith', '0.65'). The field `_id` is reserved by MongoDB to act as a document's `primary key`, and it must be unique throughout the entire `collection`.

Connecting to MongoDB

We can use `mongosh` to interact with a MongoDB database from the command line by passing the connection string. Note that `27017/tcp` is the default port for MongoDB.

mayala@htb[/htb] `$ mongosh mongodb://127.0.0.1:27017 Current Mongosh Log ID: 636510136bfa115e590dae03 Connecting to: mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+1.6.0 Using MongoDB: 6.0.2 Using Mongosh: 1.6.0 For mongosh info see: https://docs.mongodb.com/mongodb-shell/ test>`

We can check which databases exist like this:

Code: javascript

```javascript
test> show databases
admin        72.00 KiB
config      108.00 KiB
local        40.00 KiB
```

Creating a Database

MongoDB does not create a `database` until you first store data in that `database`. We can "switch" to a new `database` called `academy` by using the `use` command:

Code: javascript

```
test> use academy
switched to db academy
academy>
```

We can list all collections in a database with `show collections`.

Inserting Data

Similarly to creating a database, MongoDB only creates a `collection` when you first insert a `document` into that `collection`. We can insert data into a `collection` in several ways.

We can insert a `single` document into the `apples` collection like this:

Code: javascript

```javascript
academy> db.apples.insertOne({type: "Granny Smith", price: 0.65})
{
  acknowledged: true,
  insertedId: ObjectId("63651456d18bf6c01b8eeae9")
}
```

And we can insert `multiple` documents into the `apples` collection like this:

Code: javascript

```javascript
academy> db.apples.insertMany([{type: "Golden Delicious", price: 0.79},
{type: "Pink Lady", price: 0.90}])
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId("6365147cd18bf6c01b8eeaea"),
    '1': ObjectId("6365147cd18bf6c01b8eeaeb")
  }
}
```

Selecting Data

Let's say we wanted to check the price of `Granny Smith` apples. One way to do this is by specifying a document with fields and values we want to match:

Code: javascript

```
academy> db.apples.find({type: "Granny Smith"})
{
  _id: ObjectId("63651456d18bf6c01b8eeae9"),
  type: 'Granny Smith',
  price: 0.65
}
```

Or perhaps we wanted to list all documents in the collection. We can do this by passing an empty document (since it is a subset of all documents):

Code: javascript

```javascript
academy> db.apples.find({})
[
  {
    _id: ObjectId("63651456d18bf6c01b8eeae9"),
    type: 'Granny Smith',
    price: 0.65
  },
  {
    _id: ObjectId("6365147cd18bf6c01b8eeaea"),
    type: 'Golden Delicious',
    price: 0.79
  },
  {
    _id: ObjectId("6365147cd18bf6c01b8eeaeb"),
    type: 'Pink Lady',
    price: 0.90
  }
]
```

If we wanted to do more advanced queries, such as finding `all apples whose type starts with a 'G' and whose price is less than 0.70`, we would have to use a combination of query operators. There are many `query operators` in MongoDB, but some of the most common are:

Type	Operator	Description	Example
Comparison	`$eq`	Matches values which are `equal to` a specified value	`type: {$eq: "Pink Lady"}`
Comparison	`$gt`	Matches values which are `greater than` a specified value	`price: {$gt: 0.30}`

