

Damn Vulnerable Web Application (DVWA)

Overview

Damn Vulnerable Web Application (DVWA) is an intentionally insecure PHP/MySQL web application designed as a learning and testing platform for web-application security. It provides a controlled environment where security professionals, developers, instructors, and students can explore common vulnerabilities, practice exploitation and mitigation techniques, and validate security tools without risking real production systems.

Purpose and Intended Use

DVWA's primary goals are to:

- Help security practitioners sharpen offensive and defensive testing skills in a legal, contained setting.
- Enable developers to understand how typical vulnerabilities arise and how to remediate them.
- Support educators in demonstrating real-world security concepts in classroom or lab environments.

Users are encouraged to approach DVWA either module-by-module (progressing through difficulty levels) or by focusing on individual vulnerabilities until they feel they have fully understood and successfully tested each scenario.

Contents and Features

- Multiple vulnerability categories covering common web threats (with configurable difficulty settings).
- Hints and tips available on each challenge page to guide learning.
- Supplementary references and reading links to deepen understanding of each issue.
- Both documented and intentionally undocumented weaknesses — included to encourage exploration and discovery.

Safety Recommendations

DVWA is deliberately insecure. To reduce risk:

- Never deploy DVWA on a public or production web server.
- Run DVWA only within an isolated environment such as a virtual machine (VirtualBox, VMware, etc.).
- Use NAT or host-only networking to limit exposure to your local network.
- For Windows/Linux test environments, consider using XAMPP, LAMP stacks, or similar local server packages inside the VM.
- Regularly snapshot or revert the VM after exercises to remove persistent changes or compromises.

Ethical and Legal Notice

DVWA is provided for educational and defensive security training only. Exploiting web vulnerabilities outside of expressly authorized, controlled environments is illegal and unethical. Users are responsible for ensuring they have permission to test any system they target.

Disclaimer

The authors and distributors of DVWA are not liable for misuse, damage, or compromises resulting from installation or operation of this software. By using DVWA you accept responsibility for the security and configuration of your testing environment. If DVWA is installed on an Internet-facing server and that server is compromised, the installer is solely responsible for any consequences.

Steps:

1. Configure DVWA on Docker,

```
(root@kali)-[~]
# docker pull sagikazarmark/dvwa
Using default tag: latest
latest: Pulling from sagikazarmark/dvwa
693502eb7dfb: Pull complete
e6c91bb380b4: Pull complete
e111b9773d58: Pull complete
55f12e04cfae: Pull complete
8f1b50e10184: Pull complete
Digest: sha256:1224167ccb59ad64751d52d7beb75fd445a252ae3c136
Status: Downloaded newer image for sagikazarmark/dvwa:latest
docker.io/sagikazarmark/dvwa:latest
```

2. Open kali terminal and start docker.

```
(root@kali)-[~]
# docker run --rm -it -p 8080:80 sagikazarmark/dvwa

[ ok ] Starting MySQL database server: mysqld ..
[info] Checking for tables which need an upgrade, are corrupt or were
not closed cleanly..
[....] Starting web server: apache2AH00558: apache2: Could not reliably dete
. ok
⇒ /var/log/apache2/access.log ⇐
tail: unrecognized file system type 0x794c7630 for '/var/log/apache2/access
⇒ /var/log/apache2/error.log ⇐
[Thu Jun 05 11:51:46.779714 2025] [mpm_prefork:notice] [pid 545] AH00163: Ap
[Thu Jun 05 11:51:46.779763 2025] [core:notice] [pid 545] AH00094: Command
```

DVWA configuration & Login Steps

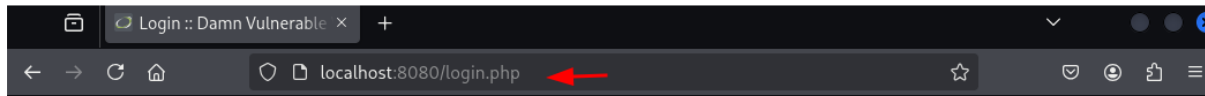
3. Access the Login Page

- Open DVWA in your browser (**http://localhost:8080/login.php**).

- Default Credentials:

- Username: admin
- Password: password

- Click Login.



Username


Password

Login

[Damn Vulnerable Web Application \(DVWA\)](#)

Welcome :: Damn Vulnerable

localhost:8080/index.php



HomeInstructionsSetup / Reset DBBrute ForceCommand InjectionCSRFFile InclusionFile UploadInsecure CAPTCHASQL InjectionSQL Injection (Blind)XSS (Reflected)XSS (Stored)DVWA SecurityPHP InfoAboutLogout

Welcome to Damn Vulnerable Web Application!

Damn Vulnerable Web Application (DVWA) is a PHP/MySQL web application that is damn vulnerable. Its main goal is to be an aid for security professionals to test their skills and tools in a legal environment, help web developers better understand the processes of securing web applications and to aid both students & teachers to learn about web application security in a controlled class room environment.

