*The development team that handles gift requests from Children migrated over to a new technology stack. In doing so, they left their application vulnerable, and Grinch Enterprises now controls access to the system. Fortunately, Grinch enterprises forgot to patch the system so you can use the same vulnerability to retrieve the gift requests for the students.*

**Please note**: the video walkthrough uses the in-browser access to complete the room. The instructions in the task are slightly different and make use of the **AttackBox/OpenVPN** to ssh into the target machine.

**Learning Objectives**

* What is NoSQL?
* Understanding NoSQL database
* Understand Why NoSQL happens
* Understand what NoSQL injection is
* Using NoSQL Injection to bypass a login form

**What is NoSQL**

A NoSQL database refers to a non-relational database that is short for non SQL and Not only SQL. It is a data-storing and data-retrieving system. NoSQL databases are commonly used nowadays for big Data and IoT devices due to their powerful features such as fast queries, ease of use to developers, scale easily, and flexible data structure.

Today, we will cover the basic concepts of NoSQL databases and information gathering, enumeration, and exploiting NoSQL vulnerabilities.

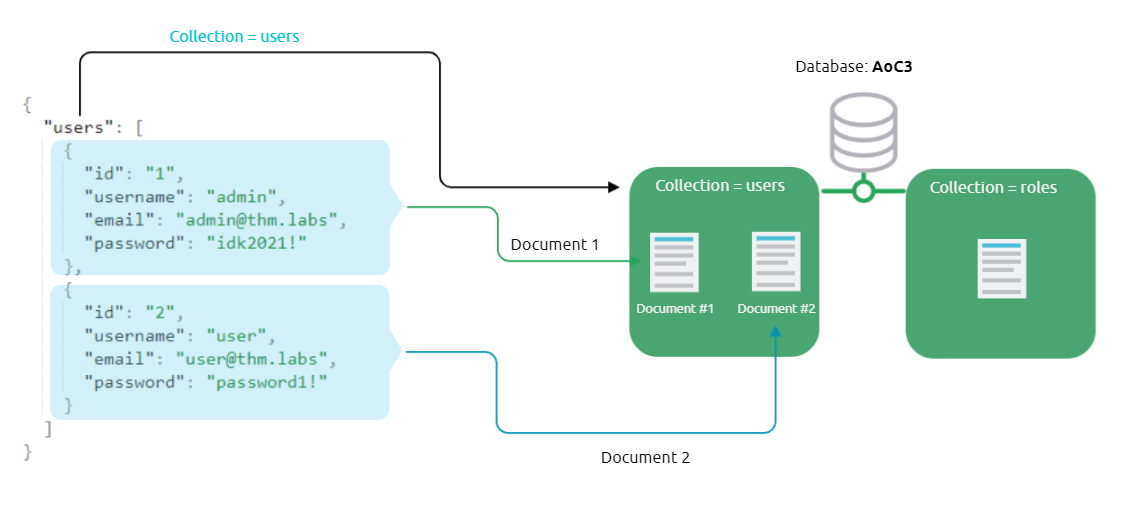
**Understanding NoSQL**

Various types of NoSQL databases can be covered, including MongoDB, Couchbase, RavenDB, etc. However, in this task, we will focus on MongoDB database, a free and popular document-store NoSQL database.

Similar to relational databases (such as MySQL and MSSQL), MongoDB consists of databases, tables, fields but with different names where

* **Collections**are similar to tables or views in MySQL and MSSQL.
* **Documents**are similar to rows or records in MySQL and MSSQL.
* **Fields**are similar to columns in MySQL and MSSQL.

The following graph shows a visual example of these terms as we have a database named AoC3 that has two collections: users, roles. The users collection has two documents (2 records). Documents in MongoDB are objects stored in a format called [BSON](https://bsonspec.org/), which supports [JSON](https://www.json.org/json-en.html) data types for document storing.



Also, it is useful to briefly look at and compare the query [operators](https://docs.mongodb.com/manual/reference/operator/query/)between MongoDB and MySQL:

* $and equivalent to AND in MySQL
* $or equivalent to OR in MySQL
* $eq equivalent to = in MySQL

**Interacting with a MongoDB server**

**Note**: To follow up with interacting with the MongoDB server, please deploy the attached machine and use the AttackBox or connect to the VPN in order to log into the SSH server. Use the following credentials:thm:tryhackme.