Type	Operator	Description	Example
Comparison	`$gte`	Matches values which are `greater than or equal to` a specified value	`price: {$gte: 0.50}`
Comparison	`$in`	Matches values which exist `in the specified array`	`type: {$in: ["Granny Smith", "Pink Lady"]}`
Comparison	`$lt`	Matches values which are `less than` a specified value	`price: {$lt: 0.60}`
Comparison	`$lte`	Matches values which are `less than or equal to` a specified value	`price: {$lte: 0.75}`
Comparison	`$nin`	Matches values which are `not in the specified array`	`type: {$nin: ["Golden Delicious", "Granny Smith"]}`
Logical	`$and`	Matches documents which `meet the conditions of both` specified queries	`$and: [{type: 'Granny Smith'}, {price: 0.65}]`
Logical	`$not`	Matches documents which `do not meet the conditions` of a specified query	`type: {$not: {$eq: "Granny Smith"}}`
Logical	`$nor`	Matches documents which `do not meet the conditions` of any of the specified queries	`$nor: [{type: 'Granny Smith'}, {price: 0.79}]`
Logical	`$or`	Matches documents which `meet the conditions of one` of the specified queries	`$or: [{type: 'Granny Smith'}, {price: 0.79}]`
Evaluation	`$mod`	Matches values which divided by a `specific divisor` have the `specified remainder`	`price: {$mod: [4, 0]}`
Evaluation	`$regex`	Matches values which `match a specified RegEx`	`type: {$regex: /^G.*/}`
Evaluation	`$where`	Matches documents which [satisfy a JavaScript expression]()	`$where: 'this.type.length === 9'`

Going back to the example from before, if we wanted to select `all apples whose type starts with a 'G' and whose price is less than 0.70`, we could do this:

Code: javascript

```
academy> db.apples.find({
    $and: [
        {
            type: {
                $regex: /^G/
            }
        },
        {
            price: {
                $lt: 0.70
            }
        }
    ]
});
[
  {
    _id: ObjectId("63651456d18bf6c01b8eeae9"),
    type: 'Granny Smith',
    price: 0.65
  }
]
```

Alternatively, we could use the `$where` operator to get the same result:

Code: javascript

```
academy> db.apples.find({$where: `this.type.startsWith('G') && this.price <
0.70`});
[
  {
    _id: ObjectId("63651456d18bf6c01b8eeae9"),
    type: 'Granny Smith',
    price: 0.65
  }
]
```

If we want to sort data from `find` queries, we can do so by appending the sort function. For example, if we want to select the `top two apples sorted by price in descending order` we can do so like this:

Code: javascript

```
academy> db.apples.find({}).sort({price: -1}).limit(2)
[
  {
    _id: ObjectId("6365147cd18bf6c01b8eeaeb"),
    type: 'Pink Lady',
    price: 0.9
  },
  {
    _id: ObjectId("6365147cd18bf6c01b8eeaea"),
    type: 'Golden Delicious',
    price: 0.79
  }
]
```

If we wanted to reverse the sort order, we would use `1 (Ascending)` instead of `-1 (Descending)`. Note the `.limit(2)` at the end, which allows us to set a limit on the number of results to be returned.

Updating Documents

Update operations take a `filter` and an `update` operation. The `filter` selects the documents we will update, and the `update` operation is carried out on those documents. Similar to the `query operators`, there are update operators in MongoDB. The most commonly used update operator is `$set`, which updates the specified field's value.

Imagine that the price for `Granny Smith` apples has risen from `0.65` to `1.99` due to inflation. To update the document, we would do this:

Code: javascript

```
academy> db.apples.updateOne({type: "Granny Smith"}, {$set: {price: 1.99}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

If we want to increase the prices of all apples at the same time, we could use the `$inc` operator and do this:

Code: javascript

```
academy> db.apples.updateMany({}, {$inc: {quantity: 1, "price": 1}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 3,
  modifiedCount: 3,
  upsertedCount: 0
}
```

The `$set` operator allows us to update specific fields in an existing document, but if we want to completely replace the document, we can do that with `replaceOne` like this:

Code: javascript

```
academy> db.apples.replaceOne({type:'Pink Lady'}, {name: 'Pink Lady', price: 
0.99, color: 'Pink'})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

Removing Documents

Removing a document is very similar to selecting documents. We pass a query, and the matching documents are removed. Let's say we wanted to `remove apples whose prices are less than 0.80`:

Code: javascript

```
academy> db.apples.remove({price: {$lt: 0.8}})
{ acknowledged: true, deletedCount: 2 }
```

Conclusion

By now, you should have a basic understanding of NoSQL databases and how to use MongoDB. The following section will cover some fundamentals of NoSQL injection attacks.

Note: To connect to the exercise, use the command `mongosh mongodb://SERVER_IP:PORT` with the IP and PORT provided with the question.