The aim of DVWA is to **practice some of the most common web vulnerability**, with **various difficulty levels**, with a simple straightforward interface.

General Instructions

It is up to the user how they approach DVWA. Either by working through every module at a fixed level, or selecting any module and working up to reach the highest level they can before moving onto the next one. There is not a fixed object to complete a module; however users should feel that they have successfully exploited the system as best as they possible could by using that particular vulnerability.

Please note, there are **both documented and undocumented vulnerability** with this software. This is intentional. You are encouraged to try and discover as many issues as possible.

DVWA also includes a Web Application Firewall (WAF), PHPIDS, which can be enabled at any stage to further increase the difficulty. This will demonstrate how adding another layer of security may block certain malicious actions. Note, there are also various public methods at bypassing these protections (so this can be seen as an extension for more advance users)!

There is a help button at the bottom of each page, which allows you to view hints & tips for that vulnerability. There are also additional links for further background reading, which relates to that security issue.

WARNING!

Damn Vulnerable Web Application is damn vulnerable! **Do not upload it to your hosting provider's public html folder or any Internet facing servers**, as they will be compromised. It is recommend using a virtual machine (such as [VirtualBox](#) or [VMware](#)), which is set to NAT networking mode. Inside a guest machine, you can downloading and install [XAMPP](#) for the web server and database.

Disclaimer

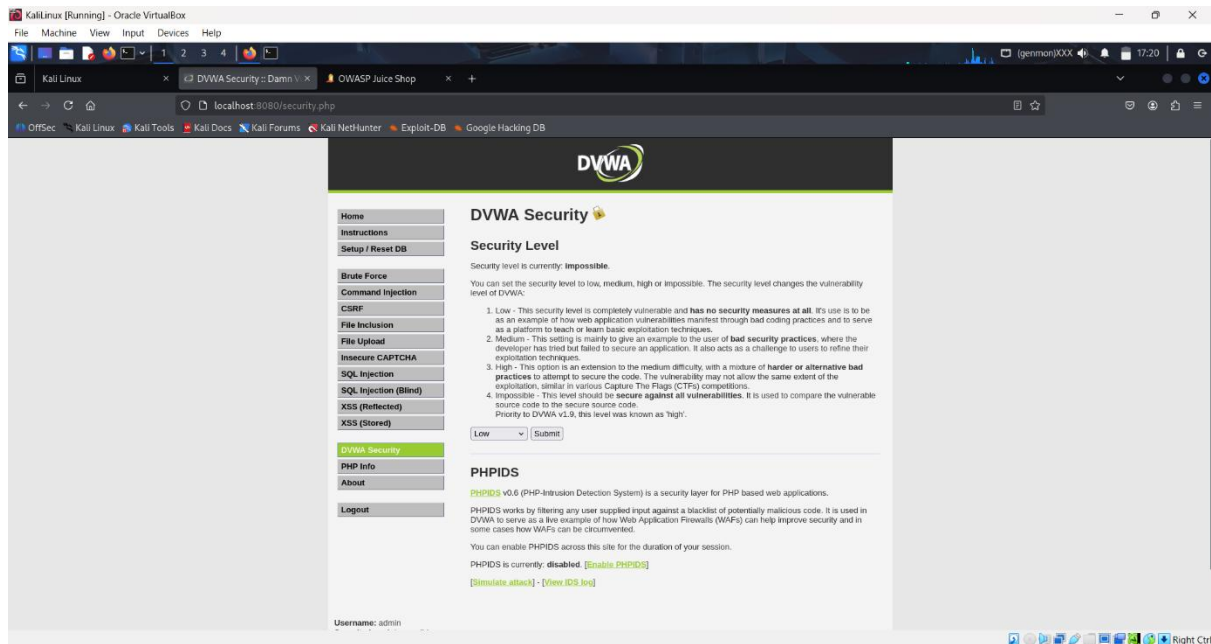
We do not take responsibility for the way in which any one uses this application (DVWA). We have made the purposes of the application clear and it should not be used maliciously. We have given warnings and taken measures to prevent users from installing DVWA on to live web servers. If your web server is compromised via an installation of DVWA it is not our responsibility it is the responsibility of the person/s who uploaded and installed it.

More Training Resources

DVWA aims to cover the most commonly seen vulnerabilities found in today's web applications. However there are plenty of other issues with web applications. Should you wish to explore any additional attack vectors, or want more difficult challenges, you may wish to look into the following other projects:

- [bWAPP](#)

4. Go to DVWA Security section and adjust as per your preferences.



Command Injection Vulnerability

Command injection occurs when an application incorporates untrusted input into operating-system commands without proper separation or validation, allowing an attacker to modify the intended command flow and execute arbitrary shell instructions. This typically happens when developers concatenate user data into functions like `system()`, `exec()`, `shell_exec()` or pass strings to a shell interpreter; an attacker can terminate the expected argument and append additional commands or use alternate encodings to bypass fragile filters. Because the web process hands off execution to the OS, the vulnerability effectively grants the attacker whatever capabilities the service account has, turning seemingly benign features (ping, diagnostics, file operations) into remote execution vectors.

