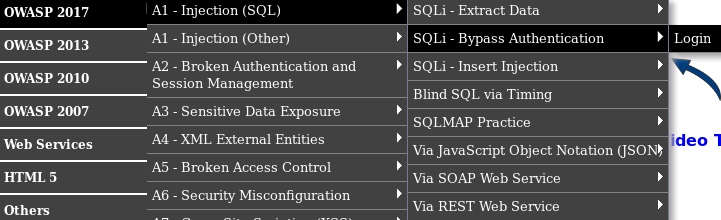
 OWASP Top Labs

Based on Mutillidae II Web App on Firefox Bowser and Kali Linux OS

A1 – Injection (SQL)

Opening the Login page, by navigating to the OWASP 2017 -> A1 - Injection (SQL) -> SQLi Bypass Authentication -> Login menu item.



Often, login SQL queries look something like this: **SELECT \* from user where user='<username>' and password='<password>'**

where <username> and <password> are the values supplied by the user.

Using the values in the instruction, the SQL became: **SELECT \* from user where user='admin' and password='whatever' or 1='1';**

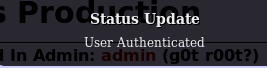
In this query, if the password is incorrect, it will use the OR 1='1' clause, which always evaluates to true. Thus SQL query will successfully return a row, even though the password is wrong. So our input changed the structure of the query to succeed even when the password is incorrect.

We will bypass authentication by abusing both the username and password fields. We will start with

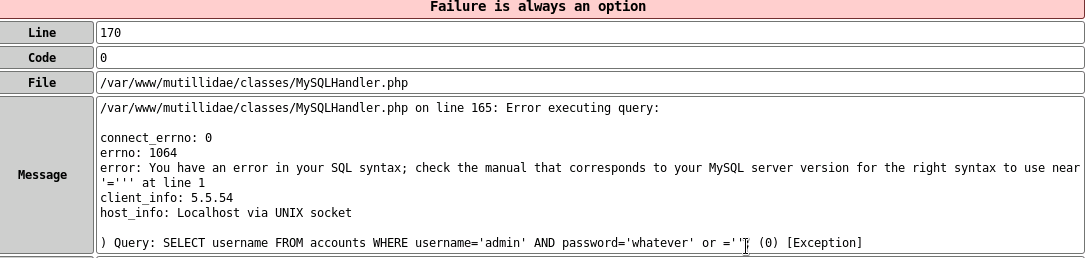
the password field adding **whatever' or 1='1**



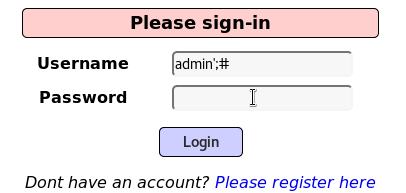
Successfully authenticated



This type of error message appear when I fail an injection

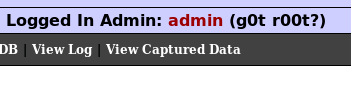


Now instead of the password field, I target the user table with the Username field only



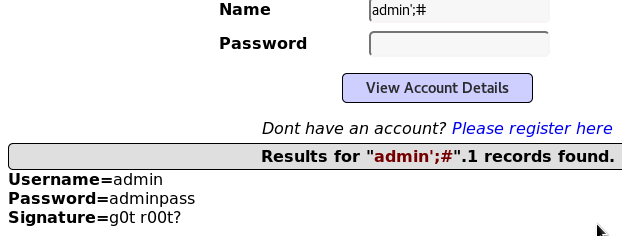
In this case, we changed the SQL query to this: SELECT \* from user where username='admin';#' and password=''; The important piece of information here is that the # character is a comment for this particular version of SQL. All SQL databases have similar symbols. Some are -- . But in any case, what happened is that we simply terminated the SQL statement after the username check and the comment told the database to ignore everything else.

Result :



Now I Open the **User Info page**, by navigating to the OWASP 2017 -> A1 - Injection (SQL) -> SQLi Extract Data -> User Info (SQL) menu item.

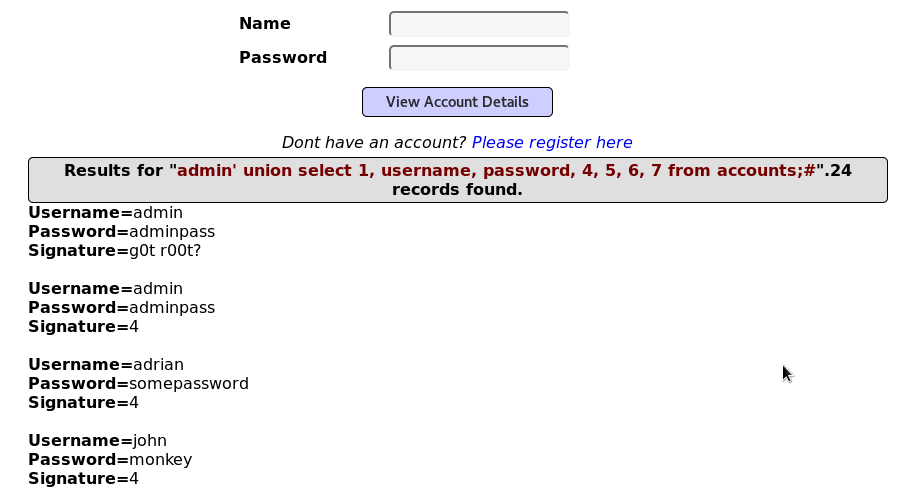




But if we put those lines on the name field **admin' or 1=1;#**



the SQL query becomes: **SELECT \* from accounts where username='admin' or 1=1;#'** **and password='**'; Since the where condition is always true, it returns all rows in the table.



**admin' union select 1, username, password, 4, 5, 6, 7 from accounts;#**

This injection makes use of the union select operation in SQL. That allows me to run a second SQL query and append it to the results of the query. Fortunately this application tells us that the number of columns is wrong. So the way to develop this sort of exploit is to keep increasing the number of constant values (that is the 1, 4, 5, 6, and 7) that we select until the attack works. Then we know that the number of columns is correct.

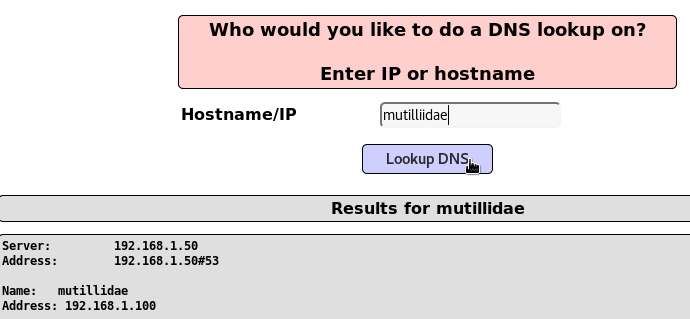
It would be natural to try selecting username, password, 3, 4, 5, 6, 7. That tells us that username should be in the second position, and that password should be in the third position.

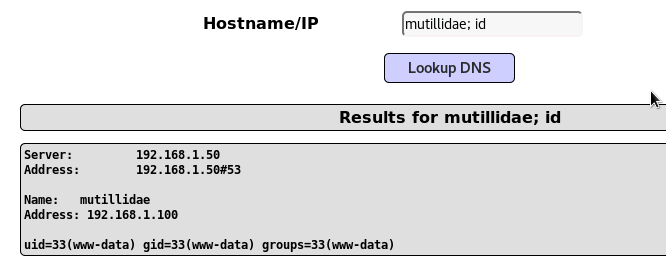
One nice thing for attackers in a union select SQL injection is that I can select from any table that the web application user has access to. If permissions are not set properly, the attacker can even dump the database system credentials.

**Command Injection**

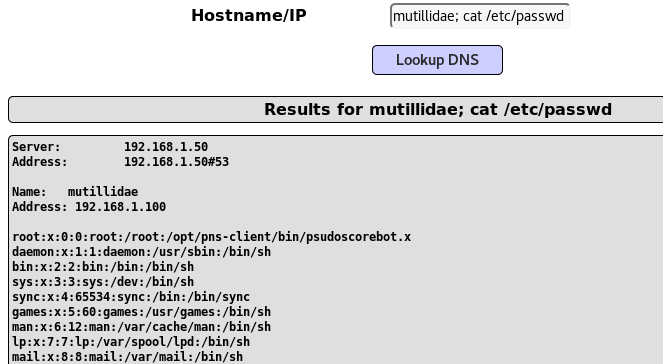
If the user data is not properly filtered, the command sent to the server operating system can be manipulated,

Open the DNS Tool page, by navigating to the OWASP 2017 -> A1 - Injection (Other) -> Command Injection -> DNS Lookup menu item.





