PENETRATION TEST

REDACTED_COMPANY_NAME BLOG

Table of Contents

EXECUTIVE SUMMARY

SYNOPSIS

This Pentest was conducted within recruitment process for REDACTED_COMPANY_NAME sp. z o.o. sp. k.

FINDINGS OVERVIEW

While conducting the external black box penetration test, there were several critical vulnerabilities discovered in provided webapplication. Most critical ones consist of: sensitive information disclosure, admin account takeover, which allowed further full administrative control over the application and SQL Injection.

SEVERITY SCALE

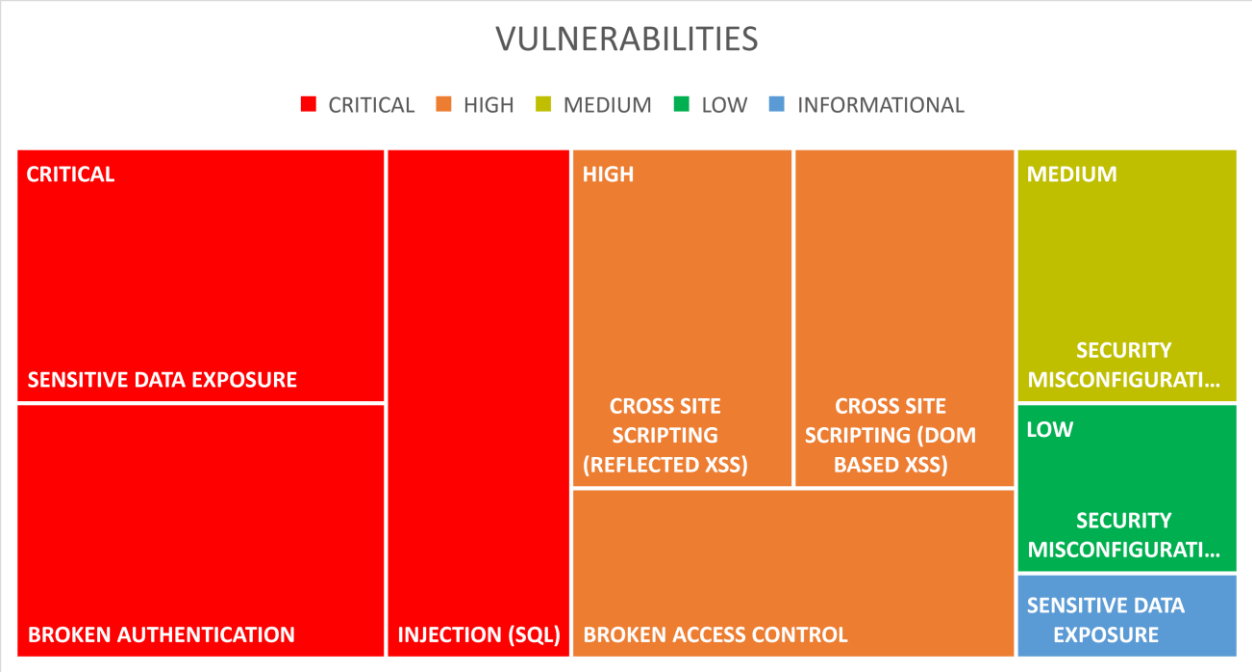
CRITICAL **Severity:** Poses immediate danger to systems, network, and/or data security and should be addressed as soon as possible. Exploitation requires little to no special knowledge of the target. Exploitation doesn't require highly advanced skill, training, or tools.

HIGH **Severity:** Poses significant danger to systems, network, and/or data security. Exploitation commonly requires some advanced knowledge, training, skill, and/or tools. Issue(s) should be addressed promptly.

MEDIUM **Severity:** Vulnerabilities should be addressed in a timely manner. Exploitation is usually more difficult to achieve and requires special knowledge or access. Exploitation may also require social engineering as well as special conditions.

LOW **Severity:** Danger of exploitation is unlikely as vulnerabilities offer little to no opportunity to compromise system, network, and/or data security. Can be handled as time permits.

INFORMATIONAL **Severity:** Meant to increase client's knowledge. Exploitation is highly unlikely.



CRITICAL Severity:

- SENSITIVE DATA EXPOSURE
- BROKEN AUTHENTICATION
- INJECTION (SQL)

HIGH Severity:

- CROSS SITE SCRIPTING (REFLECTED XSS)
- CROSS SITE SCRIPTING (DOM BASED XSS)
- BROKEN ACCESS CONTROL

MEDIUM Severity:

- SECURITY MISCONFIGURATION

LOW Severity:

- SECURITY MISCONFIGURATION

INFORMATIONAL Severity:

- SENSITIVE DATA EXPOSURE

RECOMMENDATIONS

To increase the security posture, it is recommend the following mitigations and/or remediations to be performed:

- **Implement Prepared Statements with Parameterized Queries**. Injection attacks remains the most common attacks leveraged against web applications. One of the most effective mitigation strategies for preventing SQL Injection attacks is the implementation of Prepared Statements with Parameterized Queries.
- **Implement User Input Whitelisting**. Another very useful mitigation against SQL and XSS Injection attacks is to validate the supplied user input. One should never trust that user input is safe and therefore should be checked for a set of disallowed characters.
- **Require Secure Coding Training for Developers**. Developers are on the front lines of security for any organization and should be prepared to be the first line of defense. Training in secure coding techniques and practices will help ensure that your organization's applications

are developed using the most secure code possible, thus reducing your attack-surface and lowering your overall risk.

FINAL REPORT

METHODOLOGY

Vulnerabilities are assessed according to CVSS 3.1 standard. And classified according to OWASP TOP 10 standard.

During the Penetration test, only manual audit techniques were used to ensure the best possible results with minimum bandwidth impact.

INFORMATION GATHERING

TARGET IN SCOPE: <REDACTED>

IP ADRESS: <REDACTED_IP>

TECHNOLOGY: Apache, PHP 5.1.3

ENUMERATION

During manual enumeration, some directories/files with sensitive information were disclosed. They contained data about the product itself, its environment or the related system that is not intended be disclosed by the application.

