# PortSwigger NoSQL Injection Lab Notes

1. Detecting NoSQL injection

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

To solve the lab, perform a NoSQL injection attack that causes the application to display unreleased products.

Solution

1. In Burp's browser, access the lab and click on a product category filter.
2. In Burp, go to **Proxy > HTTP history**. Right-click the category filter request and select **Send to Repeater**.

A screenshot of a computer

AI-generated content may be incorrect.

1. In Repeater, submit a ' character in the category parameter. Notice that this causes a JavaScript syntax error. This may indicate that the user input was not filtered or sanitized correctly.

A screenshot of a computer

AI-generated content may be incorrect.

1. Submit a valid JavaScript payload in the value of the category query parameter. You could use the following payload:

***Gifts'+'***

Make sure to URL-encode the payload by highlighting it and using the Ctrl-U hotkey. Notice that it doesn't cause a syntax error. This indicates that a form of server-side injection may be occurring.

1. Identify whether you can inject boolean conditions to change the response:
   1. Insert a false condition in the category parameter. For example:

***Gifts' && 0 && 'x***

Make sure to URL-encode the payload. Notice that no products are retrieved.

* 1. Insert a true condition in the category parameter. For example:

***Gifts' && 1 && 'x***

Make sure to URL-encode the payload. Notice that products in the **Gifts** category are retrieved.

1. Submit a boolean condition that always evaluates to true in the category parameter. For example:

***Gifts'||1||'***

1. Right-click the response and select **Show response in browser**.
2. Copy the URL and load it in Burp's browser. Verify that the response now contains unreleased products. The lab is solved.
3. Exploiting NoSQL operator injection to 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.

it builds a MongoDB query like:

db.users.findOne({ username: input\_username, password: input\_password })

MongoDB allows **special operators** inside queries (e.g. $ne, $regex).  
If the app doesn’t sanitize inputs, we can inject these operators instead of normal strings.

Solution

1. In Burp's browser, log in to the application using the credentials wiener:peter.
2. In Burp, go to Proxy > HTTP history. Right-click the POST /login request and select Send to Repeater.
3. In Repeater, test the username and password parameters to determine whether they allow you to inject MongoDB operators:
   1. Change the value of the username parameter from "***wiener" to {"$ne":""},*** then send the request. Notice that this enables you to log in.

* ($ne = "not equal")
* This tells MongoDB: *“find a user whose username is not an empty string”* → basically **any user**.
* You get logged in without knowing the username.
  1. Change the value of the username parameter from ***{"$ne":""} to {"$regex":"wien.\*"},*** then send the request. Notice that you can also log in when using the $regex operator.
*  ($regex = regular expression match)
*  Matches any username starting with "wien".
*  Logs you in as wiener.
  1. With the username parameter set to ***{"$ne":""},*** change the value of the password parameter from "***peter" to {"$ne":""},*** then send the request again. Notice that this causes the query to return an unexpected number of records. This indicates that more than one user has been selected.
* This means “password not equal to empty string” → matches for every user with a non-empty password.
* Multiple users are matched → the app is confused.

1. With the password parameter set ***as {"$ne":""},*** change the value of the username parameter to ***{"$regex":"admin.\*"},*** then send the request again. Notice that this successfully logs you in as the admin user.

*  Keep password as {"$ne":""}.
*  This matches the admin user with any password that isn’t empty.
*  You are now logged in as the administrator.

1. Right-click the response, then select Show response in browser. Copy the URL.
2. Paste the URL into Burp's browser to log in as the administrator user. The lab is solved.
3. Exploiting NoSQL injection to extract data

The user lookup functionality for this lab is powered by a MongoDB NoSQL database. It is vulnerable to NoSQL injection.

To solve the lab, extract the password for the administrator user, then log in to their account.

You can log in to your own account using the following credentials: wiener:peter.

**Tip**

The password only uses lowercase letters.

this lab is about using NoSQL injection (in MongoDB) not just to bypass login, but to extract the actual admin’s password.