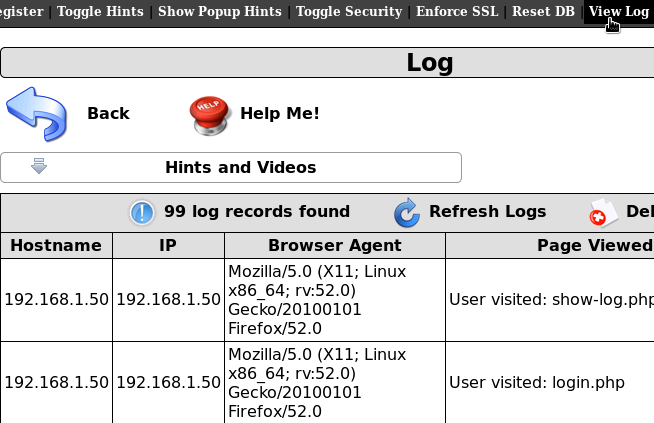
The command lines lead to even worse data leak

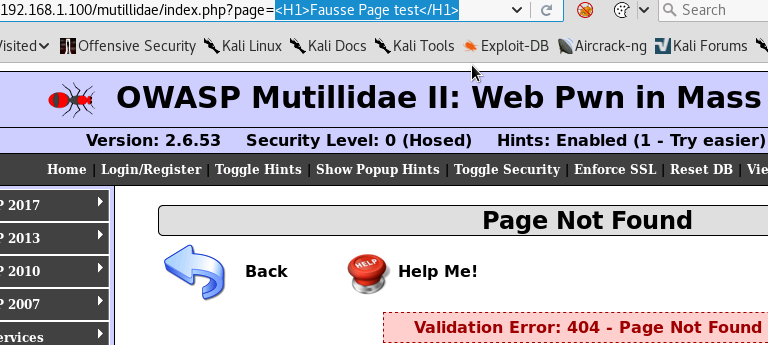


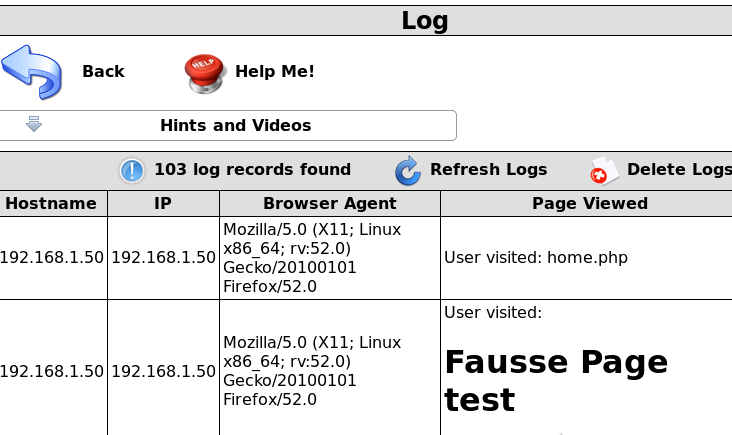
Html injection

injection of HTML content into a web page, changing the look of the page and possibly even the location that the browser goes to.

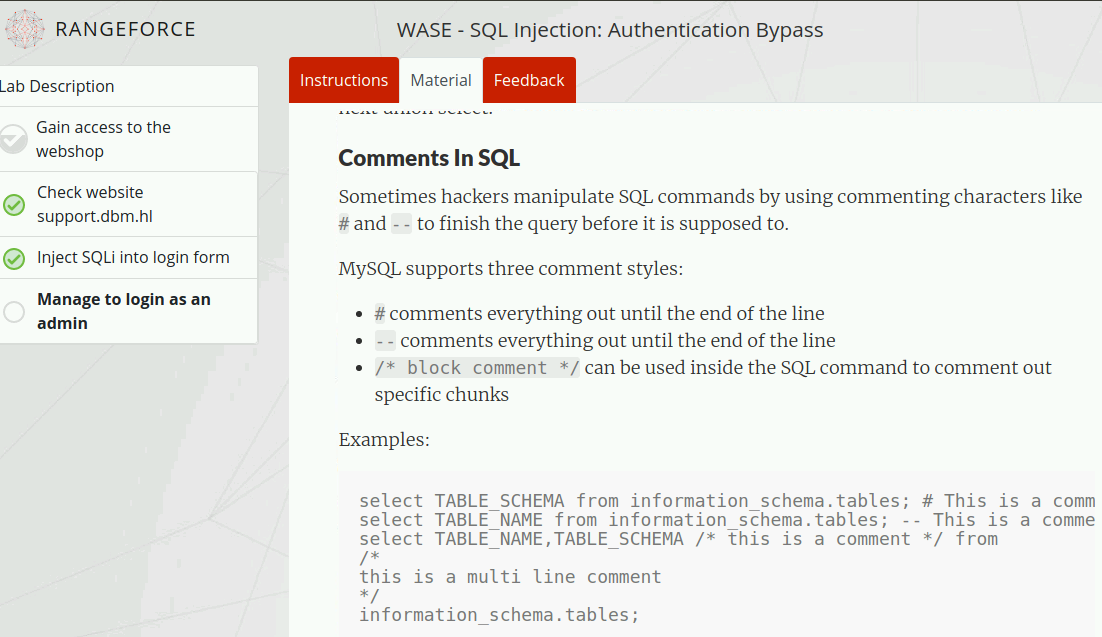
An attacker can change the look and behavior for anyone else on the application who views the injected pages.

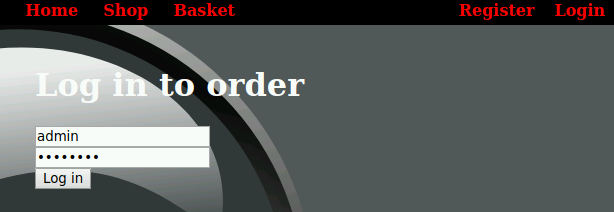






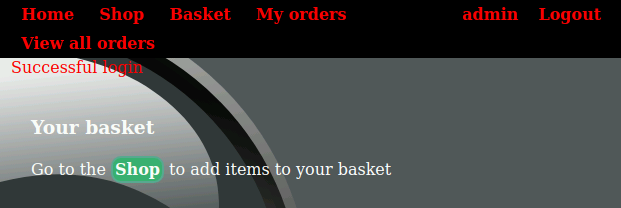
**Additional Info**

****

****

Trying to bypass the authentication in the password field:

**randompassword' or '1'='1' and id <> 1; -- #**

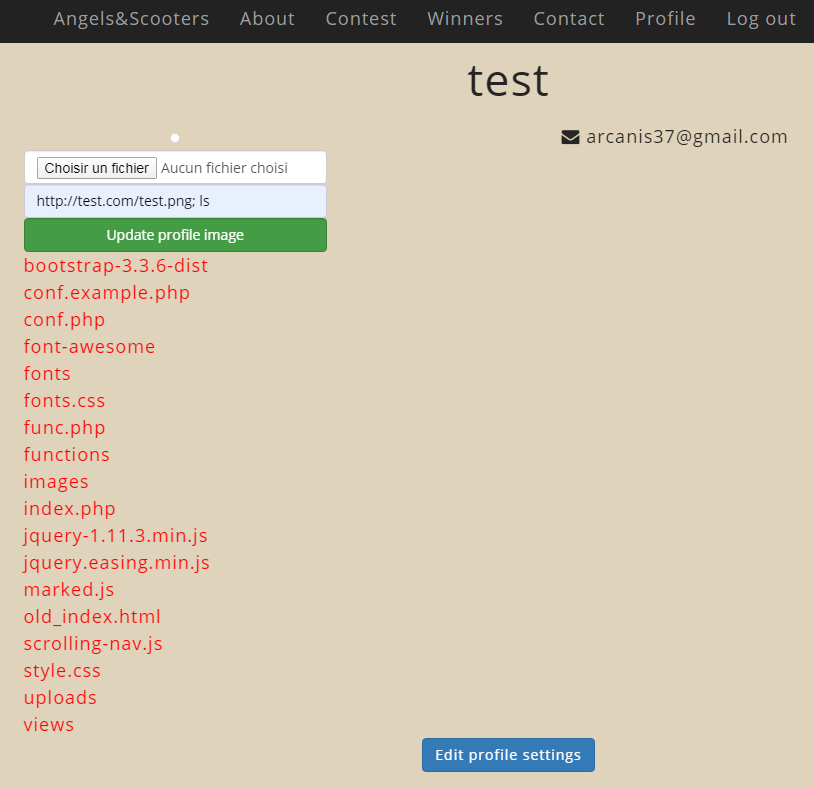


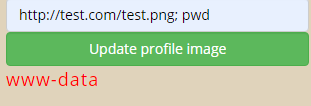
Command Injection (PHP)