Terminal

user@machine**$** ssh thm@MACHINE\_IP -p 2222

Before we dive into the details of NoSQL injection, it is important to understand how MongoDB works in a general sense. Let's start with connecting to MongoDB on our deployed target machine using the default port 27017. We will be creating a new database and collections as well as documents.

MongoDB

user@machine**$** mongo

connecting to: mongodb://127.0.0.1:27017/?compressors=disabled&gssapiServiceName=mongodb

Implicit session: session { "id" : UUID("1273eab2-4dae-42a8-8f67-a133b870872b") }

MongoDB server version: 4.4.10

Now let's use the show command to list all the databases that we have in MongoDB on our target machine:

MongoDB

> show databases

admin 0.000GB

config 0.000GB

local 0.000GB

logindb 0.000GB

thm-test 0.000GB

thmdb 0.000GB

Next, if we want to connect to a database, we can do this by using the use command. However, we will also show how to create a new database with collections and data. To create a new database, we also use the same use command to create and connect to it. Therefore, the use command is used to connect to a database if it exists or create a new one if it doesn't exist. Once the database is created, we can create two new collections, named users and roles using the db.createCollection() function, and then show all available collections within the database.

MongoDB

> use AoC3

switched to db AoC3

> db.createCollection("users")

{ "ok" : 1 }

> db.createCollection("roles")

{ "ok" : 1 }

> db.getCollectionNames();

[ "roles", "users" ]

Next, we create a document within the users collection and insert data into it.

MongoDB

> db.users.insert({id:"1", username: "admin", email: "admin@thm.labs", password: "idk2021!"})

WriteResult({ "nInserted" : 1 })

> db.users.insert({id:"2", username: "user", email: "user@thm.labs", password: "password1!"})

WriteResult({ "nInserted" : 1 })

>

> db.users.find()

{ "\_id" : ObjectId("6183dc871ebe3f0c4b779a31"), "id" : "1", "username" : "admin", "email" : "admin@thm.labs", "password" : "idk2021!" }

{ "\_id" : ObjectId("6183dc911ebe3f0c4b779a32"), "id" : "2", "username" : "user", "email" : "user@thm.labs", "password" : "password1!" }

We successfully created two documents into users collection and then showed available documents within the collection using db.users.find(). Note that MongoDB automatically creates a unique ID called \_id for each document within the collection from the above output. Let's also try to update the document as well as delete it using MongoDB commands. We will be using the db.<collection>.update  function to update the document with id=2 and update the username to be tryhackme and finally shows all documents to make sure our document gets updated.

MongoDB

> db.users.update({id:"2"}, {$set: {username: "tryhackme"}});

WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })

> db.users.find()

{ "\_id" : ObjectId("6183dc871ebe3f0c4b779a31"), "id" : "1", "username" : "admin", "email" : "admin@thm.labs", "password" : "idk2021!" }

{ "\_id" : ObjectId("6183dc911ebe3f0c4b779a32"), "id" : "2", "username" : "tryhackme", "email" : "user@thm.labs", "password" : "password1!" }

Finally, let's remove the document using db.users.remove()  and then drop db.users.drop() the collection as follows,

MongoDB

> db.users.remove({'id':'2'})

WriteResult({ "nRemoved" : 1 })

> db.users.find()

{ "\_id" : ObjectId("6183dc871ebe3f0c4b779a31"), "id" : "1", "username" : "admin", "email" : "admin@thm.labs", "password" : "idk2021!" }

> db.users.drop()

true

Now that we know how to interact with the MongoDB server, look around by using the MongoDB commands we've learned, and find the flag to answer Question 1!

**What is NoSQL Injection?**

Now that we have the basic knowledge of dealing with MongoDB commands to create databases and collections as well as insert, update, and delete documents, we will be discussing the NoSQL injection and the risks of having such vulnerability in the application. NoSQL injection is a web security vulnerability that allows the attacker to have control over the database. A NoSQL injection happens by sending queries via untrusted and unfiltered web application input, which leads to leaked unauthorized information. In addition, the attacker can use the NoSQL injection to perform various operations such as modifying data, escalating privileges, DoS attacks, and others.