VULNERABILITY ASSESSMENT AND EXPLOITATION

1. **Vulnerability: SENSITIVE DATA EXPOSURE**
   Severity: CRITICAL
   CVSS: 9.3 CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:C/C:H/I:L/A:N

Explanation:

Sensitive Data Exposure vulnerabilities can have both a direct and indirect impact depending on the purpose of the website and, therefore, what information an attacker is able to obtain. In some cases, the act of disclosing sensitive information alone can have a high impact on the affected parties.

Mitigation:

- Ensure you have nothing sensitive exposed within robots.txt file, such as the path of an administration panel. If disallowed paths are sensitive and you want to keep it from unauthorized access, do not write them in the Robots.txt, and ensure they are correctly protected by means of authentication.
  The following block can be used to tell the crawler to index files under /web/ and ignore the rest:

```
User-Agent: *
Allow: /web/
Disallow: /
```

  Please note that when you use the instructions above, search engines will not index your website except for the specified directories..

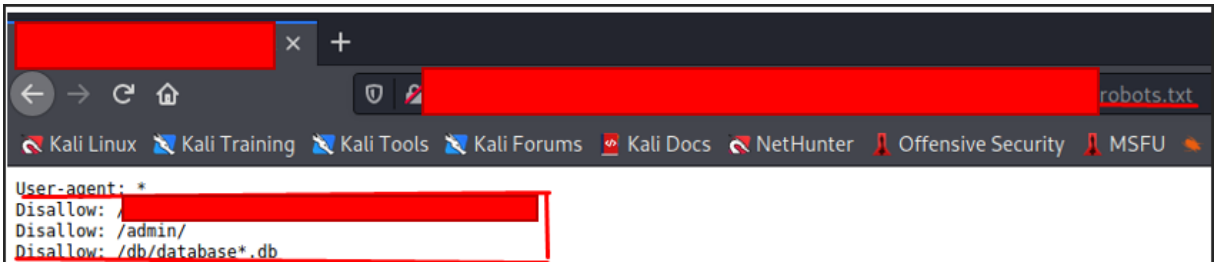
- Make sure that everyone involved in producing the website is fully aware of what information is considered sensitive. Sometimes seemingly harmless information can be much more useful

to an attacker than people realize. Highlighting these dangers can help make sure that sensitive information is handled more securely in general by your organization.

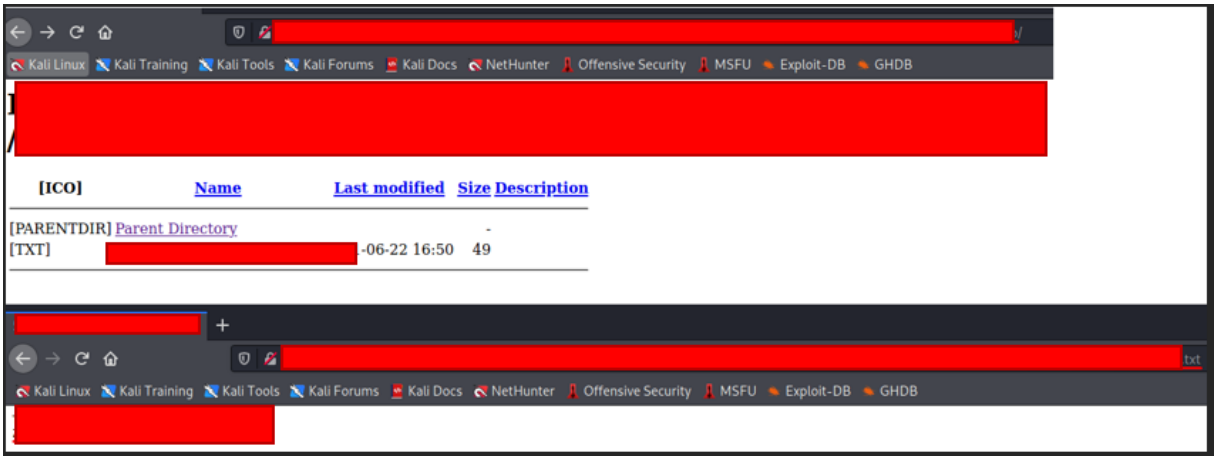
Reference:

Reproduce:

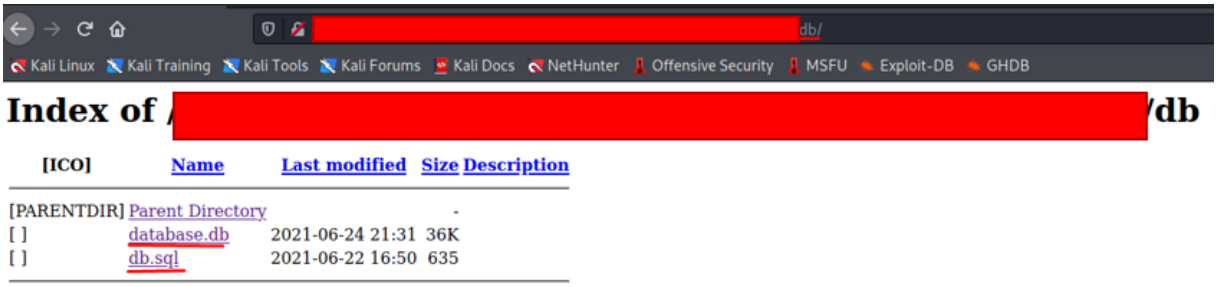
1. Robots.txt file was discovered. It contained **3 (three) sensitive locations** which were accessed manually.

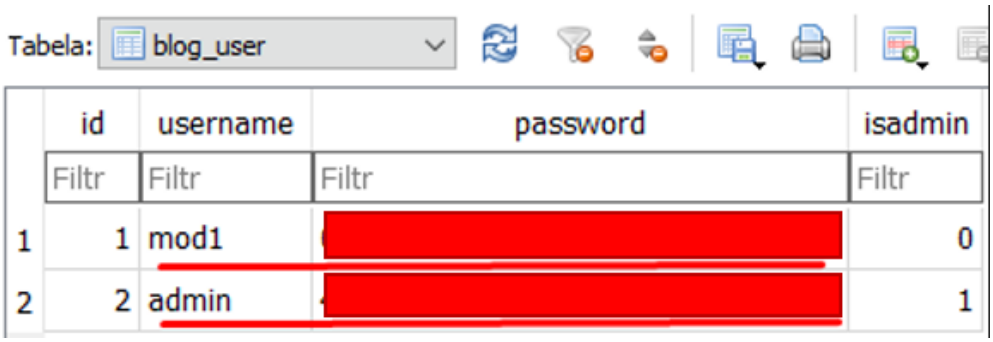


1) **Directory: /<REDACTED>/** Consisted of Company's Super Secret Documentation.