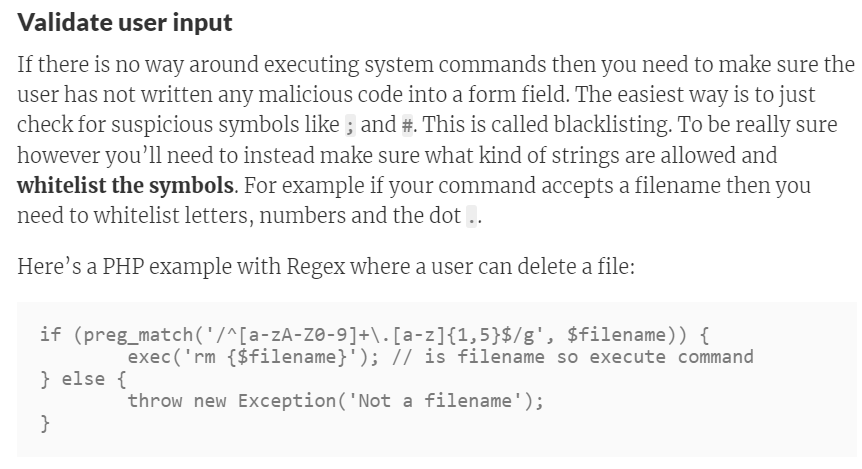
Targeting a blog page



This field require a URL to an image,



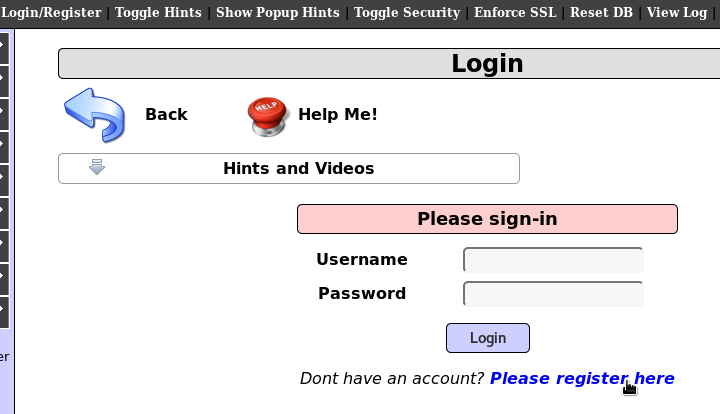


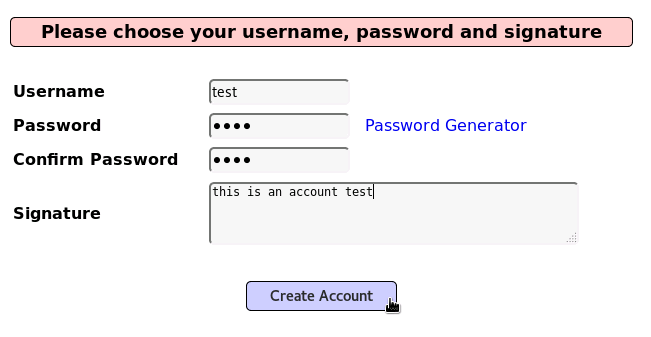


# A2 - Broken Authentication

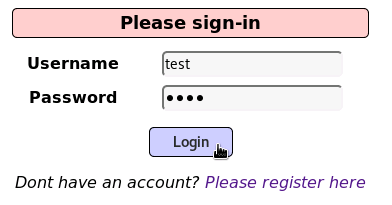
In this application, too much information is stored in the cookies, which are vulnerable to modification by the user. The first thing we will do is create a normal user, and then we will try to elevate our permissions.

Back in the vulnerable web application, I register an account





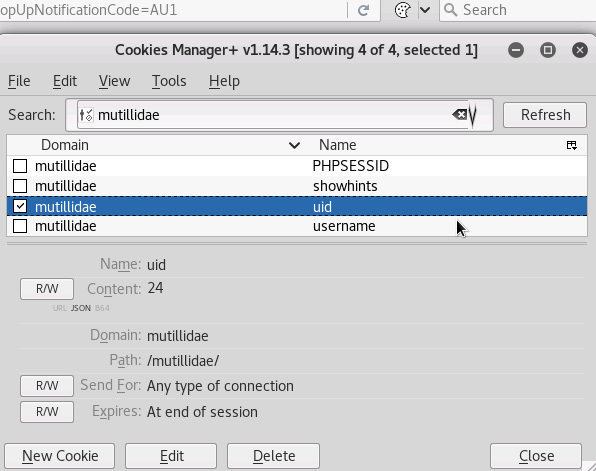
And now I log in the freshly created account



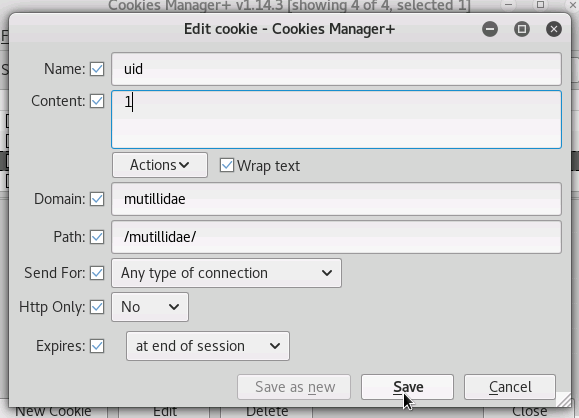
Instead of using a session token to identify the session, which would indicate the user that is actually logged in, this application stores the user directly in cookies that are stored in the browser. The user can edit these cookies and change them to any value.

As we are now logged in as a limited privilege user we will elevate our privileges.

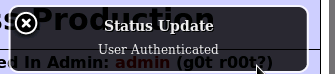
Using a browser plugin such as **Cookie Manager** it is possible to modify cookies



Let’s edit the account ID cookie. The current value of the cookie (24) is the userid of the test user that was created earlier. Let's change it to another user and see what happens. Our user id is fairly small, so there is a good chance that the admin user has a userid of 0 or 1.



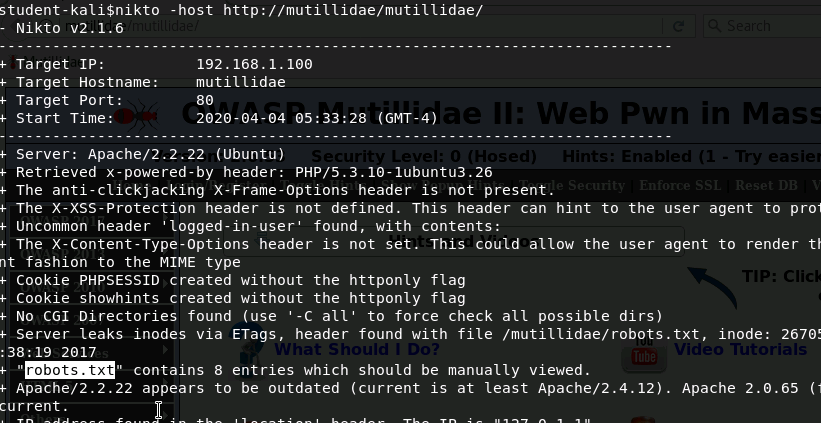
Once I refresh the page, the popup informs us that we are now admin

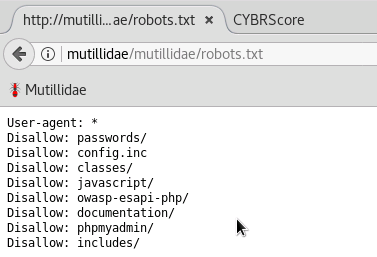


# A3 - Sensitive Data Exposure

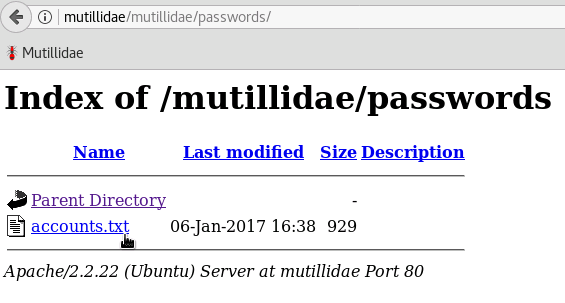
Sensitive Data Exposure is a general term that simply means exposure of data that should otherwise not be viewed by a regular user. This can include pages accessible to the general public that should be removed or hidden, backups, development files, verbose error messages, and many other types of sensitive data…

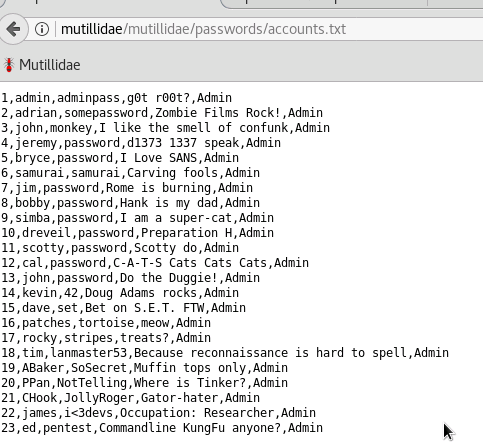
In this case we only use Nikto but Dirb, Dirbuster, Gobuster could also be used to find out even more sensitive data. The **robots.txt** file is a classic example of a security by obscurity failure.