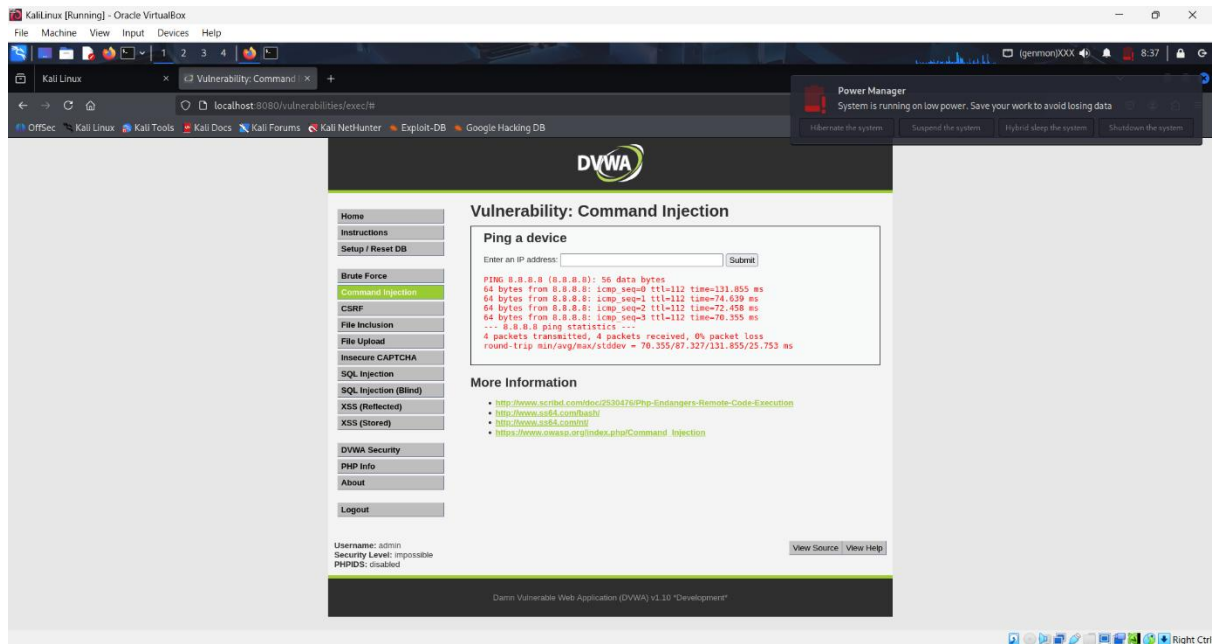
The consequences can be severe—data theft, service disruption, installation of persistent backdoors, or lateral movement within a network—each limited only by the privileges of the web process and the host operating system. Mitigation is straightforward in principle: remove the need to call a shell (use native libraries/APIs), enforce strict whitelist validation and canonicalization of inputs, and never interpolate raw user data into command strings. If an external binary is unavoidable, invoke it without a shell (use argument vectors or safe process libraries), use absolute paths, apply timeouts and least-privilege execution, escape and encode any output returned to users, and complement these measures with logging, rate-limiting, and runtime confinement (SELinux/AppArmor) to reduce impact.

insecure design (low security level)

At a low security setting, an application might build a command string by directly appending user input and then executing it. An attacker can extend that string — for example, by separating the intended command and adding additional commands — to force the server to run unintended operations. In short: instead of supplying only the expected argument, the attacker supplies additional shell syntax that changes program flow and executes extra commands.

Steps:

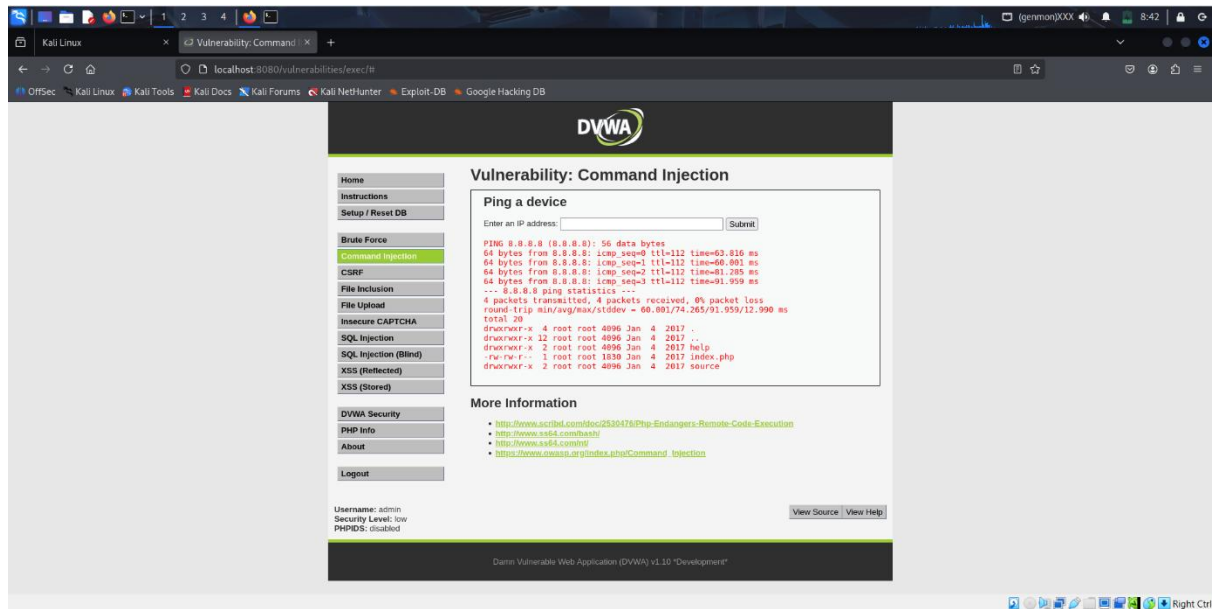
1. Test user input with enter IP address.



2. View source code for analysis.



3. Use payload to exploit vulnerability.



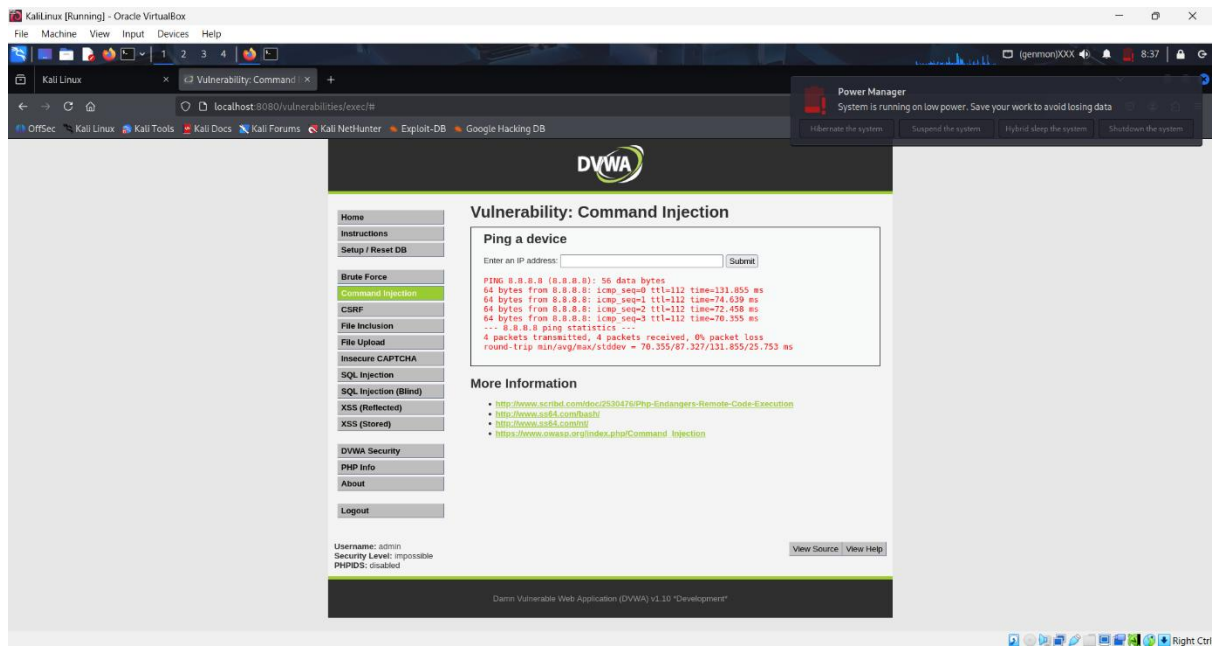
Medium Level Security

The developer has read up on some of the issues with command injection, and placed in various pattern patching to filter the input. However, this isn't enough.