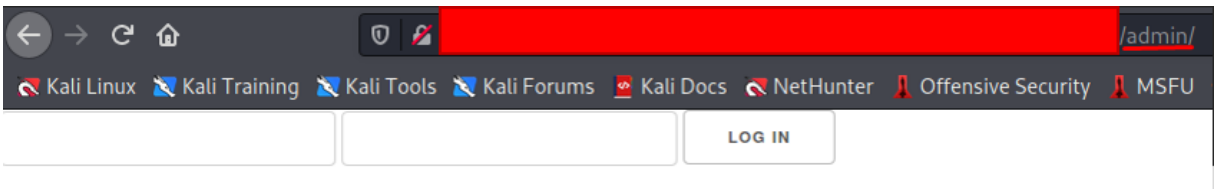


2) **Directory: /db/** Consisted of Web Application's database which was possible to download for unathorized users.





3) **Directory: /admin/** Consisted of compromised admin/user login panel.



2. Vulnerability: INJECTION (Boolean-Based Blind SQLi)
Severity: CRITICAL
CVSS: 10.0 CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:L

Explanation:

SQL injection vulnerability allows an attacker to inject malicious input into an SQL statement.

Mitigation:

- Keep export.php file from unauthorized access.
- Use prepared statements. SQL prepared statements store a prepared statement, feed it with the data, and it assembles and sanitizes it for you upon execution. They therefore ensure that user input cannot interfere with the structure of the intended SQL query. Prepared statement Example:

```
$stmt = $dbh->prepare("SELECT * FROM users WHERE USERNAME = ? AND PASSWORD = ?");

$stmt->execute(array($username, $password));
```

- Match against common SQL query keywords in URLs and block them:

```
RewriteCond %{QUERY_STRING} [^a-z](declare¦char¦set¦cast¦convert¦delete¦drop¦exec¦insert¦meta¦script¦selec
t¦truncate¦update)[^a-z] [NC]

RewriteRule (.*) - [F]
```

- Maintain and stick to the Least Privilege rule, so that any user or program should have only the absolute very least amount of privileges necessary to complete its tasks.

Reference:

Reproduce:

After capturing response for particular blog post, for example:

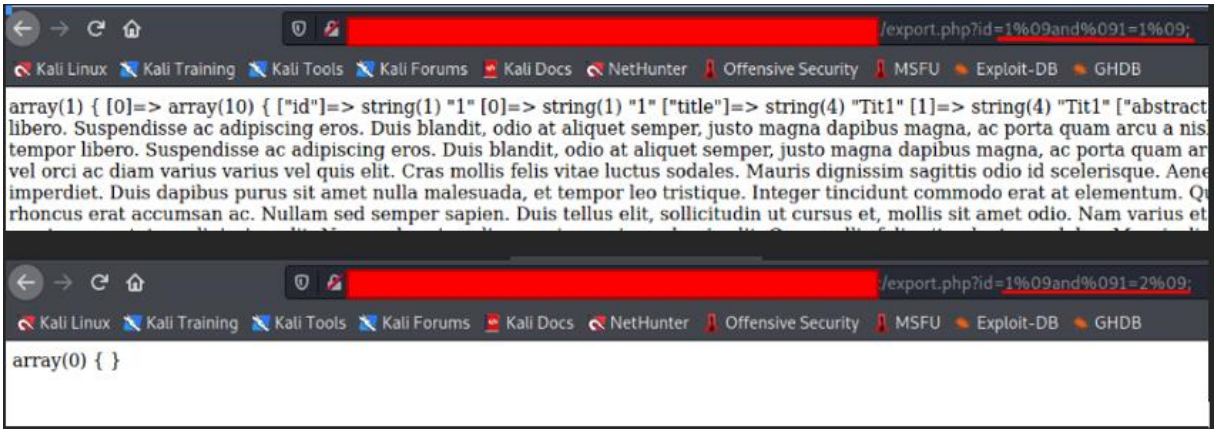
I came across export.php file which turned out to be an SQL Query entry point:

```
            </label>
            <textarea class="u-full-width" name="comment" rows="5" cols="100" id="comment" placeholde
            </textarea>
            <br>
            <input type="hidden" name="note_id" value="15">
            <input type="hidden" name="accepted" value="0">
          </div>
          <input class="button-primary" type="submit" value="Submit">
        </form>
      </div>
      <!-- TODO access only from /admin <a href="export.php?id=15">export</a> -->
    </div>
    <div class="three columns">
      ++++++++[&gt;+&gt;++&gt;+++&gt;+ +++&gt;+++++&gt;++++++&gt;+++ ++++&gt;++++++++&gt;++++++ +++&g
      t;&gt;&gt;&gt;&gt; &gt;&gt;&gt;+.-&lt;&lt;&lt;&lt;&lt;&lt;&lt;&lt;&lt;&lt;&lt;&lt;&lt;&lt;&gt;&gt; &gt;
      t;&gt;&gt; &gt;&gt;&gt;----.++++&lt;&lt;&lt;&lt;&lt;&lt;&lt;&lt;&lt; &lt;&lt;&lt;&lt;&lt;&gt;&gt;&g
      &gt;. &lt;&lt;&lt;&lt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;--- .+++&lt;&lt;&lt;&
      &gt;&gt;&gt;&gt;&gt;&gt; &gt;&gt;+++.---&lt;&lt;&lt;&lt;&lt;&lt;&lt;&lt;&lt;&lt; &lt;&gt;&g
    </div>
  </div>
  <div class="row">
    <div class="footer">
```

Then I have tampered with the query encoding, until I confirmed, there is a Boolean based blind SQL injection vulnerability:

Payloads:

- 1%09and%091=1%09;
- 1%09and%091=2%09;



Using logic operators I have confirmed that webapplication turns blog post content if the query is true (1=1) and empty array if the query is false (1=2).

