Day 7 : Web Exploitation

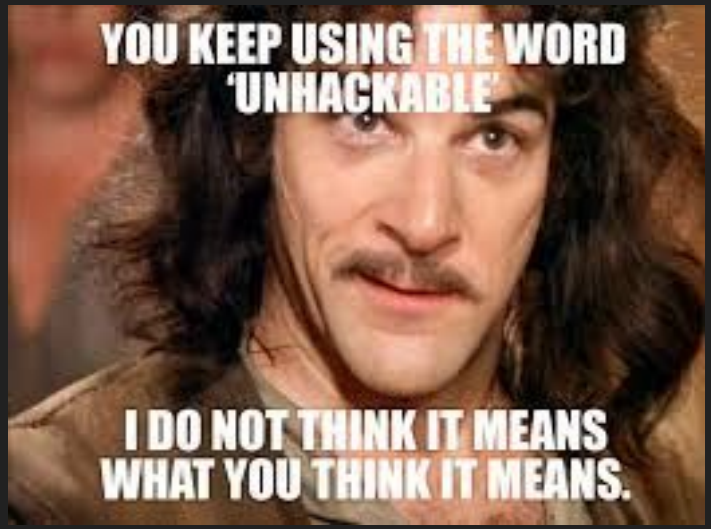
~ [CCSC] CIT Cyber Security Cell ~
OUSSAMA RAHALI
OMAR AOUAJ

CLUB INFORMATIQUE & TÉLÉCOM

cat README.md

Presentation outline

1. Web Exploitation :
   - 1.1 Introduction & Fundamentals
   - 1.2 SQL Injection
   - 1.3 Command Injection
   - 1.4 Directory Traversal
   - 1.5 Cross Site Request Forgery
   - 1.6 Server Side Request Forgery
   - 1.7 XSS



Introduction & Fundamentals

Firstly, we should clarify that websites are NOT an abstract concept.. Each website is just a collection of files (scripts, databases,...) located in a distant machine called Server, while the website's user machine is called Client.

These two entities communicate with each other via the HTTP or the HTTPS protocol (which is an application-layer protocol) as the following:

The client requests the server for a resource. The server processes the request and returns the requested resource (as a response).

Introduction & Fundamentals

The default port for HTTP communication is 80 (443 for HTTPS); however, this can be changed.

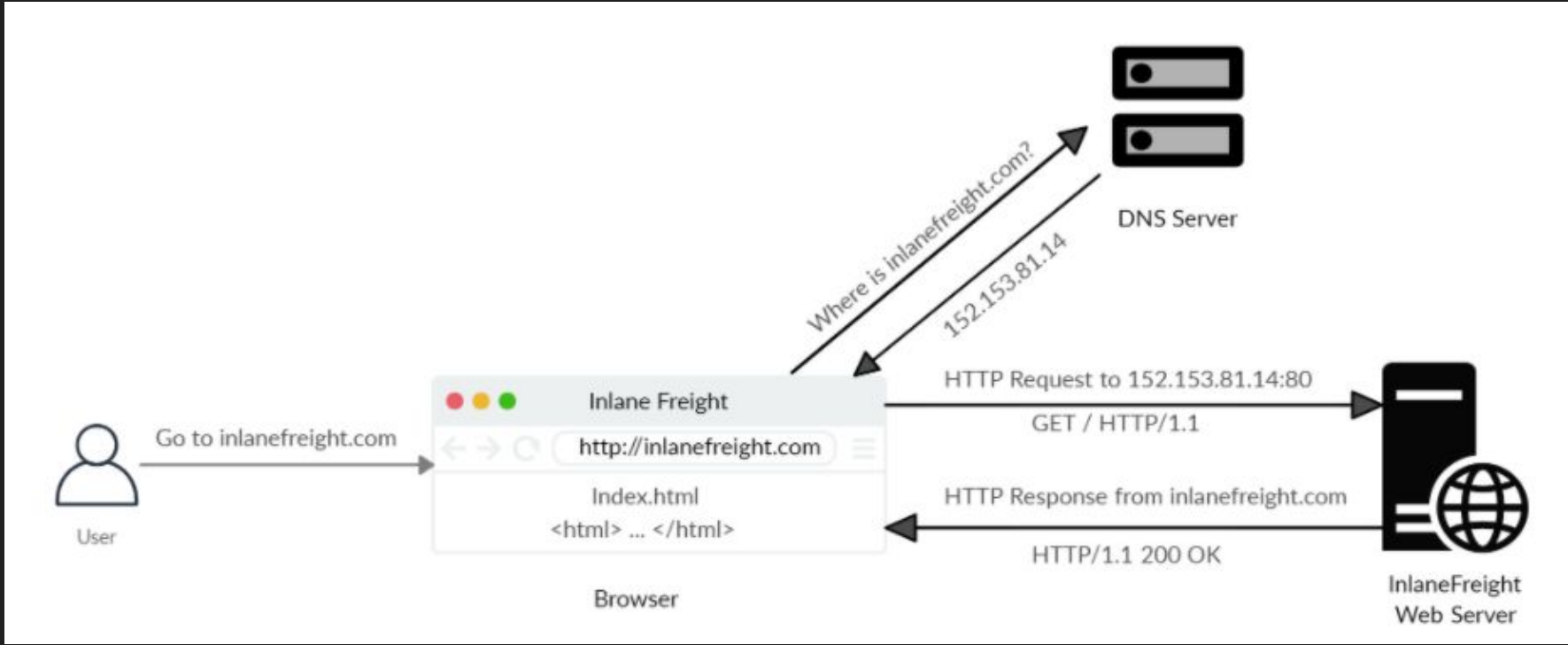
The client requests a resource from a server via a URL (Uniform Resource Locator) as the following example:

http://example.com:80/dashboard.php?login=true

protocol host or ip port path of resource query string or parameter

Introduction & Fundamentals

What really happens:

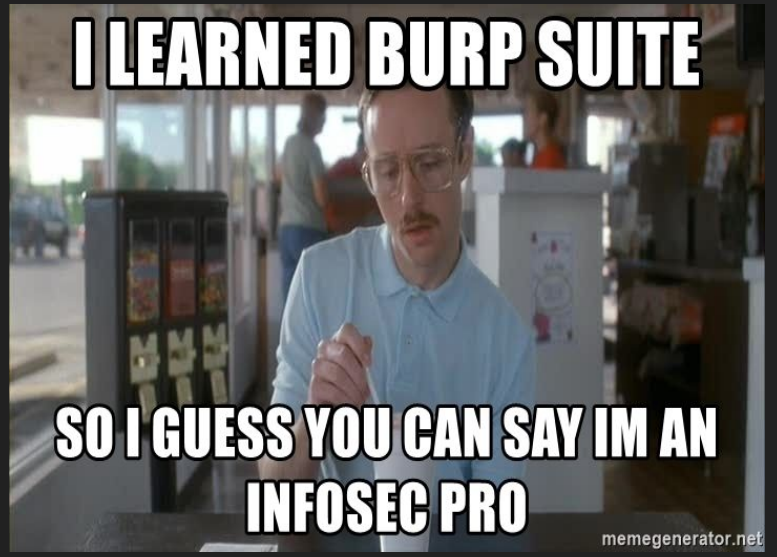


Introduction & Fundamentals

Burp Suite: an indispensable web tool

Before we move further, it's better to set Burp Suite tool and know how to use it.

It'll act as a **proxy server** (it will be located between our machine (as client) and the server). This will let us the HTTP traffic and even change it.



Introduction & Fundamentals

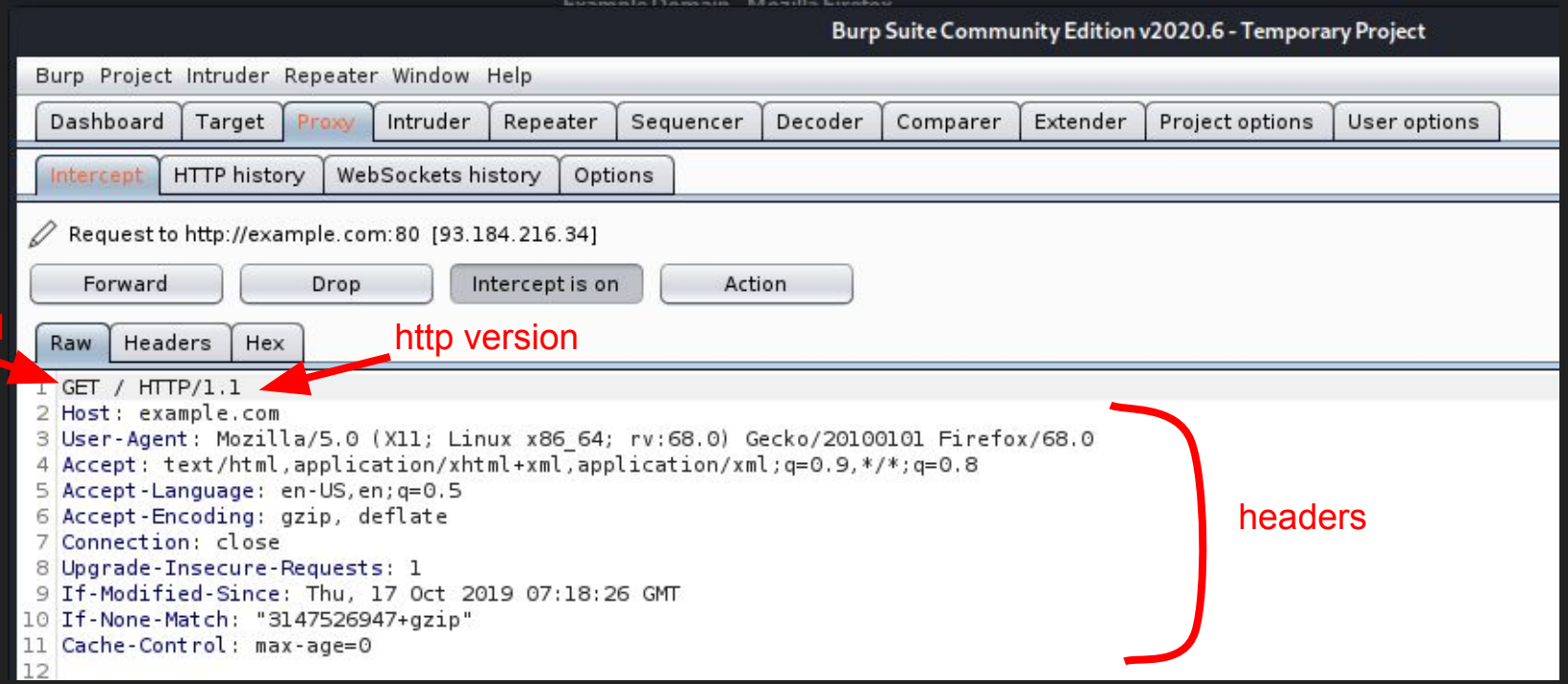
In Mozilla Firefox: Preferences->Search for proxy->Settings

In Burp Suite: proxy->intercept is on ->options->Proxy Listeners

Running	Interface	Invisible	Redirect	Certificate	TLS Protocols
☐	127.0.0.1:8080			Per-host	Default