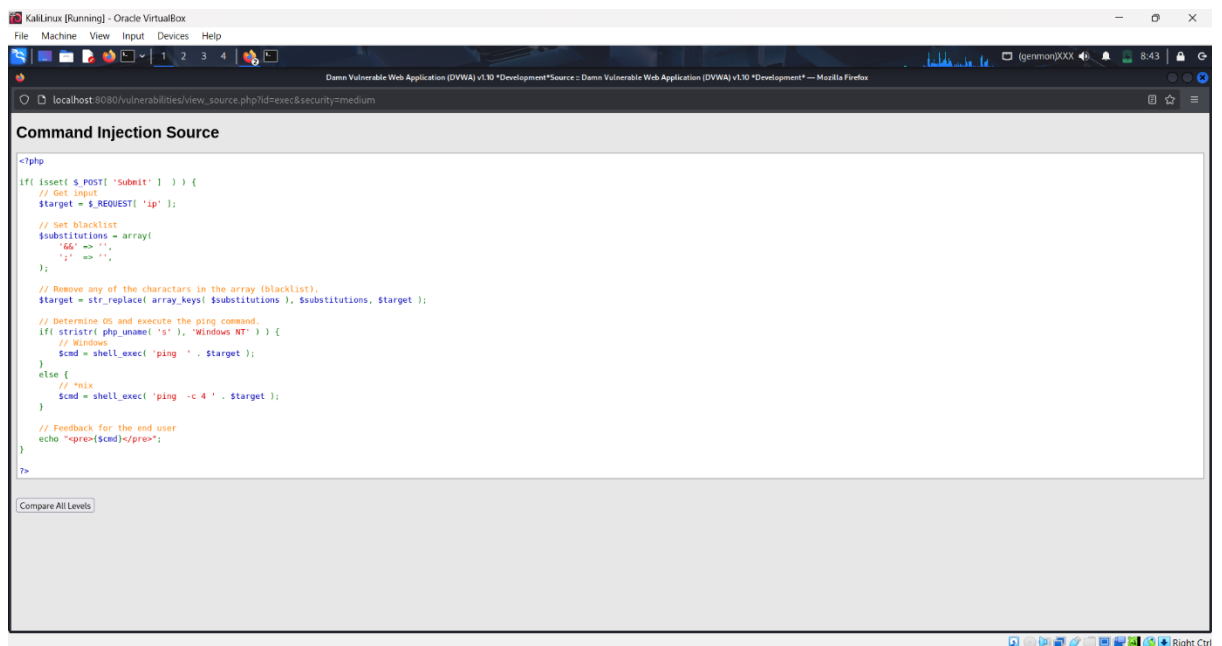
Various other system syntaxes can be used to break out of the desired command.

Steps:

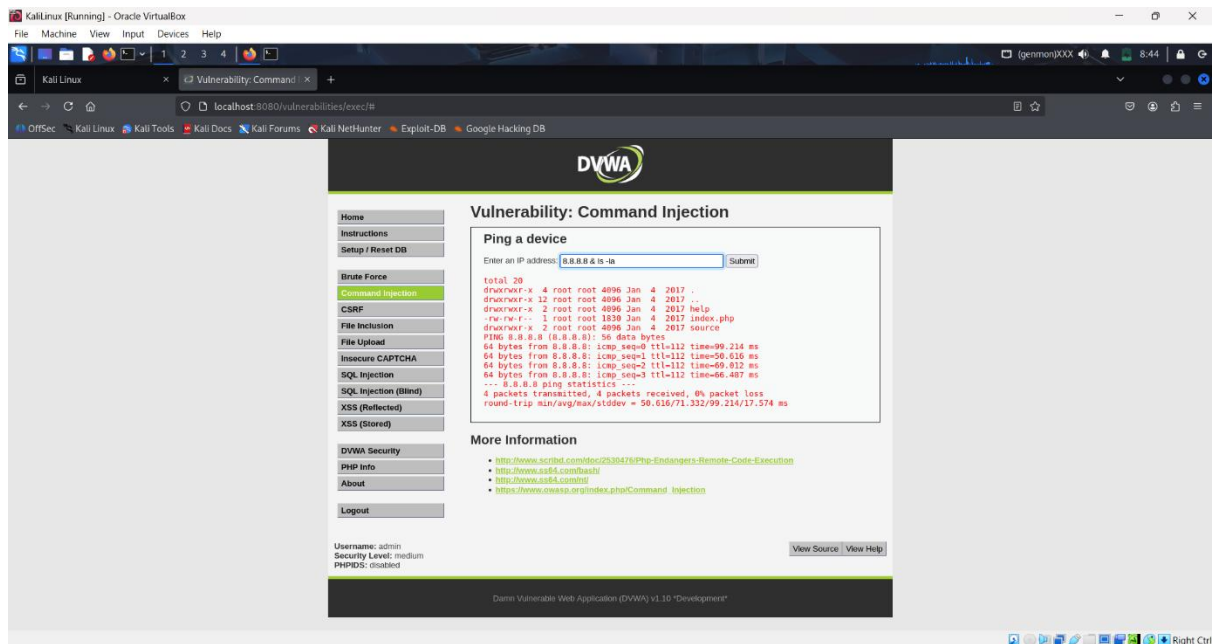
1. Test user input with enter IP address.