**Bypassing login pages!**

The logic of login pages is similar in most databases: first, connect to the database and then look for a certain username and password; if they exist in the collection (in the database), then we have a valid entry. The following is the query that is used in the web applications used on our login page:  db.users.find({query}) or db.users.findOne(query)  functions where the query is JSON data that's send via the application:  {"username": "admin", "password":"adminpass"}.  Note that when we provide the correct credentials, a document returns, while a null reply is received when providing the wrong credentials when nothing matches!

MongoDB

> db.users.findOne({username: "admin", password: "adminpass"})

{

"\_id" : ObjectId("6183ef6292dea43d75f0c820"),

"id" : "1",

"username" : "admin",

"email" : "admin@thm.labs",

"password" : "adminpass"

}

> db.users.findOne({username: "admin", password: "adminpasss"})

null

Before exploiting the NoSQL injection, there are MongoDB operators that we need to be familiar with that are heavily used in the injections, which are:

$eq - matches records that equal to a certain value

$ne - matches records that are not equal to a certain value

$gt - matches records that are greater than a certain value.

$where - matches records based on Javascript condition

$exists - matches records that have a certain field

$regex - matches records that satisfy certain regular expressions.

Visit the [MongoDB website](https://docs.mongodb.com/manual/reference/operator/query/) for more information about the MongoDB operators. Next, we will be exploiting the logic of the login query by injecting a JSON object which includes one of the NoSQL operators, which is $ne.

MongoDB

> db.users.findOne({username: "admin", password: {"$ne":"xyz"}})

{

"\_id" : ObjectId("6183ef6292dea43d75f0c820"),

"id" : "1",

"username" : "admin",

"email" : "admin@thm.labs",

"password" : "adminpass"

}

We injected a JSON objection {"$ne": "XYZ"}  in the password field, and we changed the logic to become as follows:

* We are telling MongoDB to find a document (user) with a username equal to admin and his password is not equal to xyz, which turns this statement to TRUE because the admin's password is not xyz.

As a result, we have successfully retrieved the document from MongoDB since our statement's logic is true. By applying this concept against a login page of a web application, we will be able to bypass the login page.

Now let's say if we wanted to log in to a system as another user who is not admin. In this case, we can also inject into a username field to be as follows,

MongoDB

> db.users.findOne({username:{"$ne":"admin"},password:{"$ne":"xyz"}})

{

"\_id" : ObjectId("6183ef5b92dea43d75f0c81f"),

"id" : "2",

"username" : "user",

"email" : "user@thm.labs",

"password" : "password1!"

}

If this were a login page, we would be logged in as a not admin, which is the user. We injected two JSON objects into a username as well as password fields: we are telling MongoDB to find a document that its username is not equal to admin and its password is not equal to xyz, which returns the statement as true.

**Exploiting NoSQL injection**

To exploit NoSQL injection within the web application, first, you need to find an entry point that doesn't sanitize the user's input. Next, you need to understand how the web application passes the request to the database! Once you find an entry point, passing queries could be varied. Sometimes, the web app accepts the user's input via GET or POST queries, and sometimes web applications accept a JSON object, as is the case with APIs.

Injecting a NoSQL query has different forms if we deal with GET or POST queries than JSON objects but still have the same concept. Let's discuss how to inject into regular  GET or POST requests. To interact with MongoDB via GET or POST is by injecting an array of the MongoDB operator to match the JSON objection to match the Key: Value. The following is an example of how to inject via URL:

http://example.thm.labs/search?username=admin&role[$ne]=user

Note that we inject the MongoDB operator [$ne] (not equal) in the role parameter. We can also use the same concept to pass the MongoDB operators in the POST requests.

We built a playground web application to search for the user(s) with a specific role! To perform NoSQL injection, first, you need to bypass the login page, as we discussed in the previous section.

Let's see the normal case where we search for username is equal ben with the user role.

http://example.thm.labs/search?username=ben&role=user

As a result, the web application returns the expected result from MongoDB. Now let's inject MongoDB to show unexpected results! We will try to list all usernames that have a user role!

http://example.thm.labs/search?username[$ne]=ben&role=user

As a result, the web application will return many records from MongoDB that satisfy the query. It is practice time; now make sure you have deployed the attached machine and the AttackBox (to use Burp), try to list all usernames that have guest roles. **Find the flag and answer question #2 below**

You can access and visit the website via the following link: http://LAB\_WEB\_URL.p.thmlabs.com or http://MACHINE\_IP in case you're using the AttackBox.

http://10.10.216.111/search?username[$ne]=ben&role=guest