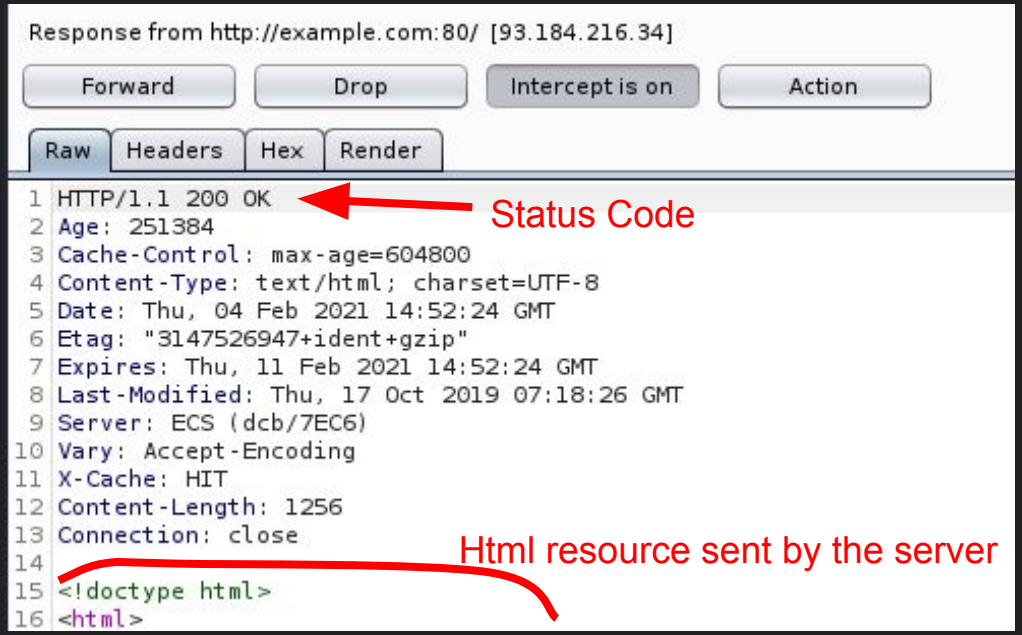
Introduction & Fundamentals

After this configuration, you can intercept your HTTP requests and responses. And by typing the url http://example.com, I can see the request in Burp Suite:



Introduction & Fundamentals

I can also see the response to my request (after forwarding the latter):



Introduction & Fundamentals

Headers:

General Headers: Describe the message (date, connection)
Entity Headers: Describe the content (Content-Length, COntent-Type)
Request Headers: Provide authorization, and context of the request (Host, Authorization, Cookie, User-Agent)
Response Headers: Provide context of the response (Server, WWW-authenticate, Set-Cookie)
Security Headers: Provide security of the web application (Strict-Transport-Security, Content-Security-Policy)

Introduction & Fundamentals

Methods:

GET: requests a specific resource.
POST: send data to the server. It can handle multiple types of input, such as text, PDFs, and other forms of binary data.
HEAD: requests the headers that would be returned if a GET request was made to the server. It doesn't return the request body and is usually made to check the response length before downloading resources.
PUT: similar to POST, yet idempotent.
DELETE: lets users delete an existing resource on the web server.

Introduction & Fundamentals

Codes:

1xx: Usually provides information and continues processing the request.
2xx: Positive response codes returned when a request succeeds.
3xx: Returned when the server redirects the client.
4xx: This class of codes signifies improper requests from the client. For example, requesting a resource that doesn't exist or requesting a bad format.
5xx: Returned when there is some problem with the HTTP server itself.

SQL Injection

It's a vulnerability related to user input, the web application doesn't validate that this user input doesn't contain additional SQL.

It's used to cause a data breach of information stored in a database without having authorized access to it, but by adding extra code to the query (inject code) to perform a new one, it can even delete an entire table.

SQL Injection

In order to understand its concept, we'll do a live demo.

SQL Injection

Suppose we have a web server containing this php file (index.php)

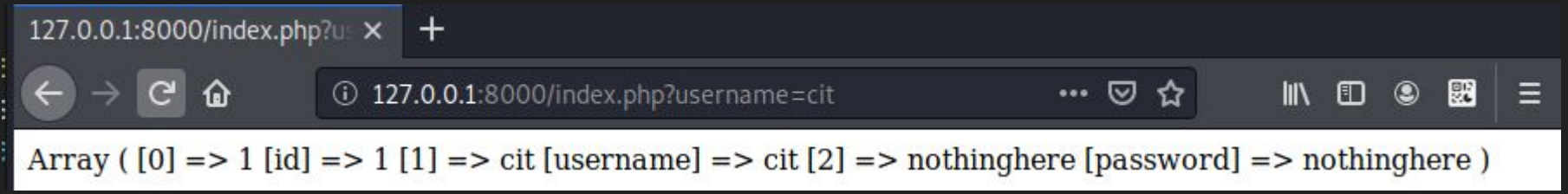
```php
<?php
    $mysqli = new mysqli("localhost","root","root","cit");

    // Check connection
    if ($mysqli -> connect_errno) {
    print("Failed to connect to MySQL: " . $mysqli -> connect_error);
    exit();
    }

    $username = $_GET['username'];
    $result = mysqli_query($mysqli,"SELECT * FROM user WHERE username='$username'");
    while($row = mysqli_fetch_array($result))
     {
        print_r($row);
     }
?>
```