The app does a **user lookup** like:

db.users.findOne({ username: input })

If the input isn’t sanitized, you can inject **JavaScript conditions** into MongoDB queries. This lets you ask yes/no questions about the data (like the password length or specific characters) and infer the answer from how the app responds.

Solution

1. In Burp's browser, access the lab and log in to the application using the credentials wiener:peter.
2. In Burp, go to **Proxy > HTTP history**. Right-click the GET /user/lookup?user=wiener request and select **Send to Repeater**.
3. In Repeater, submit a ' character in the user parameter. Notice that this causes an error. This may indicate that the user input was not filtered or sanitized correctly.
4. Submit a valid JavaScript payload in the user parameter. For example, you could use wiener'+'

Make sure to URL-encode the payload by highlighting it and using the hotkey Ctrl-U. Notice that it retrieves the account details for the wiener user, which indicates that a form of server-side injection may be occurring.

**Confirm injection works**

* Normal request:
* /user/lookup?user=wiener
* Test with ' → error → **input is unsanitized**.
* Test with trick string:
* wiener'+'

→ still returns Wiener’s details.  
✅ Means the query is being evaluated as code, so injection is possible.

1. Identify whether you can inject boolean conditions to change the response:
   1. Submit a false condition in the user parameter. For example: wiener' && '1'=='2

Make sure to URL-encode the payload. Notice that it retrieves the message Could not find user.

* 1. Submit a true condition in the user parameter. For example: wiener' && '1'=='1

Make sure to URL-encode the payload. Notice that it no longer causes an error. Instead, it retrieves the account details for the wiener user. This demonstrates that you can trigger different responses for true and false conditions.

**Check true/false conditions**

* Inject a false condition:
* wiener' && '1'=='2

→ response: *Could not find user*.

* Inject a true condition:
* wiener' && '1'=='1

→ response: still shows Wiener’s account.

✅ Confirms we can toggle behavior with conditions → we have a **boolean-based injection**.

1. Identify the password length:
   1. Change the user parameter to administrator' && this.password.length < 30 || 'a'=='b, then send the request.

Make sure to URL-encode the payload. Notice that the response retrieves the account details for the administrator user. This indicates that the condition is true because the password is less than 30 characters.

* 1. Reduce the password length in the payload, then resend the request.
  2. Continue to try different lengths.
  3. Notice that when you submit the value 9, you retrieve the account details for the administrator user, but when you submit the value 8, you receive an error message because the condition is false. This indicates that the password is 8 characters long.

1. Right-click the request and select **Send to Intruder**.
2. In Intruder, enumerate the password:
   1. Change the user parameter to administrator' && this.password[§0§]=='§a§. This includes two payload positions. Make sure to URL-encode the payload.
   2. Select **Cluster bomb attack** from the attack type drop-down menu.
   3. In the **Payloads** side panel, select position 1 from the **Payload position** drop-down list. Add numbers from 0 to 7 for each character of the password.
   4. Select position 2 from the **Payload position** drop-down list, then add lowercase letters from a to z. If you're using Burp Suite Professional, you can use the built-in a-z list.
   5. Click **Start attack**.
   6. Sort the attack results by **Payload 1**, then **Length**. Notice that one request for each character position (0 to 7) has evaluated to true and retrieved the details for the administrator user. Note the letters from the **Payload 2** column down.
3. In Burp's browser, log in as the administrator user using the enumerated password. The lab is solved.
4. Exploiting NoSQL operator injection to extract unknown fields

The user lookup functionality for this lab is powered by a MongoDB NoSQL database. It is vulnerable to NoSQL injection.

To solve the lab, log in as carlos.

**Tip**

To solve the lab, you'll first need to exfiltrate the value of the password reset token for the user carlos.