2. View source code for analysis.



3. Enter payload and exploit vulnerability.



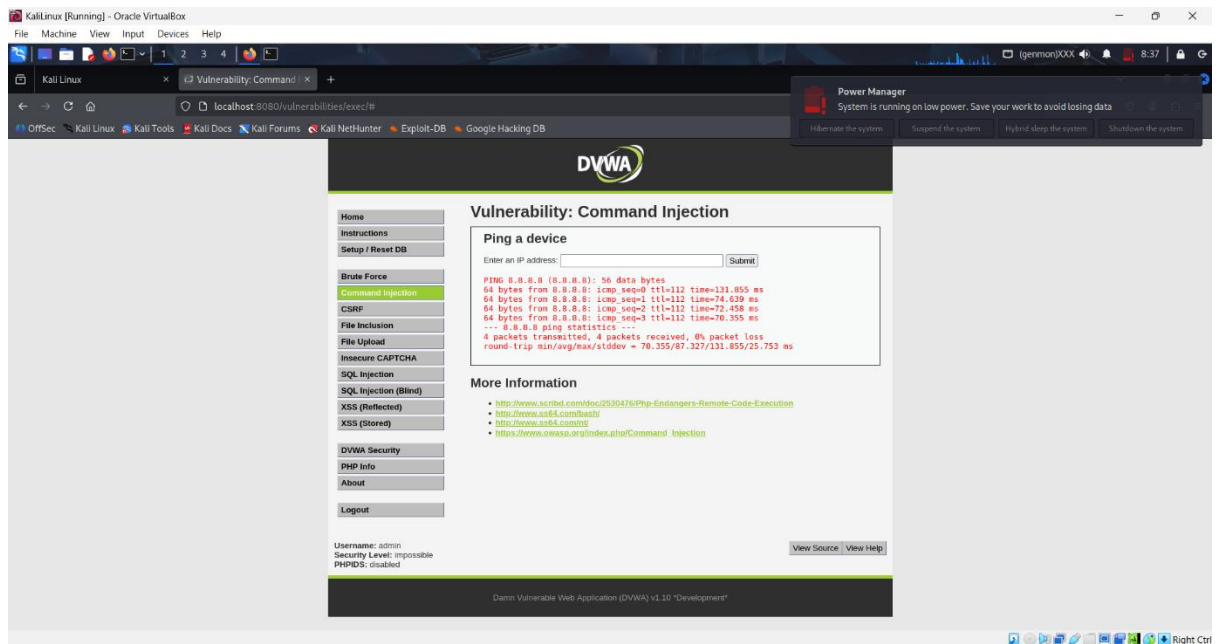
High Level Security

In the high level, the developer goes back to the drawing board and puts in even more pattern to match. But even this isn't enough.

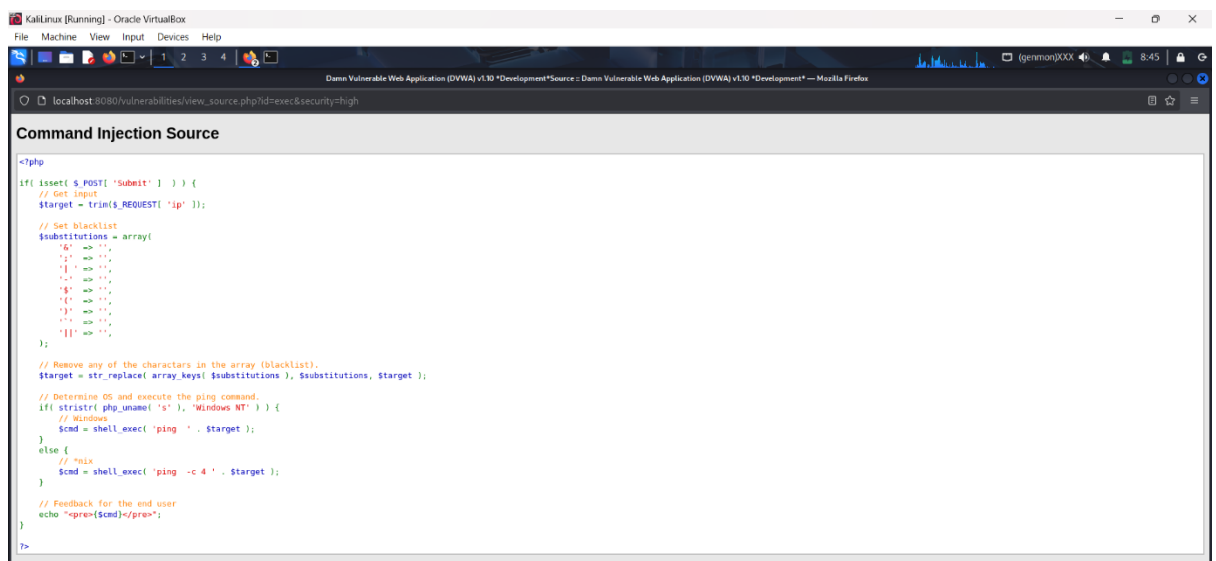
The developer has either made a slight typo with the filters and believes a certain PHP command will save them from this mistake.