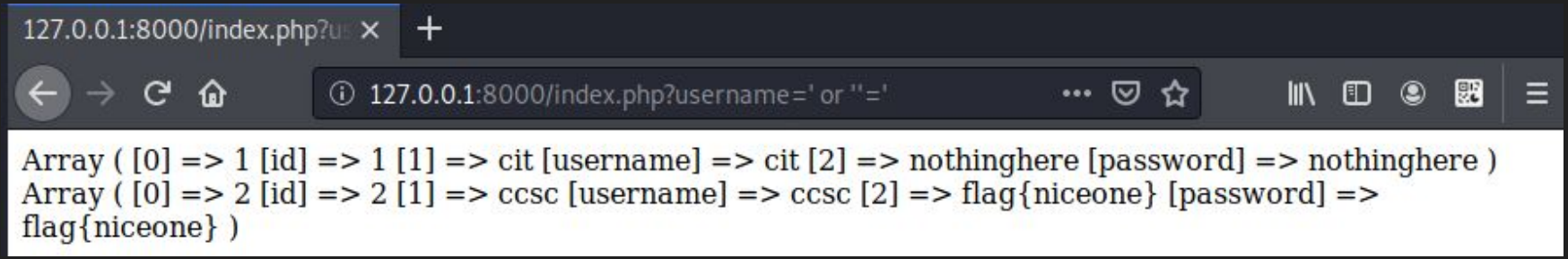
SQL Injection

Suppose we only know one user **cit** but we wanna see all the user table content (all usernames and **all passwords**)



SQL Injection

Using this payload: ' or '' = ' , we can see all the table content



So what did we do exactly? we simply injected a string inside the query which gave us all the table content.

In fact, this is the actual query executed by the server:

SELECT * FROM user WHERE username = '' or '' = ''

We should notice that the condition '' = '' is always true, therefore all the rows of the table verify it.

Command Injection

It's a vulnerability that allows an attacker to submit system commands to a server running a website. This happens when the application fails to encode user input that goes into a system shell.

It is very common to see this vulnerability when a developer uses the system() command or its equivalent in the programming language of the application.

Command Injection

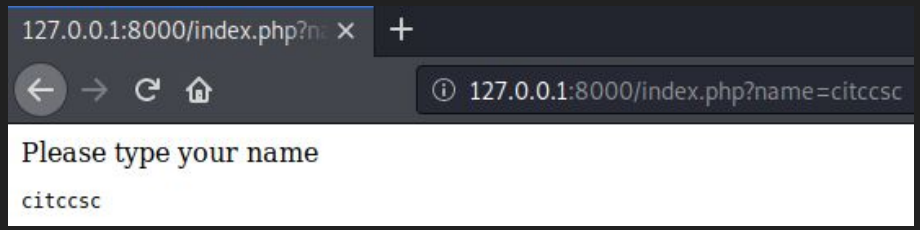
In order to understand its concept, we'll do a live demo.

Command Injection

Suppose we have a web server containing this php file (index.php)

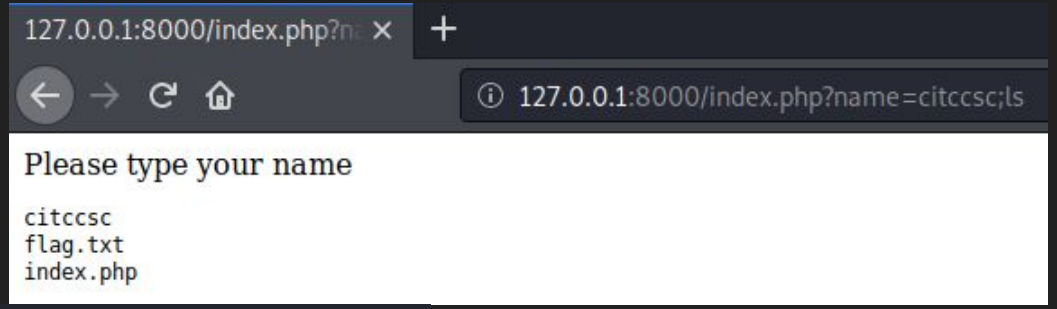
```php
<?php
    print("Please type your name in the url");
    print("<pre>");
    $name=$_GET["name"];
    system("echo $name");
?>
```

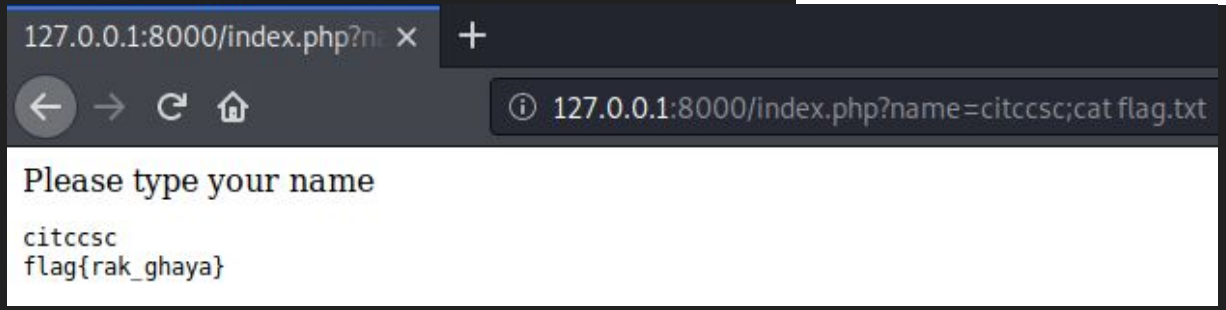
It's simply a script which takes the name of a user and prints it via the server's shell.



Command Injection

But how about this??





We managed to execute any command we want on the server even if the only command available is **echo**, why is that?

Well, it's simple, by adding **;ls** for example, the shell executes the following command: **echo citccsc; ls** which is like **echo citccsc && ls**.

Directory Traversal

It’s a vulnerability where a web application takes in user input and uses it in a directory path.

Any kind of path controlled by user input that isn't properly sanitized could be vulnerable to directory traversal.