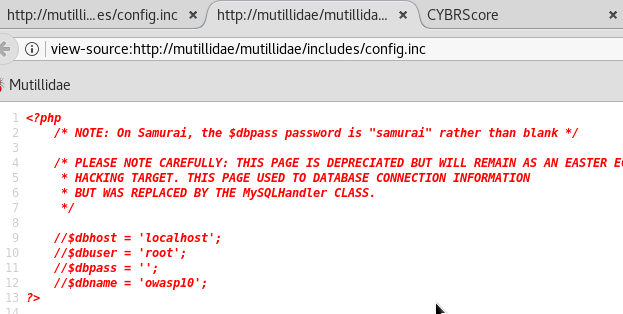




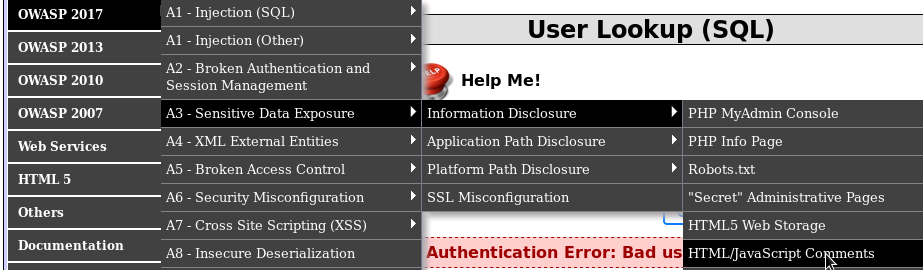
The **passwords** directory seems suspicious, almost like a honeypot trap. Still, let’s dive into it.



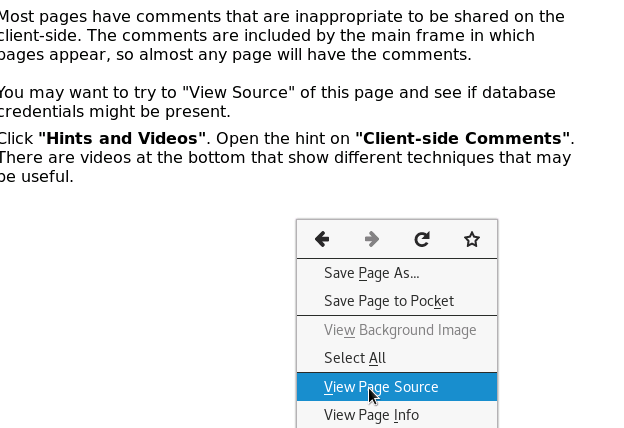




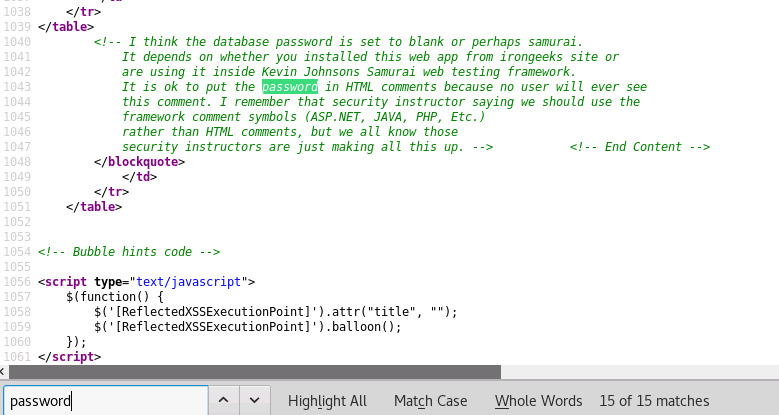
Now let’s navigate to the OWAP 2017 -> A3 - Sensitive Data Exposure -> Information Disclosure-> HTML/Javascript Comments menu item.



We check the page source to gather some sensitive informations

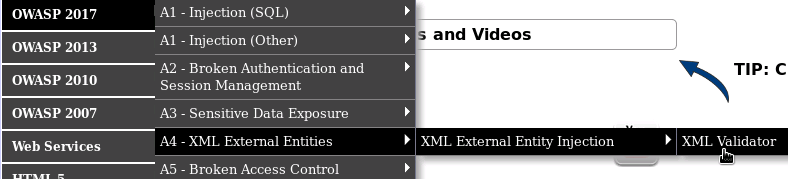


If I look for **password** keyword, some comments disclose the password to the database…



# A4 - XML External Entities

XML documents are allowed to declare entities, which enable the developer to break the document into parts, making it more modular. These entities can be defined in local or remote files, depending on how the system is configured. If a user is allowed to influence the declaration of these entities, that user might be able to access system resources or files that are otherwise prohibited.

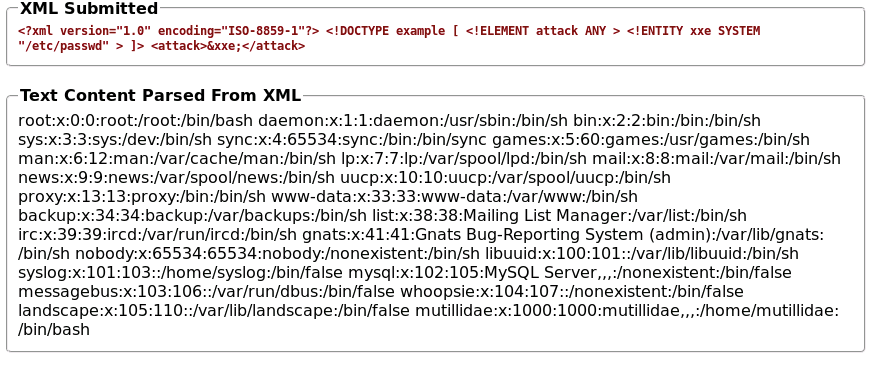


The XML parser is very picky and case sensitive.



If I add those lines I might get the content of the **/etc/passwd** file

**<?xml version="1.0" encoding="ISO-8859-1"?>**  
**<!DOCTYPE example [**  
**<!ELEMENT attack ANY >**  
**<!ENTITY xxe SYSTEM "/etc/passwd" >**  
**]>**  
**<attack>&xxe;</attack>**

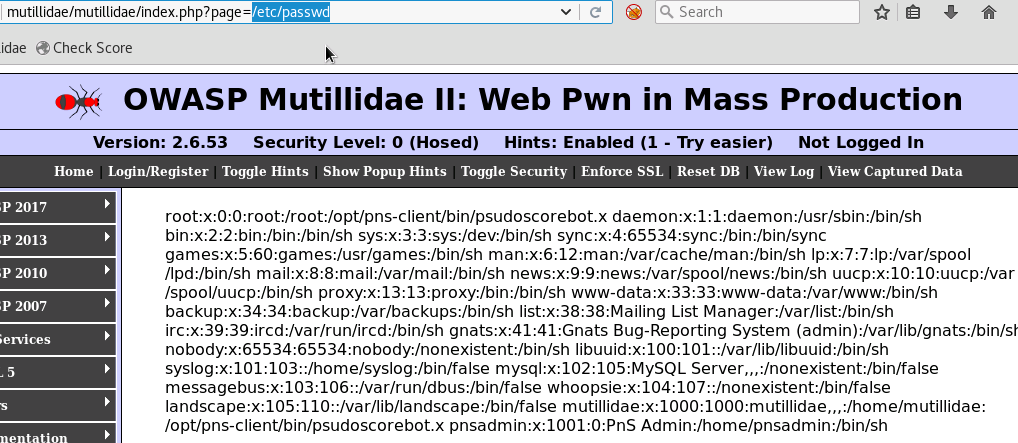


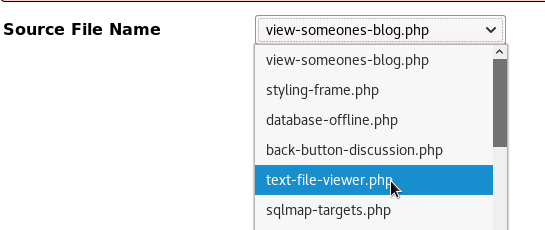
# A5 - Broken Access Control

 The URL parameter is set to **login.php**.