I was able to further exploit this with query to check for the first letter of the user's password stored in database. Of course the atacker might use this technique to query all table names first.

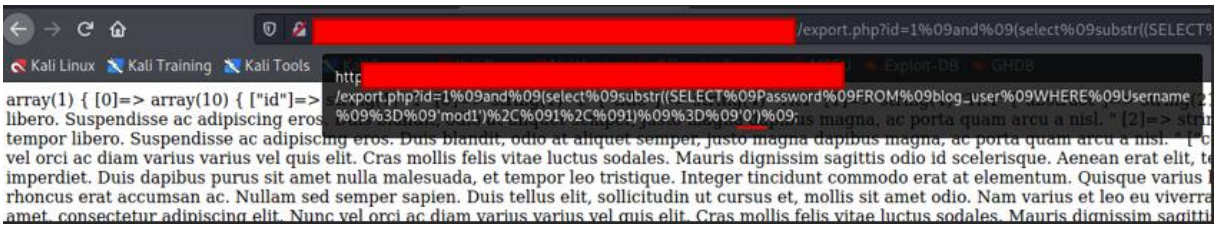
Payload:

- 1 and (select substr((SELECT Password FROM blog_user WHERE Username = 'mod1'), 1, 1) = '0') ;

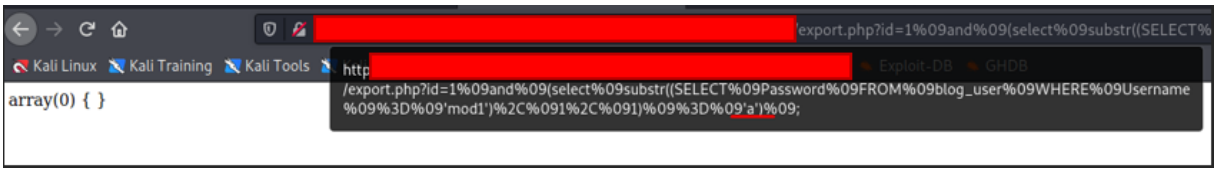
URL encoded Payload with %09 (horizontal tabs) instad of spaces:

- 1%09and%09(select%09substr((SELECT%09Password%09FROM%09blog_user%09WHERE%09Username%09%3D%09'mod1')%2C%091%2C%091)%09%3D%09'0')%09;

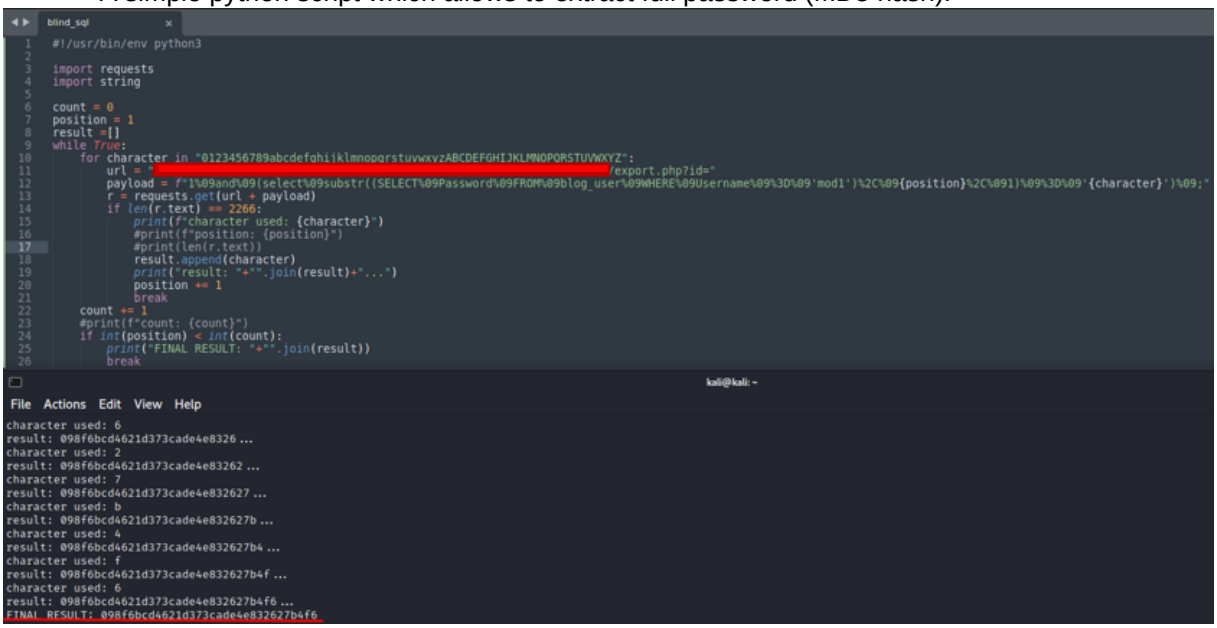
  Turns True which means first character of user's password is '0' :



Turns False which means first character of user's password is not 'a' :



PoC:
A simple python script which allows to extract full password (MD5 hash):



3. Vulnerability: BROKEN AUTHENTICATION
Severity: CRITICAL
CVSS: 10.0 CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:H

Explanation:

Authentication in web applications is used to verify user identities and authorize access to sensitive information. Security risks related to authentication and session management include password theft, session hijacking using compromised tokens, and impersonating legitimate users. Common vulnerabilities related to authentication are typically exploited via password reset functionality by tampering with tokens such as cookies and session IDs.

It was possible to craft admin session token in order to fully compromise admin account.

Mitigation:

- Use a server-side, secure, built-in session manager that generates a new random session ID with high entropy after login. Session IDs should not be in the URL, be securely stored and invalidated after logout, idle, and absolute timeouts.
- Where possible, implement multi-factor authentication to prevent automated, credential stuffing, brute force, and stolen credential re-use attacks.