Here is a live demo to understand it:

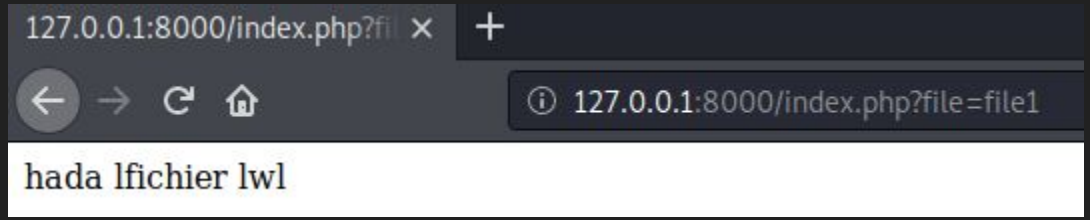
Directory Traversal

Suppose we have a web server containing this php file (index.php)

index.php

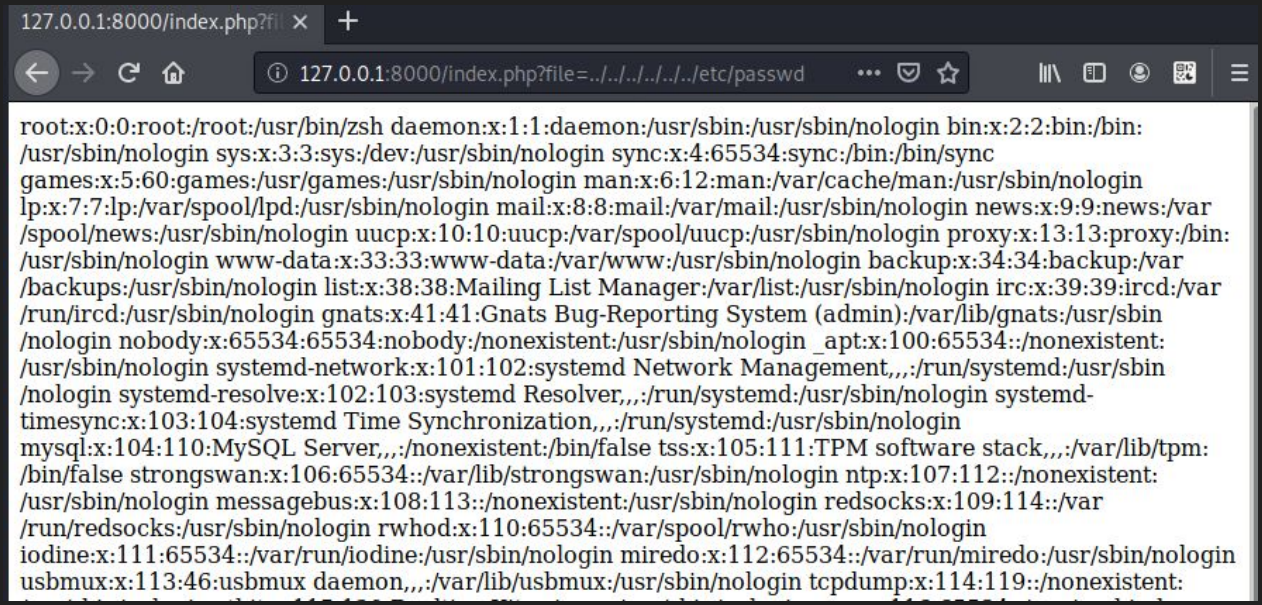
```php
<?php
    $file = $_GET['file'];
    include("./directory/" . $file);
?>
```

It's simply a script which provides users with the content of the requested file, but only if it is inside the directory **directory**



Directory Traversal

But how about this??



We managed to show the content of a file which should not be shown to us as simple users, yet since there is no filtration of our input, we managed to include **/etc/passwd** by going back to parent directories so the actual file will be **./directory/../../../../../etc/passwd**.

CSRF: Cross Site Request Forgery

It's an attack which can be used to make an authenticated user execute a request in favor of the attacker.

Many websites use cookies to keep a user authenticated during a session (from login to logout), so even if he clicks a link referring to the website from outside during the session, the action requested by the link will be perfectly performed (without demanding a login or an authentication).

CSRF: Cross Site Request Forgery

Let's take an example, suppose we have a web server where we can perform bank transfers via the following url:

`http://ccscBank.com/transfer?account=[ACCOUNT]&amount=[AMOUNT]`

An attacker chooses to send to a user this hyperlink via email:

```
<a href="http://ccscBank.com/transfer.?account=AttackerA&amount=$100">Click on me to win 5000$!</a>
```

If the user clicks on the link while he's connected to ccscBank.com, the request will be executed automatically.

SSRF: Server Side Request Forgery

It's a vulnerability which can be used by an attacker to cause a web application to send a request that the attacker defines.

Let's take an example, say there is a website that can print any page on the web (provides a pdf file which contains a screenshot of a web page so that you can download it).

A normal user can use it to print google.com or m.inpt.ac.ma.

SSRF: Server Side Request Forgery

What if a user does something more nefarious? What if they asked the site to print http://localhost ?

Or perhaps tries to access something more useful like http://localhost/server-status ?

He can even access ssh credentials if he knows its path, get to know the private details of the server or access the internal network of the server.

XSS: Cross Site Scripting

It's a vulnerability where a user of an application can send JavaScript that is executed by the browser of another user of the same application.

It's like injecting javascript code inside user input.

Let's take an example to illustrate it:

XSS: Cross Site Scripting

Context of the example:

A web application used to buy and sell cars, all the communication between clients and the server is encrypted and maintained via cookies.

Normal use of the application:

Bob wants to sell his car, he communicates with the server which identifies him via his cookie. Then he receives the following response (list of cars), he selects new entry to store his car in the website database.

After clicking the new entry button, he receives the following response, it's a form which he needs to fill.

Bob writes the name and the description of the car, the input of the description accepts html tags to customize the font or the boldness of what's written as the following.

Alice, on the other hand, wants to buy a car, so she communicates with the server and receives the list of available cars for sale.

Alice clicks on read more to check the first car, which is Bob's. Therefore, she receives the name and the customized description which is processed by her browser to show her lot of extras... as bold.