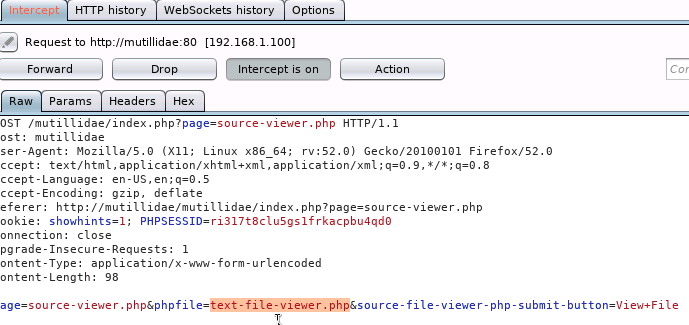
****

**index.php** file provides the constant content and the page variable in the URL determines the unique content. In PHP, this is often done by "including" the file, meaning that it grabs the content and places it in the context of the current page.

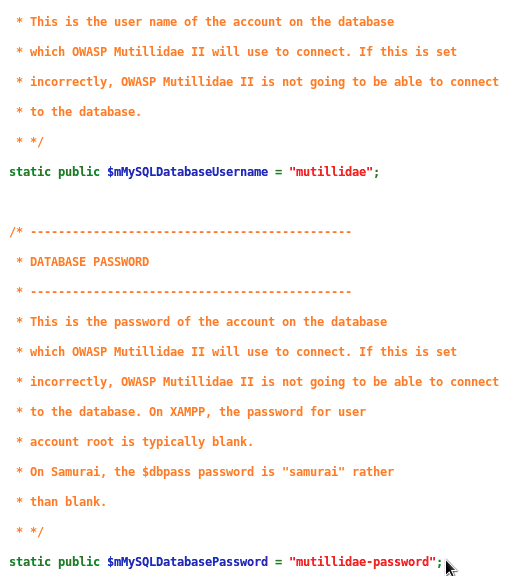




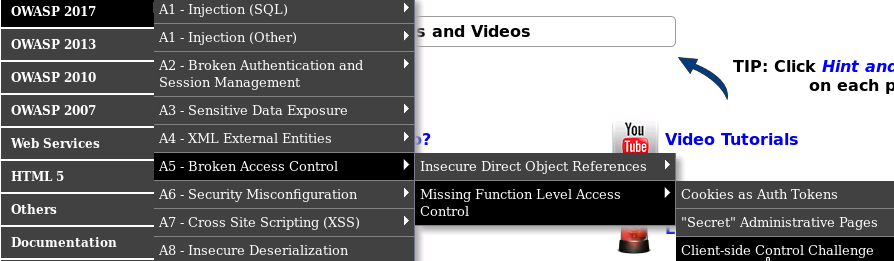
Now I launch Burp + FoxyProxy and intercept the file request



IF we change that highlighted file with **classes/MySQLHandler.php, we can access to another file.**

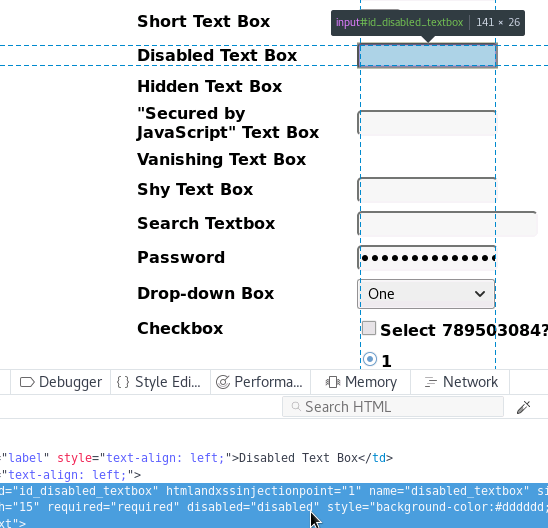


Now let’s check another client side access control

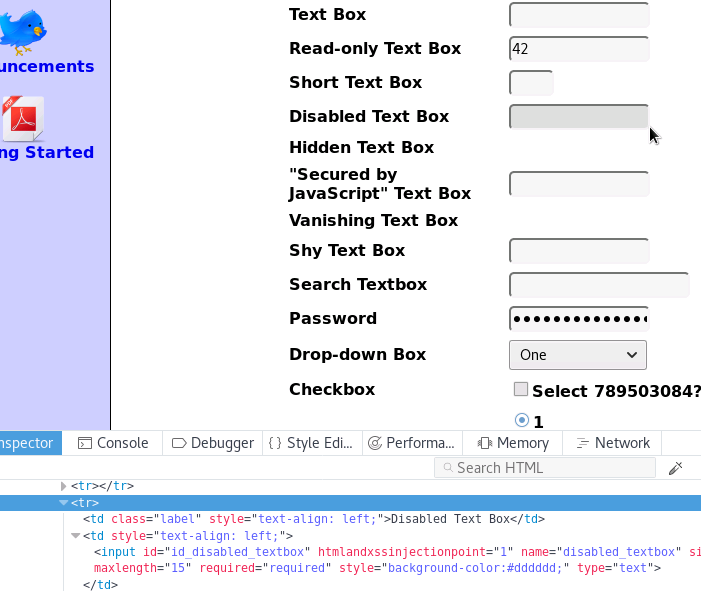


Another common failing in web applications is enforcing data integrity on the client side. This can be done through a combination of HTML attributes as well as Javascript.

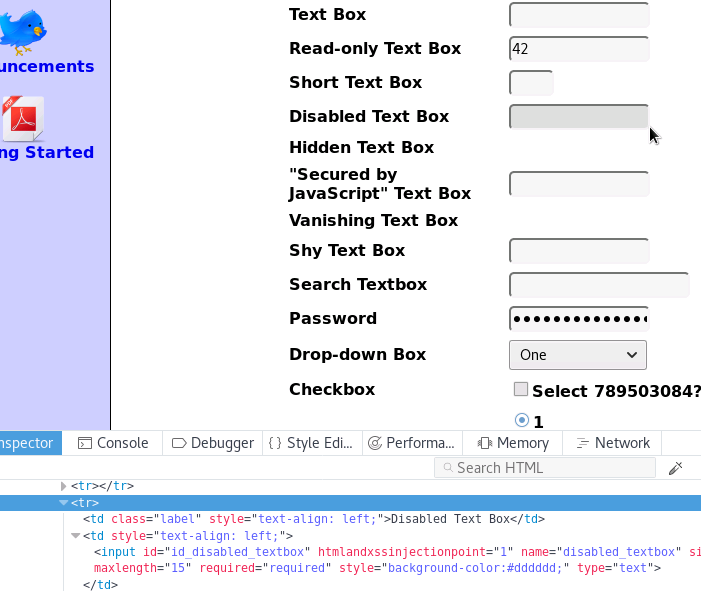
This page shows to the client some boxes that we are unable to fill up. We can modify the settings to our advantage.



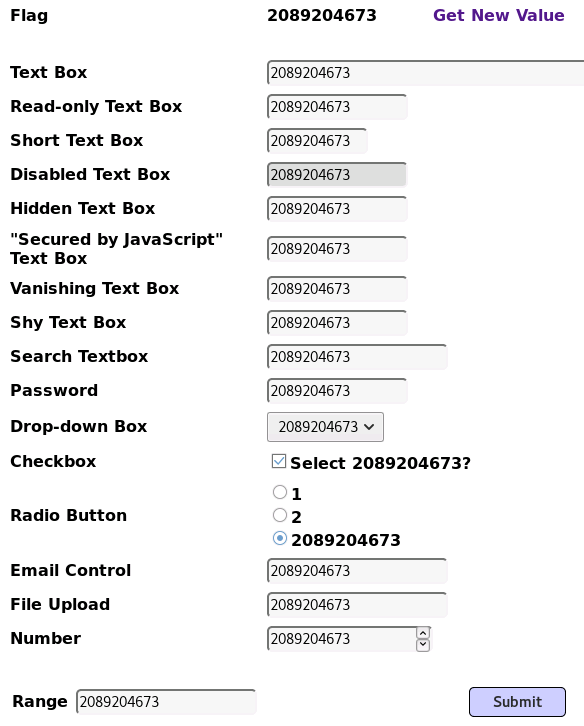
Once the line removed, it is enabled



We keep changing the html



Challenge done, all restrictions have been bypassed



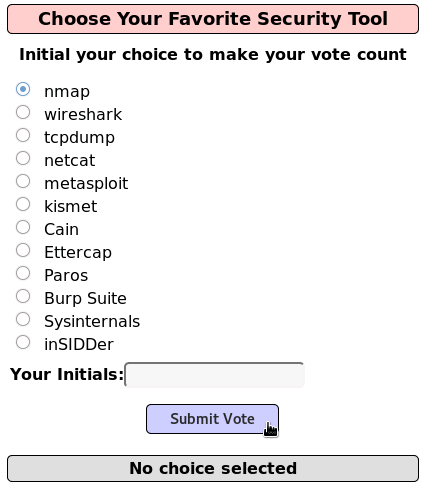
I have mostly done them manually with the browser inspection but it could have been done even more faster with Burp