Reference:

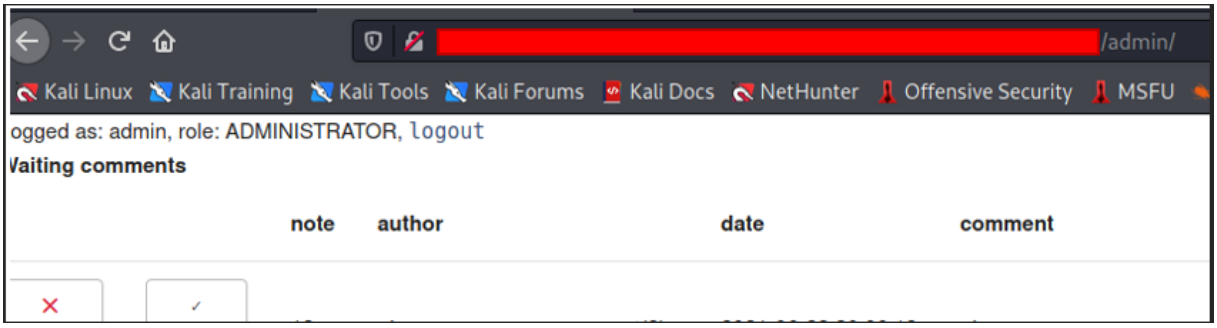
https://owasp.org/www-project-top-ten/2017/A2_2017-Broken_Authentication

Reproduce:

Intercept login request on:

Change user_id in "adminsession" cookie (base64 encoded) to user_id=2 and send the request:

You are logged as admin:



4. **Vulnerability: CROSS SITE SCRIPTING**
   Severity: HIGH

   a. (REFLECTED XSS)

After tampering with "id" parameter on main page it is possible to escape the comment and fire XSS:

Payload: --><script>alert(1)</script>

a. (DOM BASED XSS)

After tampering with endpoint:

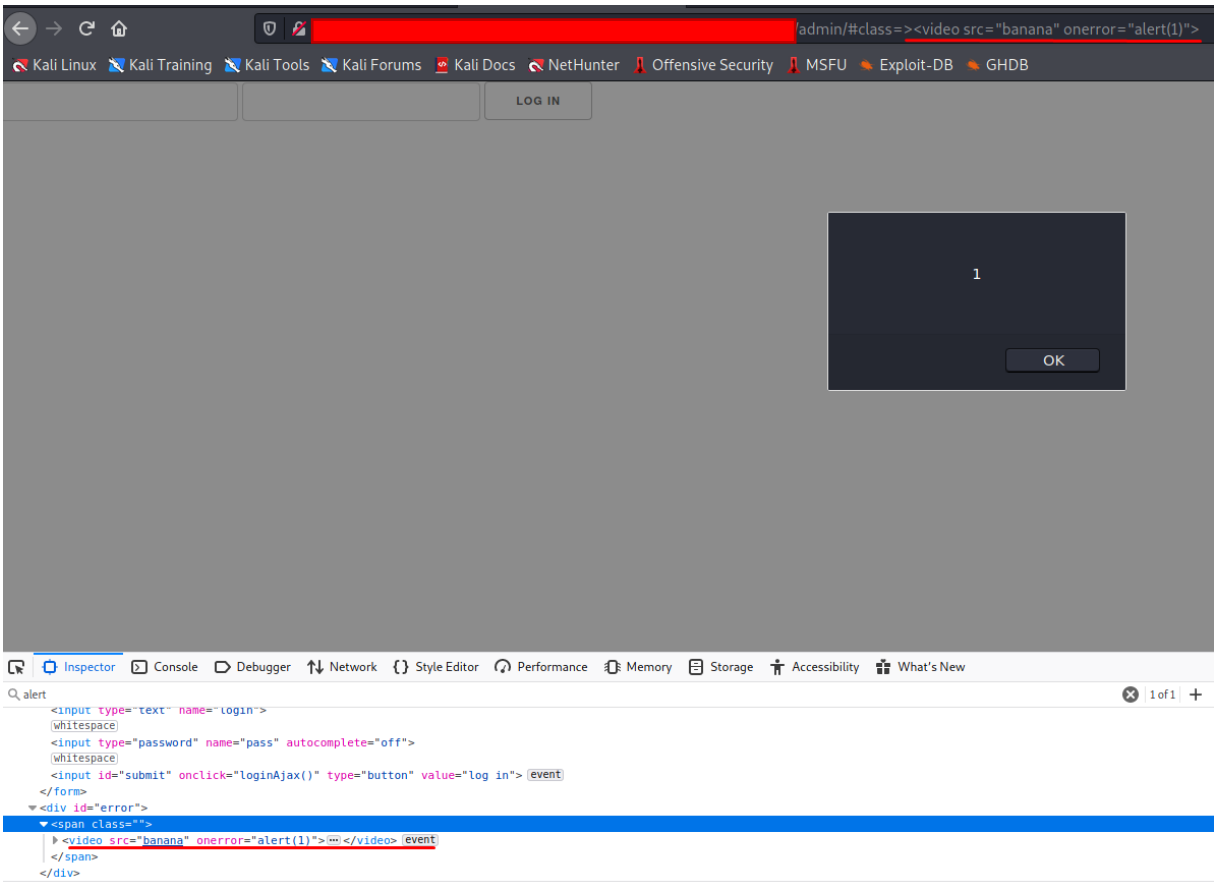
it was possible to fire DOM Based XSS through .InnerHTML.
Payload:
><video src="banana" onerror="alert(1)">
vulnerable code:

```
<script type="text/javascript">

  function hashUpdate(hash) {
    var p = hash.slice(1);
    if (p.startsWith("class=")) {
      document.getElementById("error").innerHTML = '<span ' + decodeURI(p) + '></span>';
    }
     else {
      document.getElementById("error").innerHTML = '';
    }
  }

  window.onhashchange = function() {
     hashUpdate(window.location.hash)
  };

  function loginAjax() {
    window.location.hash = "";
    var form = new FormData(document.querySelector("form"));
    var x = new XMLHttpRequest();
    x.onreadystatechange = function() {
      if (x.readyState === 4) {
        var response = JSON.parse(x.response);
        if (response.result) {
          window.location.reload(false);
        }
         else {
          window.location.hash = response.error;
        }
      }
    }
    x.open("POST", "authorize.php");
    x.send(form);
```



Mitigation:

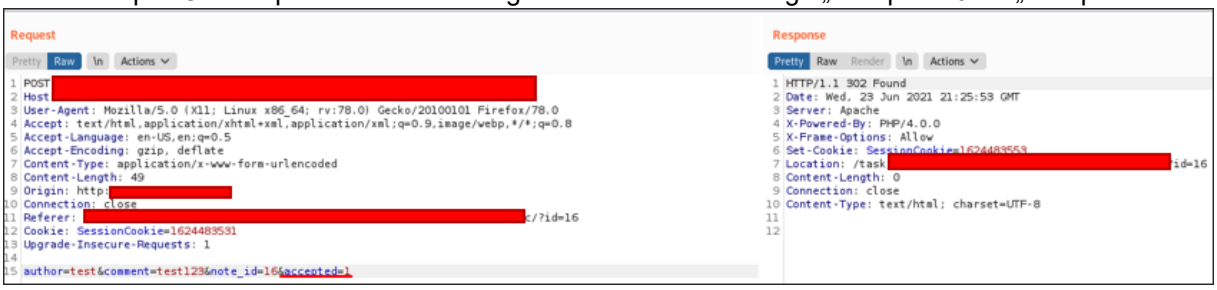
- Using frameworks that automatically escape XSS by design, such as the latest Ruby on Rails, React JS. Learn the limitations of each framework's XSS protection and appropriately handle the use cases which are not covered.
- Escaping untrusted HTTP request data based on the context in the HTML output (body, attribute, JavaScript, CSS, or URL) will resolve Reflected and Stored XSS vulnerabilities. The OWASP Cheat Sheet 'XSS Prevention' has details on the required data escaping techniques.
- Applying context-sensitive encoding when modifying the browser document on the client side acts against DOM XSS. When this cannot be avoided, similar context sensitive escaping

techniques can be applied to browser APIs as described in the OWASP Cheat Sheet 'DOM based XSS Prevention'.

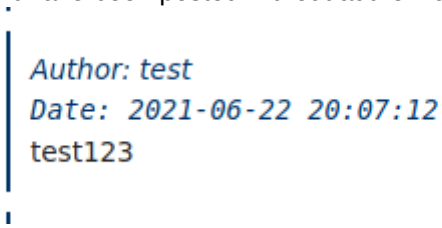
- Enabling a Content Security Policy (CSP) as a defense-in-depth mitigating control against XSS. It is effective if no other vulnerabilities exist that would allow placing malicious code via local file includes (e.g. path traversal overwrites or vulnerable libraries from permitted content delivery networks).

5. **Vulnerability: BROKEN ACCESS CONTROL**
   Severity: HIGH
   It is possible to post a comment without any authorization on any post page. For example:

   Intercept POST request after submitting a comment and change „accepted=0" to „accepted=1"



Comment have been posted without authorization and/or admin review:



Mitigation:

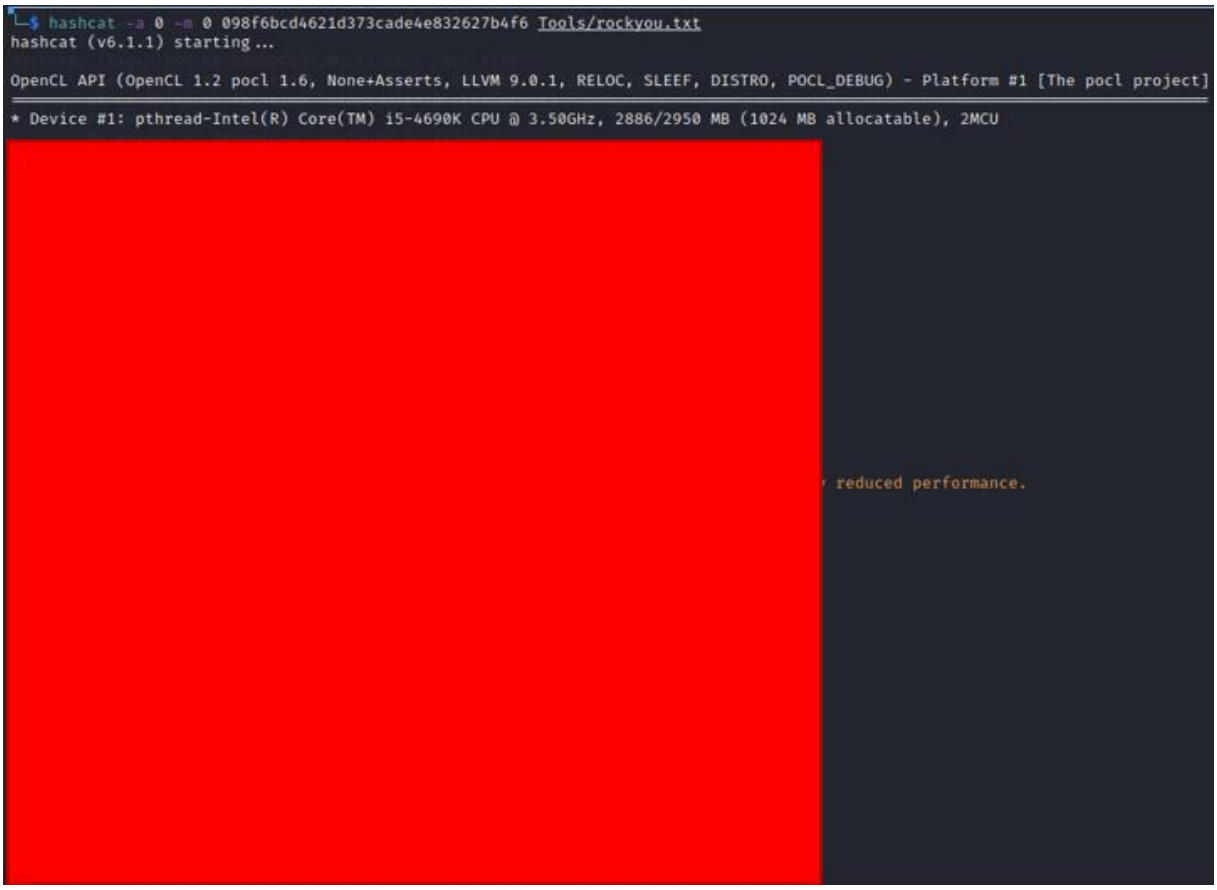
- Remove „accepted=0" parameter by default and allow only authorized users to perform post acceptance.