Steps:

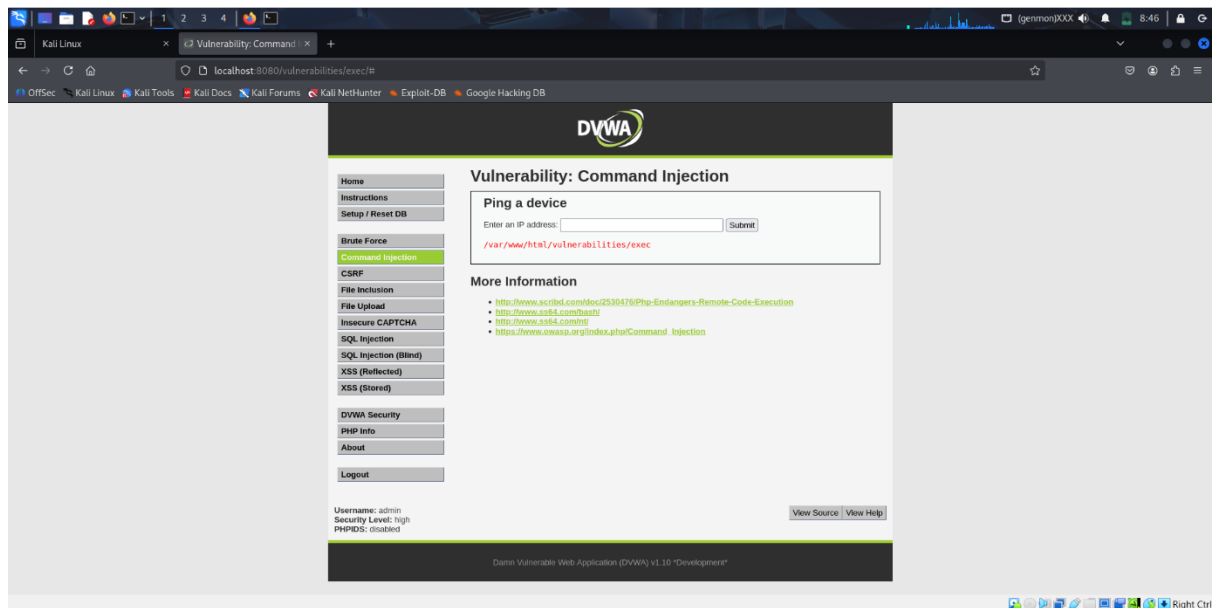
1. Test user input with IP address.



2. View source code for analysis.



3. Enter payload and exploit vulnerability.



Mitigation

1. Validate Input Strictly

Use canonical/whitelist validation—only accept inputs in a strict, predefined format (e.g., valid IPv4 addresses). Use trusted validators like `filter_var(..., FILTER_VALIDATE_IP)` or `inet_pton()`. Reject anything that doesn't match exactly, including ambiguous encodings (hex, octal) and control characters.

2. Avoid Shell Commands

Never interpolate raw user input into system commands. If absolutely necessary, use safer methods like `proc_open()` with arguments or `escapeshellarg()` with a fixed path. Prefer native network checks (e.g., socket APIs, DNS resolution, ICMP libraries) instead of shell binaries like ping.

3. Principle of Least Privilege

Run the web app under a low-privilege user. Use AppArmor/SELinux to restrict external commands and resources. Limit command execution privileges to only what's necessary.

4. Sanitize Output & Rate Limiting

Escape any output (e.g., `htmlspecialchars()`) to prevent reflected XSS. Add rate-limiting for sensitive operations (e.g., ping), restrict input length, and set timeouts to prevent abuse.

5. Logging & Alerts

Log all rejected inputs and suspicious activity. Set up alerts for high request rates or repeated failed attempts. Ensure CSRF tokens are valid and tied to sessions, regenerated periodically.

6. Error Handling

Provide generic error messages to users, logging precise reasons internally without revealing system details or command paths.

SQL Injections Vulnerability

SQL Injection (SQLi) occurs when an application improperly includes untrusted user input in SQL queries, allowing attackers to manipulate the query and execute arbitrary commands. This can lead to unauthorized data access, modification, deletion, and even system-level commands if the database user has elevated privileges. The attack typically works by injecting malicious SQL code through input fields, URL parameters, or headers, which the application fails to validate or sanitize.