# 6 Exploit Security Misconfigurations

In the URL bar, I browse to the following address:  
**http://mutillidae/mutillidae/includes/**

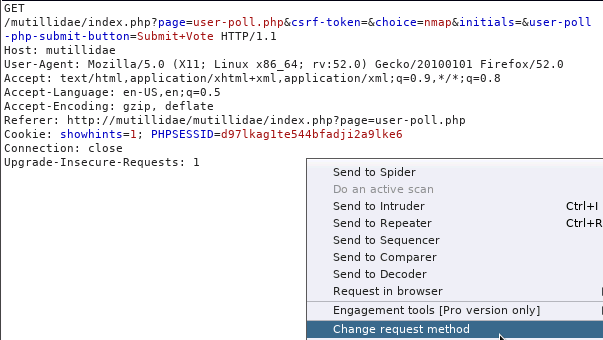
The misconfiguration here is that directory indexing is allowed. That means that if a directory does not have a default page (typically index.html or index.php), it will show the contents of the directory, as seen here. There is no 403 Forbidden message.

Now, let’s Open the User Poll page, by navigating to the OWASP 2017 -> A6 - Security Misconfiguration -> Method Tampering (GET for POST) -> Poll Question menu item.

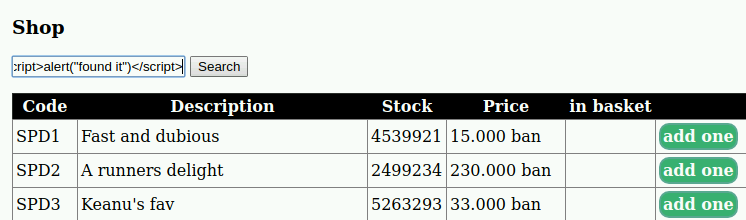


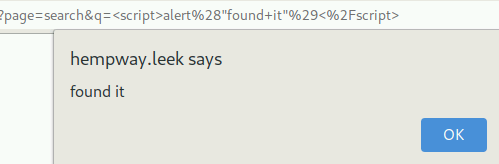
The Poll information was sent to the web application using a GET request. I can see this in the first line of the displayed request. Web servers can support a number of different request methods, but two of the main methods are GET and POST.

In this case, any request method is allowed. We will see this by changing the request method to a POST and resubmitting.



# A7 - Cross Site Scripting

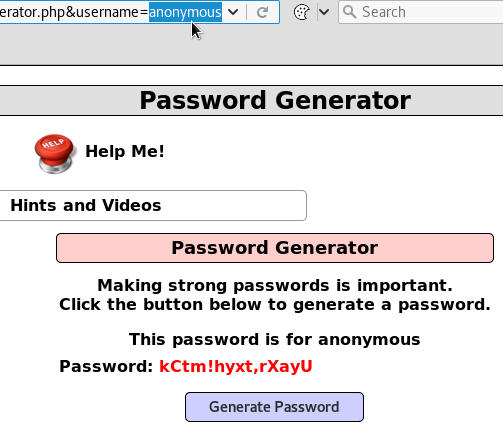


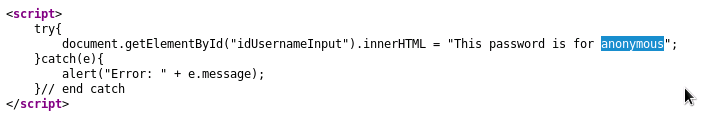


A classic example of an innocent XSS



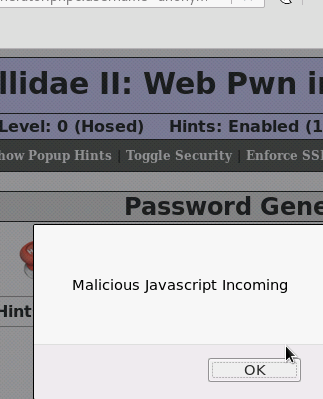
The value of username gets reflected back to us inside the web page.



Looking for the “anonymous” word in the page source.’

Let’s insert XSS Reflection





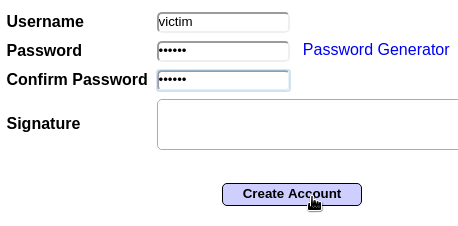
For reflected XSS, the likely attack scenario is this: a malicious link is created like the one I just did. That link is then sent in a phishing email to the victim. If they click the link, then the Javascript code runs in their browser.

Persistent XSS is a more permanent type of attack. In this case, the malicious Javascript is injected in some parameter that gets stored somewhere, often a database. This value is then used to render the page to all clients. That means that anyone who navigates to the page will be attacked.

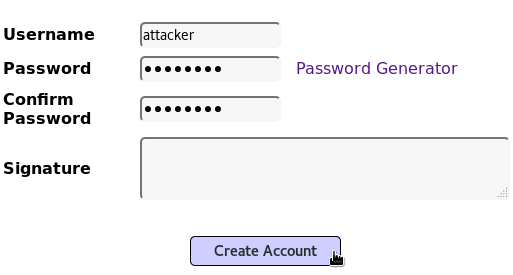
We will more fully simulate multiple parties, both the attacker and the victim. First, we will create the victim account in Chrome. Next we will create the attacker account in Firefox, and launch the attack. Then, in Chrome, the victim will view the malicious content. Then, back in Chrome, the attacker will see the results of the attack.

In the Chrome browser I create a new account

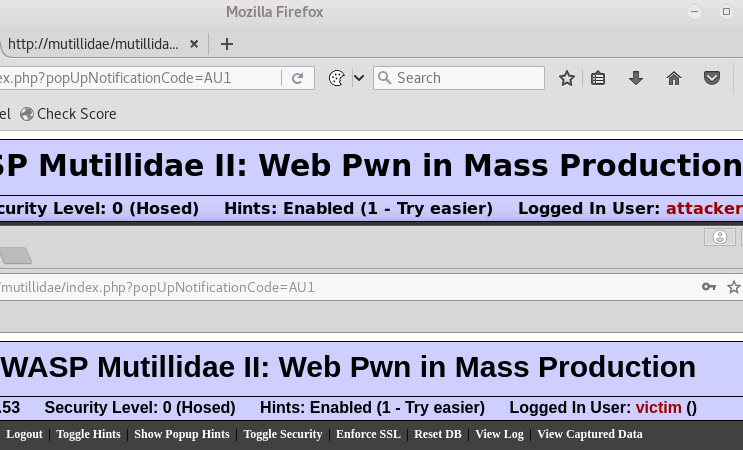


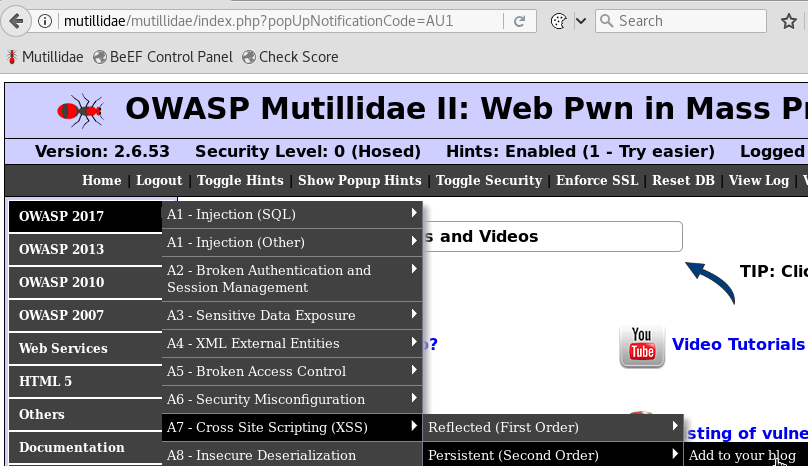


On the firefox browser



Now logging-in as the following situation



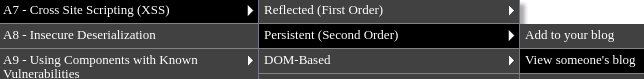




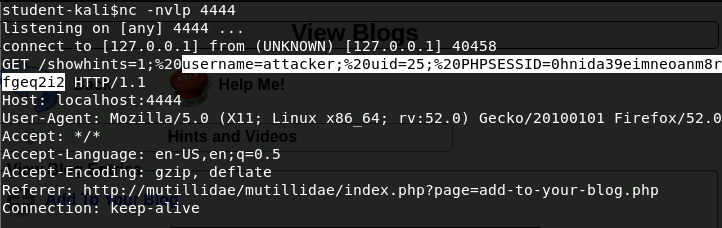
The malicious code we injected will make a request to the address we specified, namely http://localhost:4444/. In a regular attacking situation, the localhost would be replaced with the attacker's IP address or domain name.