6. **Vulnerability: SECURITY MISCONFIGURATION**
   Severity: MEDIUM
   a. X-Frame_Options ALLOW on Login page:

Mitigation:

- Set to DENY or SAMEORIGIN

   b. Password hashes in database are stored in deprecated MD5 function:



Mitigation:

- upgrade to modern password hashing algorithm, ex: Argon2id, bcrypt, and PBKDF2.
- Use salt and pepper.

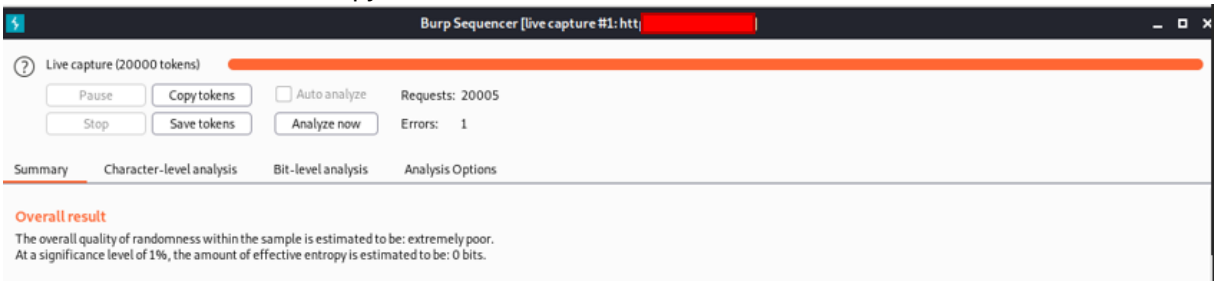
c. password policy allow weak passwords

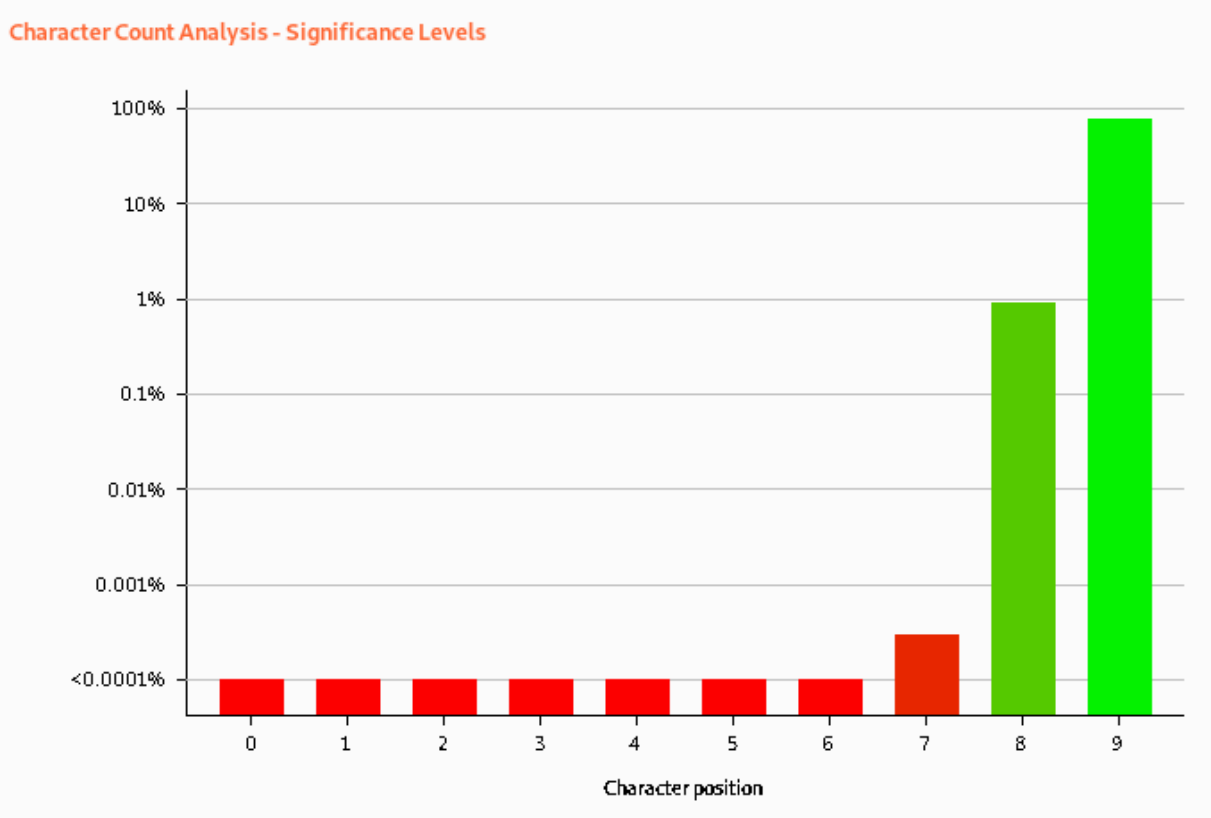
Mitigation:

- Implement Password Security Requirements policy, ex. according to ASVS:
  - Verify that user set passwords are at least 12 characters in length (after multiple spaces are combined).
  - Verify that passwords 64 characters or longer are permitted but may be no longer than 128 characters.

d. Cookies misconfiguration

Low cookie Entropy:





Mitigation:

- To ensure resistance to brute force attacks, the key generation algorithm must give truly unpredictable values with enough entropy to make guessing attacks impractical.