Now we need to bring in another player, Mel, he wants to sort of hack this web server.

Good hacking starts with analysis, so Mel has to understand how the application works. So he identifies himself in the application, he'll basically do the same thing done by Alice. He sees Bob's offer and wonders: Why is there some text written in bold letters? Hmmm.. HTML tags are allowed!!

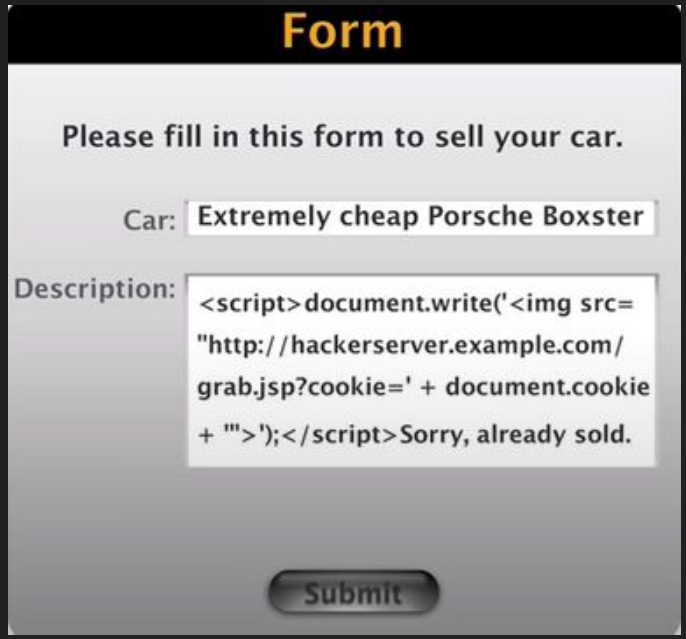
Let's see if Mel can exploit that, firstly, he needs a headline that grabs attention:

Extremely Cheap Porsche Boxster

Now what if he enters Javascript code instead of the description of his car??? And what if that code will read the cookie of whoever is opening that page and then send it to Mel's server??

Mel saves his input in the web application, all he has to do now is opening an HTTP server in his computer and wait.

P.S: you can do that by executing one of these commands:




```
python -m SimpleHTTPServer 80
            or
python3 -m http.server 80
```

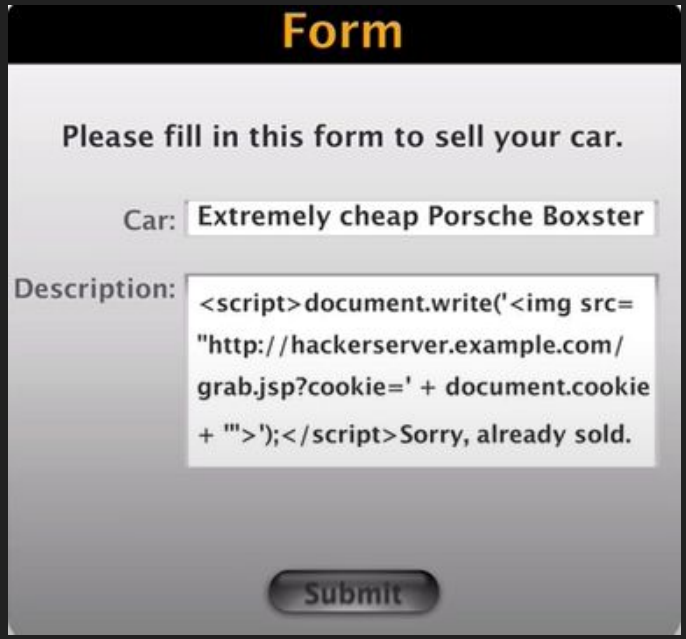
Let's see if Mel can exploit that, firstly, he needs a headline that grabs attention:

Extremely Cheap Porsche Boxster

Now what if he enters Javascript code instead of the description of his car??? And what if that code will read the cookie of whoever is opening that page and then send it to Mel's server??

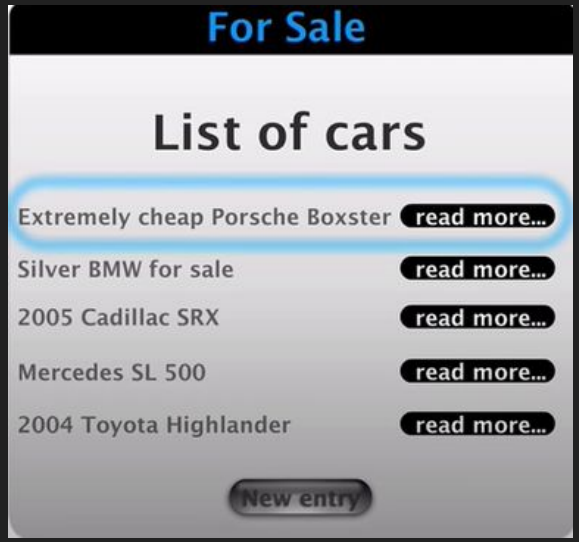
Mel saves his input in the web application, all he has to do now is opening an HTTP server in his computer and wait.

P.S: you can do that by executing one of these commands:




```
python -m SimpleHTTPServer 80
            or
python3 -m http.server 80
```

Alice wants to check new offers in the web application, the new list contains Mel malicious code. Mel's car grabs her attention, it's a Cheap Porsche!!



Once she clicks on read more, only the last part of the description will be displayed (Sorry, already sold).

The problem is that Alice's web browser executes the script embedded in the description since it sees it as an html tag, therefore, Alice's cookie will be delivered to Mel once she clicks on the read more button.

So Mel has stolen now Alice's digital identity during the connection session, he can simply send a request to the server with a header consisting of her cookie so that he can access her account.