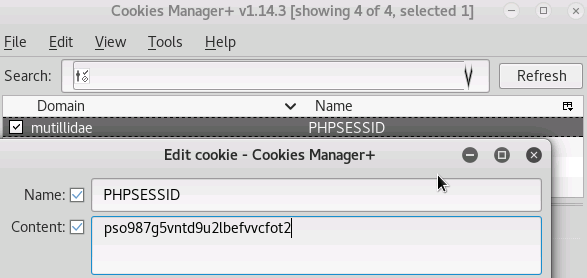


On the victim side I browse to the blog of the attacker



And get the session information in the attacker side as well

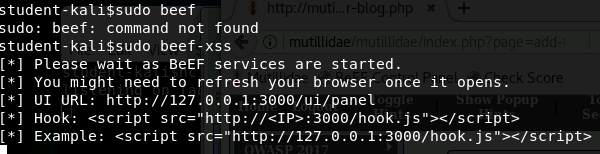




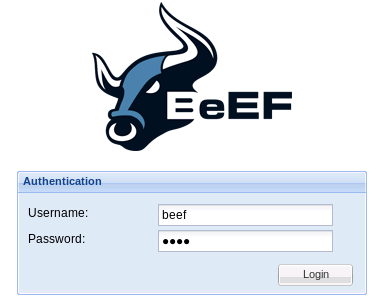
While changing the PHPSESSID with the plugin Cookies Manager we are now authenticated as the victim. We were able to steal the Victim's session and authenticate as the victim…

**Hook a Web Browser**

For the next part we will use Beef



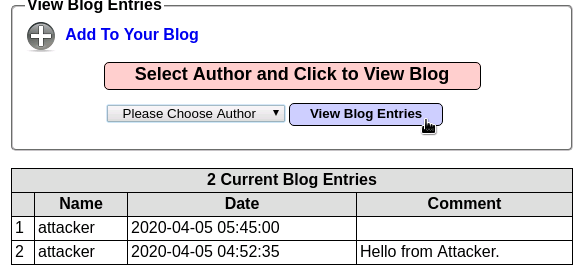
Launching beef web service



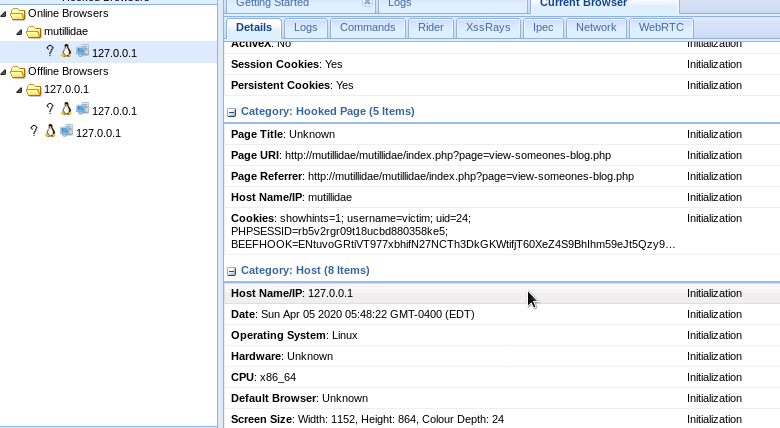
Using the sample script provided in the terminal above



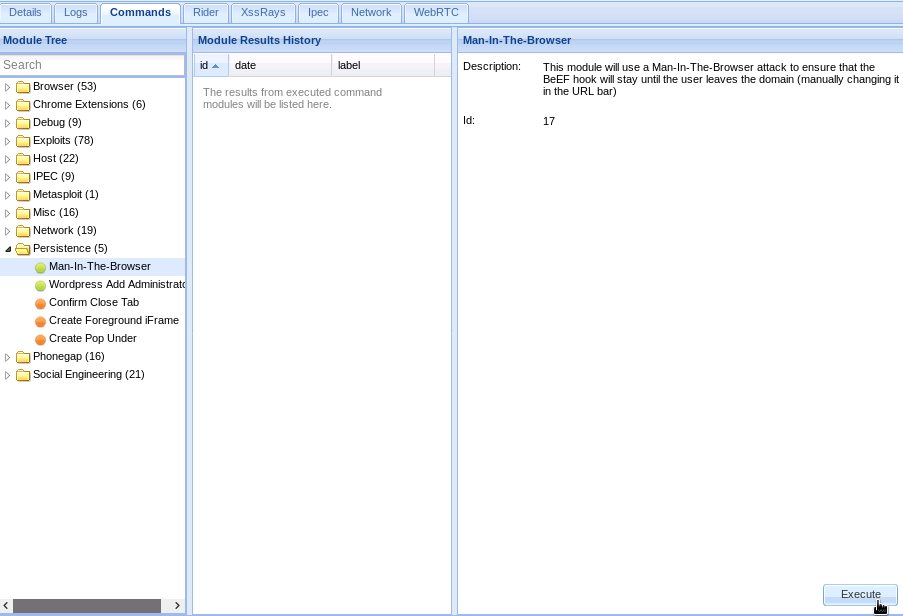
On the victim side I browse the Attacker’s post

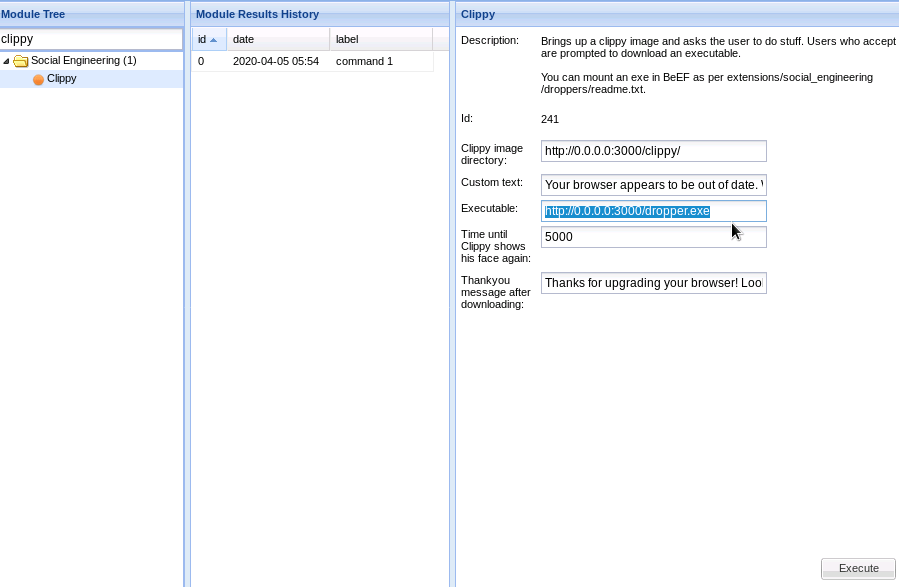


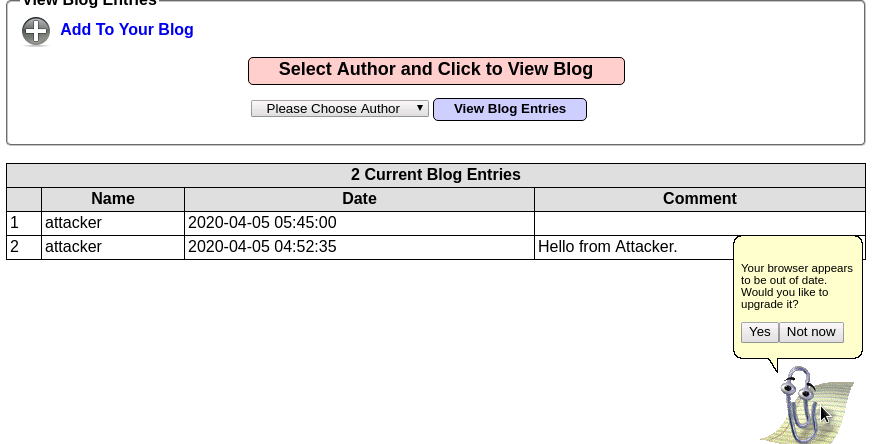
In the BeEF tab, there is an entry to an Online Browser and an Offline Broswer. The offline one is the attacker session that I closed earlier. The online one is the victim.