  Lack of Secure flags and attributes:

	HttpOnly	Secure	SameSite	L
	false	false	None	S
	false	false	None	S

Mitigation:

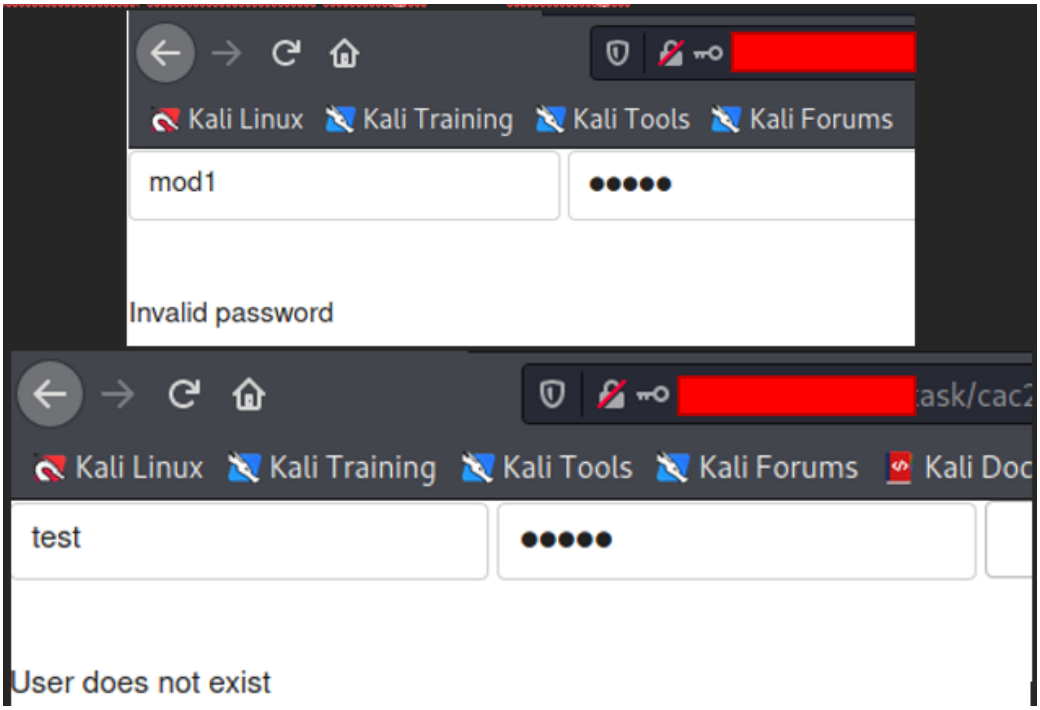
- Set the HttpOnly attribute using the Set-Cookie HTTP header to prevent access to cookies from client-side scripts. This prevents XSS and other attacks that rely on injecting JavaScript in the browser. Specifying the Secure and SameSite directives is also recommended for additional security.
- Use HTTPS to ensure SSL/TLS encryption of all session traffic. This will prevent the attacker from intercepting the plaintext session ID, even if they are monitoring the victim's traffic. Preferably, use HSTS (HTTP Strict Transport Security) to guarantee that all connections are encrypted.

7. **Vulnerability: SECURITY MISCONFIGURATION**
   Severity: LOW
   a. No rate limiting is implemented which allows to password brute force attacks.

Mitigation:

- Implement account lockout,session throttling and/or IP block.

b. Username enumeration through error message:



Mitigation:

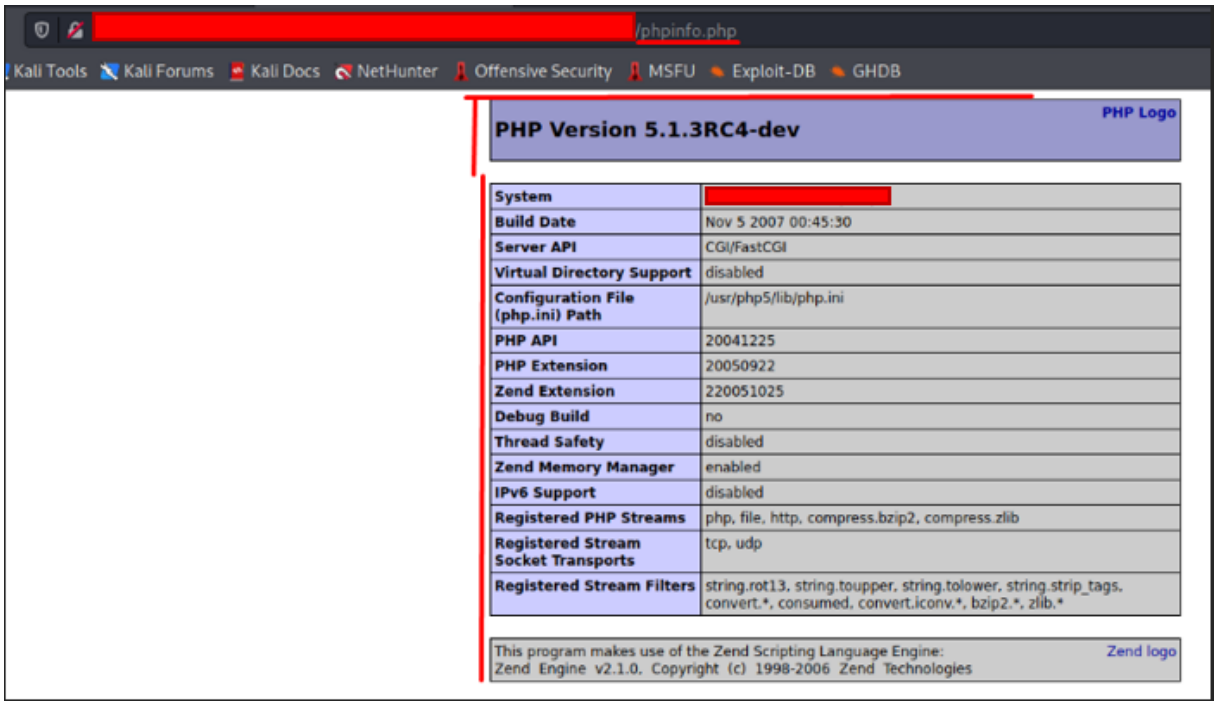
- Ensure the application returns consistent generic error messages in response to invalid account name, password or other user credentials entered during the log in process.

c. Passwords sent through HTTP.

Mitigation:

- Transmit Passwords Only Over TLS or Other Strong Transport

8. Vulnerability: SENSITIVE DATA EXPOSURE
Severity: INFORMATIONAL
Discovered that **phpinfo.php** file exposed which provided information about technology and software version:



Mitigation:

- Remove pages that call phpinfo() from the web server.