Solution

1. In Burp's browser, attempt to log in to the application with username carlos and password invalid. Notice that you receive an Invalid username or password error message.
2. In Burp, go to **Proxy > HTTP history**. Right-click the POST /login request and select **Send to Repeater**.
3. In Repeater, change the value of the password parameter from "invalid" to {"$ne":"invalid"}, then send the request. Notice that you now receive an Account locked error message. You can't access Carlos's account, but this response indicates that the $ne operator has been accepted and the application is vulnerable.
4. In Burp's browser, attempt to reset the password for the carlos account. When you submit the carlos username, observe that the reset mechanism involves email verification, so you can't reset the account yourself.
5. In Repeater, use the POST /login request to test whether the application is vulnerable to JavaScript injection:
   1. Add "$where": "0" as an additional parameter in the JSON data as follows: {"username":"carlos","password":{"$ne":"invalid"}, "$where": "0"}
   2. Send the request. Notice that you receive an Invalid username or password error message.
   3. Change "$where": "0" to "$where": "1", then resend the request. Notice that you receive an Account locked error message. This indicates that the JavaScript in the $where clause is being evaluated.
6. Right-click the request and select **Send to Intruder**.
7. In Intruder, construct an attack to identify all the fields on the user object:
   1. Update the $where parameter as follows: "$where":"Object.keys(this)[1].match('^.{}.\*')"
   2. Add two payload positions. The first identifies the character position number, and the second identifies the character itself: "$where":"Object.keys(this)[1].match('^.{§§}§§.\*')"
   3. Select **Cluster bomb attack** from the attack type drop-down menu.
   4. In the **Payloads** side panel, select position 1 from the **Payload position** drop-down list, then set the **Payload type** to **Numbers**. Set the number range, for example from 0 to 20.
   5. Select position 2 from the **Payload position** drop-down list and make sure the **Payload type** is set to **Simple list**. Add all numbers, lower-case letters and upper-case letters as payloads. If you're using Burp Suite Professional, you can use the built-in word lists a-z, A-Z, and 0-9.
   6. Click **Start attack**.
   7. Sort the attack results by **Payload 1**, then **Length**, to identify responses with an Account locked message instead of the Invalid username or password message. Notice that the characters in the **Payload 2** column spell out the name of the parameter: username.
   8. Repeat the above steps to identify further JSON parameters. You can do this by incrementing the index of the keys array with each attempt, for example: "$where":"Object.keys(this)**[2]**.match('^.{}.\*')"

Notice that one of the JSON parameters is for a password reset token.

1. Test the identified password reset field name as a query parameter on different endpoints:
   1. In **Proxy > HTTP history**, identify the GET /forgot-password request as a potentially interesting endpoint, as it relates to the password reset functionality. Right-click the request and select **Send to Repeater**.
   2. In Repeater, submit an invalid field in the URL: GET /forgot-password?foo=invalid. Notice that the response is identical to the original response.
   3. Submit the exfiltrated name of the password reset token field in the URL: GET /forgot-password?YOURTOKENNAME=invalid. Notice that you receive an Invalid token error message. This confirms that you have the correct token name and endpoint.
2. In Intruder, use the POST /login request to construct an attack that extracts the value of Carlos's password reset token:
   1. Keep the settings from your previous attack, but update the $where parameter as follows: "$where":"this.YOURTOKENNAME.match('^.{§§}§§.\*')"

Make sure that you replace YOURTOKENNAME with the password reset token name that you exfiltrated in the previous step.

* 1. Click **Start attack**.
  2. Sort the attack results by **Payload 1**, then **Length**, to identify responses with an Account locked message instead of the Invalid username or password message. Note the letters from the **Payload 2** column down.

1. In Repeater, submit the value of the password reset token in the URL of the GET / forgot-password request: GET /forgot-password?YOURTOKENNAME=TOKENVALUE.
2. Right-click the response and select **Request in browser > Original session**. Paste this into Burp's browser.
3. Change Carlos's password, then log in as carlos to solve the lab