Introduction to NoSQL Injection

What is NoSQL Injection?

When `user input` is incorporated into a NoSQL query `without being properly sanitized` first, `NoSQL injection` may occur. If an attacker can control part of the query, they may subvert the logic and get the server to carry out `unintended actions / return unintended results`. Since NoSQL has no standardized query language like SQL does, NoSQL injection attacks look different in the various NoSQL implementation.

Scenario (Node.JS)

Let's imagine an `Express / Node.JS` webserver that uses `MongoDB` to store its user's information. This server has the endpoint `/api/v1/getUser`, which allows you to retrieve a user's information from their username.

Code: javascript

```javascript
// Express is a Web-Framework for Node.JS
const express = require('express');
const app = express();
app.use(express.json()); // Tell Express to accept JSON request bodies

// MongoDB driver for Node.JS and the connection string
// for our local MongoDB database
const {MongoClient} = require('mongodb');
const uri = "mongodb://127.0.0.1:27017/test";
const client = new MongoClient(uri);

// POST /api/v1/getUser
// Input (JSON): {"username": <username>}
// Returns: User details where username=<username>
app.post('/api/v1/getUser', (req, res) => {
    client.connect(function(_, con) {
        const cursor = con
            .db("example")
            .collection("users")
            .find({username: req.body['username']});
        cursor.toArray(function(_, result) {
            res.send(result);
        });
    });
});

// Tell Express to start our server and listen on port 3000
app.listen(3000, () => {
  console.log(`Listening...`);
});
```

Note: In practice this would likely be a GET request like `/api/v1/getUser/<username>`, but for the sake of simplicity it is a POST here.

The intended use of this endpoint looks like this:

mayala@htb[/htb] `$ curl -s -X POST http://127.0.0.1:3000/api/v1/getUser -H 'Content-Type: application/json' -d '{"username": "gerald1992"}' | jq [ { "_id": "63667326b7417b004543513a", "username": "gerald1992", "password": "0f626d75b12f77ede3822843320ed7eb", "role": 1, "email": "g.sanchez@yahoo.com" } ]`

We posted the `/api/v1/getUser` endpoint with the body `{"username": "gerald1992"}`, and the server used that to generate the full query `db.users.find({username: "gerald1992"})` and returned the results to us.

The problem is that the server blindly uses whatever we give it as the username query without any filters or checks. Below is an example of code that is vulnerable to `NoSQL injection`:

Code: javascript

```javascript
...
.find({username: req.body['username']});
...
```

A simple example of exploiting this injection vulnerability is using the `$regex` operator described in the previous section to coerce the server into returning the information of all users (whose usernames match `/.*/`), like this:

mayala@htb[/htb] `$ curl -s -X POST http://127.0.0.1:3000/api/v1/getUser -H 'Content-Type: application/json' -d '{"username": {"$regex": ".*"}}' | jq [{ "_id": "63667302b7417b0045435139", "username": "bmdyy", "password": "f25a2fc72690b780b2a14e140ef6a9e0", "role": 0, "email": "football55@gmail.com" }, { "_id": "63667326b7417b004543513a", "username": "gerald1992", "password":`

```
"0f626d75b12f77ede3822843320ed7eb", "role": 1, "email": "g.sanchez@yahoo.com" }
]
```

Types of NoSQL Injection

If you are familiar with SQL injection, then you will already be familiar with the various classes of injections that we may encounter:

- `In-Band`: When the attacker can use the same channel of communication to exploit a NoSQL injection and receive the results. The scenario from above is an example of this.
- `Blind`: This is when the attacker does not receive any direct results from the NoSQL injection, but they can infer results based on how the server responds.
- `Boolean`: Boolean-based is a subclass of blind injections, which is a technique where attackers can force the server to evaluate a query and return one result or the other if it is True or False.
- `Time-Based`: Time-based is the other subclass of blind injections, which is when attackers make the server wait for a specific amount of time before responding, usually to indicate if the query is evaluated as True or False.

Moving On

With the basics of NoSQL injection explained, let's move on to some more in-depth examples.