Therefore, he can do whatever she can do, access her personal data, and even take over her account.

All in all, from the server pov, Mel is Alice.

XSS: Cross Site Scripting

Types of XSS:

- **Stored XSS:** The XSS payload is sent to the database and called once the page is loaded. The previous example is a Stored XSS in which the script sending the cookie is stored in the database until Alice called the page.

XSS: Cross Site Scripting

- **DOM* XSS:**

The DOM is how the document is represented in JS (defining its structure).

Sometimes a developer may use in his code something like location.hash* and then he outputs it in another part of the code.

What if we can modify this location.hash maliciously to make the code executes something way far from its normal functioning. Mostly we make it do an alert(1) as a proof of concept.

*DOM: Document Object Model

*location.hash: it just represents what's after the #

http://example.com/index.html#test

location.hash = "#test"

XSS: Cross Site Scripting

Example of DOM XSS:

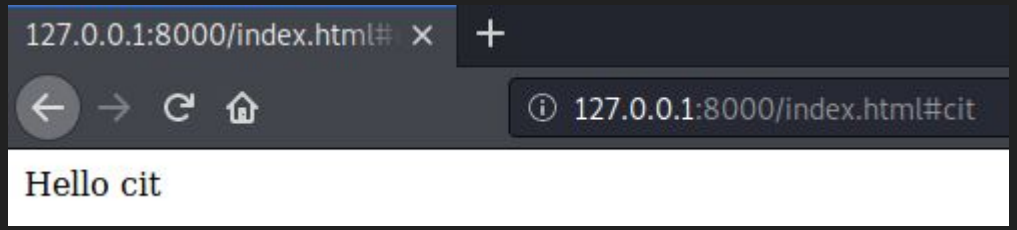
Let's consider a developer who used the following code:

```
<!DOCTYPE html>
<html>
    <body>
        <script>
            var source = "Hello " + decodeURIComponent(location.hash.split("#")[1]);
            var divElement = document.createElement("div");
            divElement.innerHTML = source;
            document.body.appendChild(divElement);
        </script>
    </body>
</html>
```

XSS: Cross Site Scripting

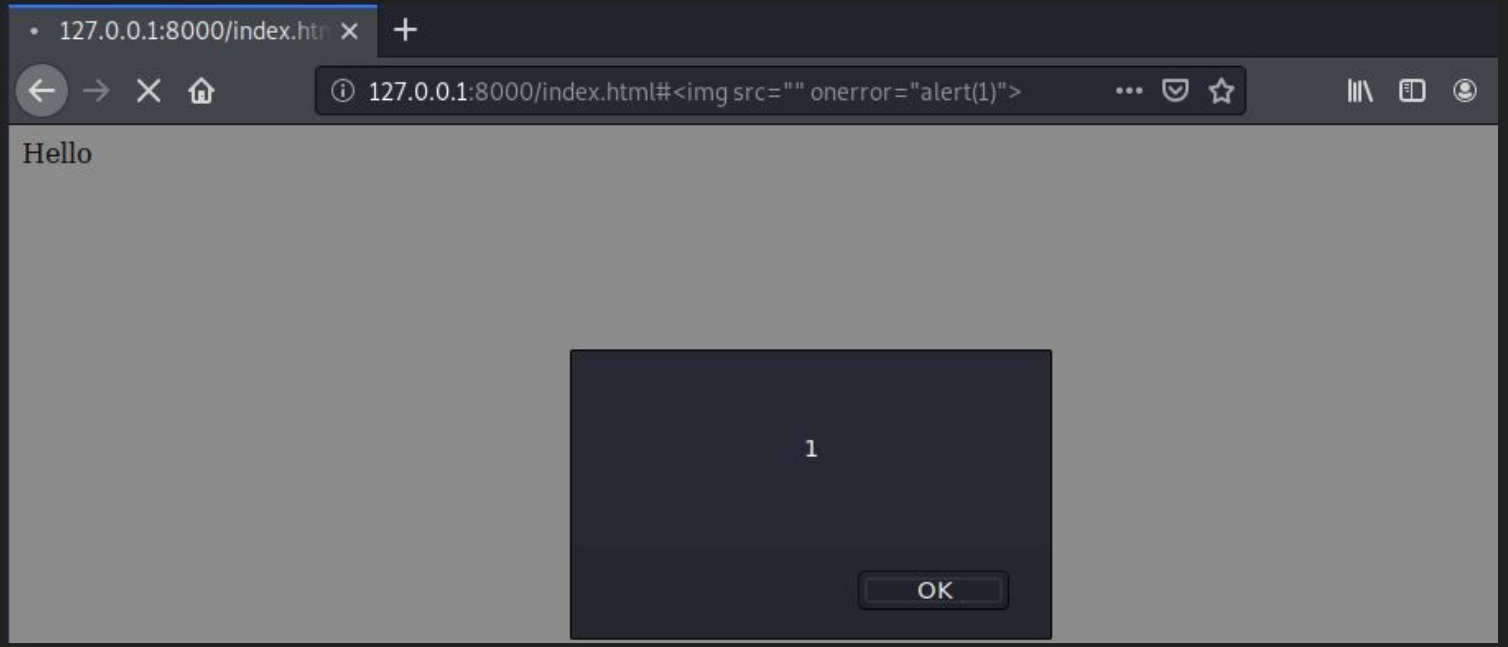
Example of DOM XSS:

A simple use of this application is as follows:



Yet we can make the application do an **alert(1)** as an example:

XSS: Cross Site Scripting



XSS: Cross Site Scripting

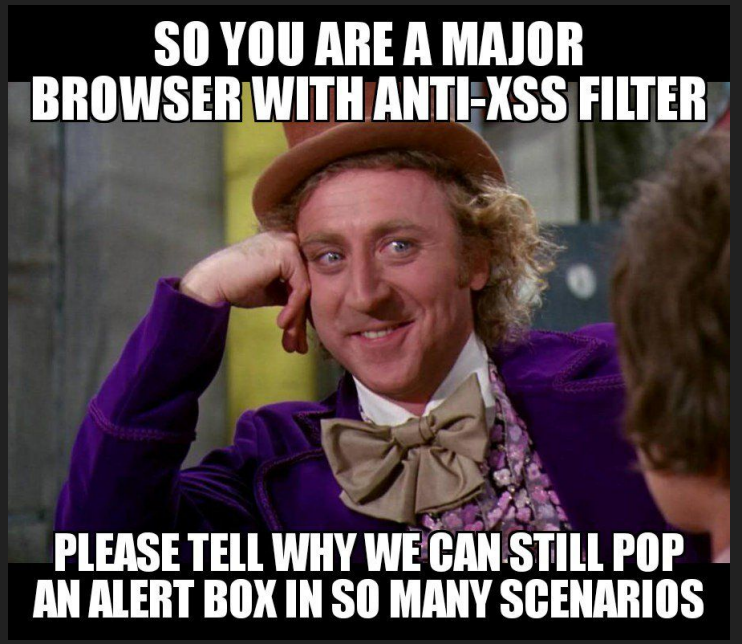
We can see that this example required using the decodeURIComponent method. This is because modern browsers encode special characters in the URL and the attack would not function without these special characters being decoded.

The content of the variable source is added to the div element using innerHTML. This is the problematic element in the code, because its assignment to innerHTML causes the value included with it to be interpreted as HTML. If the value contains JavaScript, this is executed as well.

XSS: Cross Site Scripting

Types of XSS:

- **Reflected XSS:** The difference between stored XSS and reflected XSS is that the latter concerns the user input that isn't stored in the database yet sent to another page (reflected input).



XSS: Cross Site Scripting

Example of reflected XSS:

Suppose that a developer created a web application containing this form.

After submitting it, the application redirects you to a page which prints: **Hi $firstname**
By writing the payload **"><script>alert(1);</script>** we can make the application alert 1 in the web browser when the form is submitted.

Kind of a margin

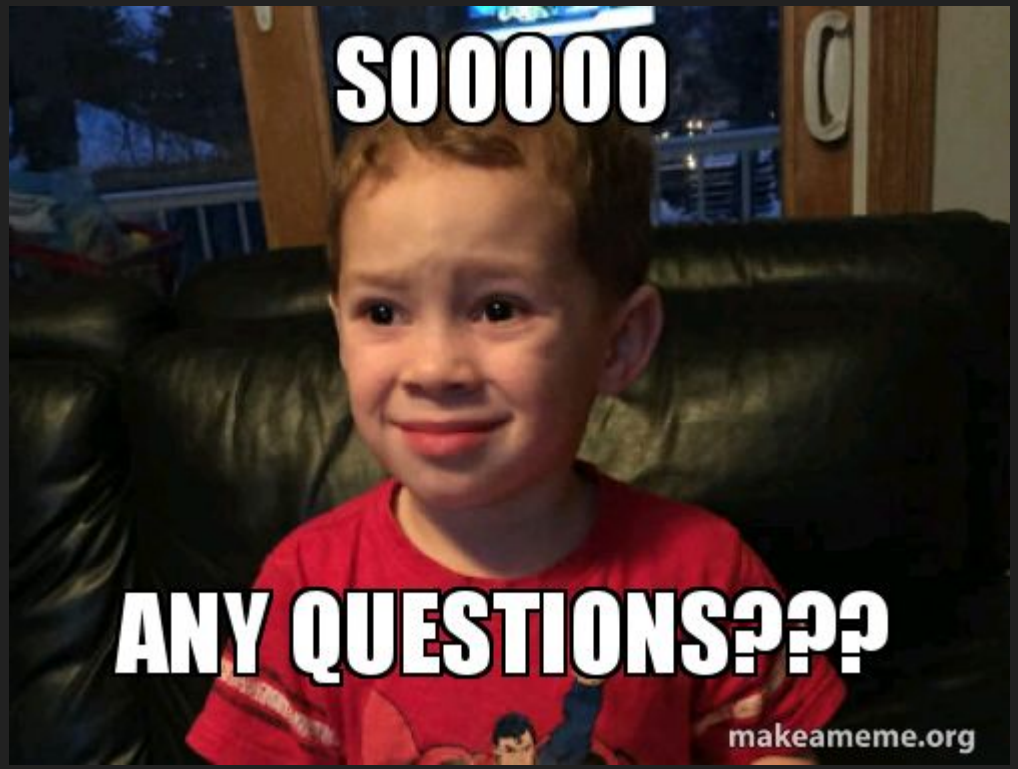
Since we didn't include all web vulnerabilities, and even for the ones mentioned, we didn't state all the details and payloads, here's a github repo which includes probably all vulnerabilities with their payloads. https://github.com/swisskyrepo/PayloadsAllTheThings

Web Exploitation is a very large subdomain of cybersecurity, and for each vulnerability within it, there are tons of payloads depending on the situation and the way the web application has been developed.

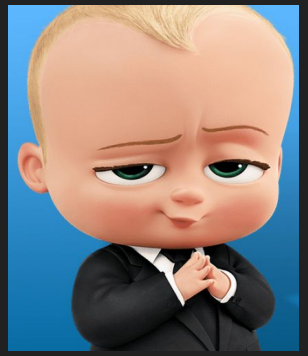
Therefore, the main key here is research, knowledge accumulation and mainly thinking outside of the box…

shutdown

tfi dak lmch9of



ls -al .Contact_us



OUSSAMA RAHALI

Facebook : /oussama.rahali.925

OMAR AOUAJ

Facebook : /omar.aouaj.77