Wa can execute a persistence module that will make the hook remain while the victim remains on the infected domain







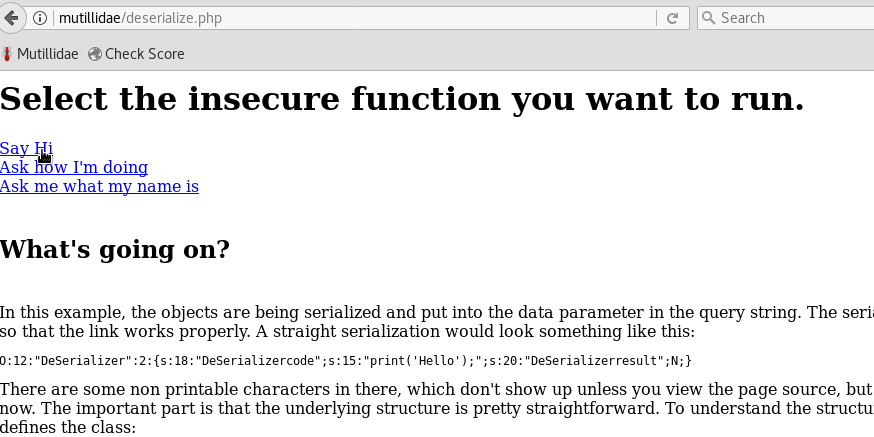
 Clippy appear in the victims browser.  I could set Clippy to download an executable malware file when the user clicks "yes".

# Exploit Insecure Deserialization

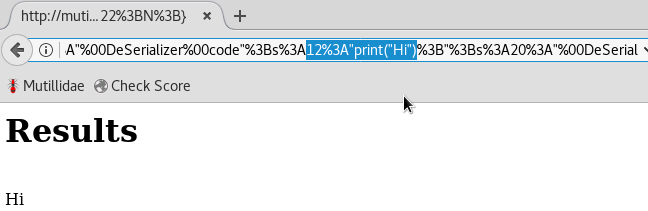
Object serialization is a technique that allows networked applications to send objects between client and server and have them recreated in that state on the other side. The process of taking an object and putting it into a form suitable for network transmission is called serialization.

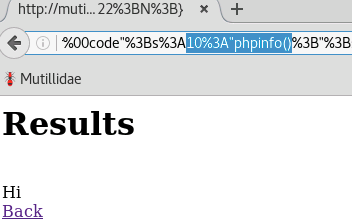
A serialized object can take many forms, depending on the underlying implementation and language, but it can be as simple as a string that allows the specification of all the member variables.

A vulnerability can arise when the end user has the ability to control at least a portion of the object in question. Depending on the level of control and the amount of information that can be changed, this can result in elevation of privileges

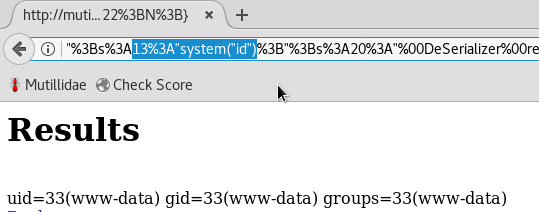


Scrolling to the right in the URL until I see the value 15 and print("Hello"). Change the 15 to 12 and Hello to Hi,



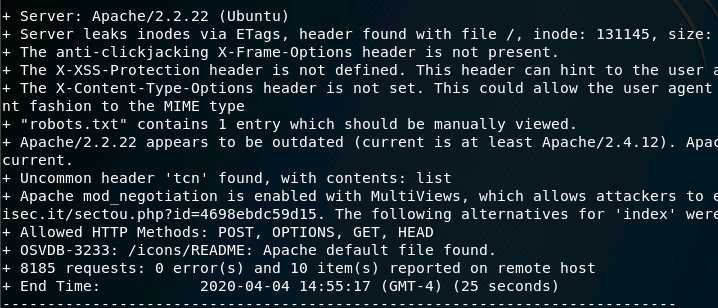


In the URL bar, now change the 10 to 13 and the **phpinfo()** to **system("id")** and press Enter. I should see the results of the id command, showing that we are running as www-data.



# 9 Detect the Insecure Component

Launching Nikto



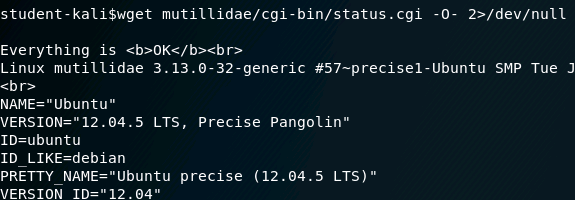
I Retrieve the robots.txt file by executing the following command in the Terminal window:

**wget mutillidae/robots.txt -O- 2>/dev/null**

the **-O-** argument indicates that wget should print the file to the screen. The **2>/dev/null** redirects all the control output to the null device, meaning it just throws it away and doesn't display it.



I Retrieve the output from the status.cgi script by executing the following command in the Terminal:  
**wget mutillidae/cgi-bin/status.cgi -O- 2>/dev/null**

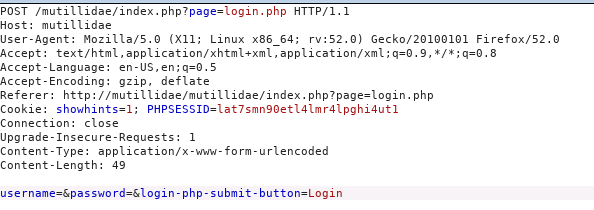


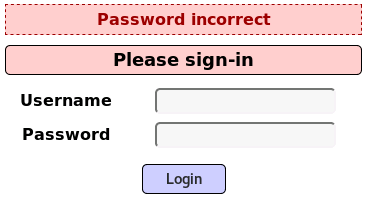
Based on this information, we can search for an exploit to target this kernel version. I won’t do it here since I have done a walkthrough with the Shellshock exploit on THB.

# 10 Insufficient Logging and Monotoring

Insufficient Logging and Monitoring refers to either logging too little information about security related events to allow for a full investigation after an incident, or sufficient logging but no one actually is monitoring the logs.

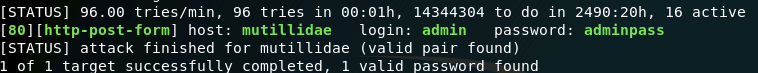
For instance let’s brute force the login page with hydra. First we need the POST data information and the response when we give false Password trying to access the admin account.



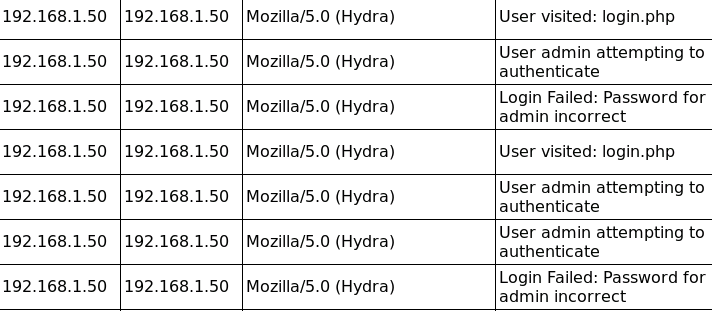


The settings are like thsese:

**hydra -f -l admin -P /usr/share/wordlists/rockyou.txt mutillidae http-post-form "/mutillidae/index.php?page=login.php:username=^USER^&password=^PASS^&login-php-submit-button=Login:Password incorrect"**



If we check the logs in the web app panel, we notice a couple of things. First, the logs are stored in the database locally. This means that a successful compromise of the system would allow for an attacker to erase or even modify the logs. So while enough data is being logged, the logs are not stored in a reliable place.



Second, there is no monitoring in place. A simple level of monitoring would be to have an automated system that watches how many login attempts are being made, and if too many are being made, the account could be locked, or the offending IP address could be blocked.