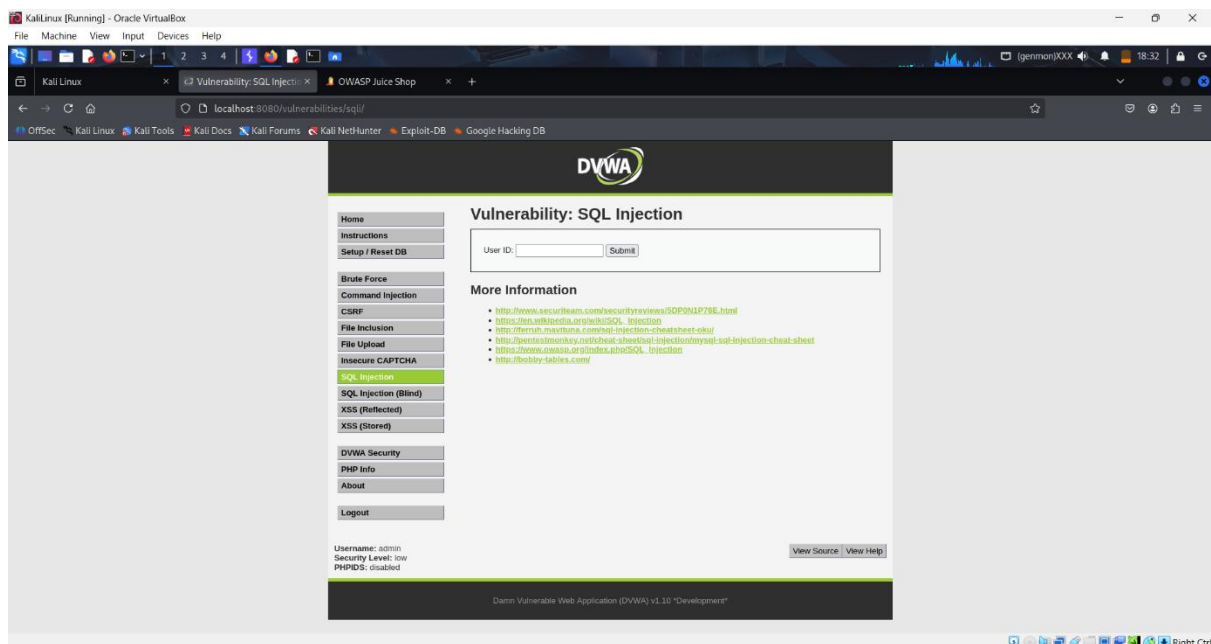
To prevent SQLi, avoid directly embedding user input into SQL queries. Use parameterized queries or prepared statements to securely pass input to the database. Additionally, validate and sanitize inputs, apply the principle of least privilege to database accounts, and monitor for suspicious activities. Properly configured stored procedures and query allow-lists can further harden the database.

Low Level Security

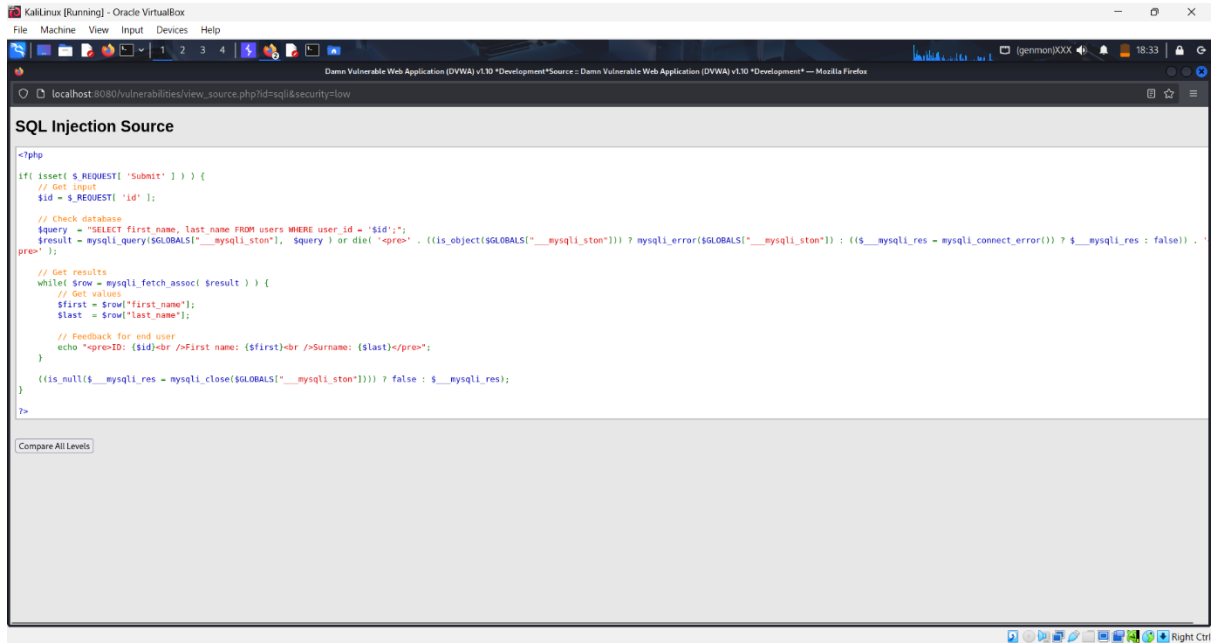
The SQL query uses RAW input that is directly controlled by the attacker. All they need to-do is escape the query and then they are able to execute any SQL query they wish.

Steps:

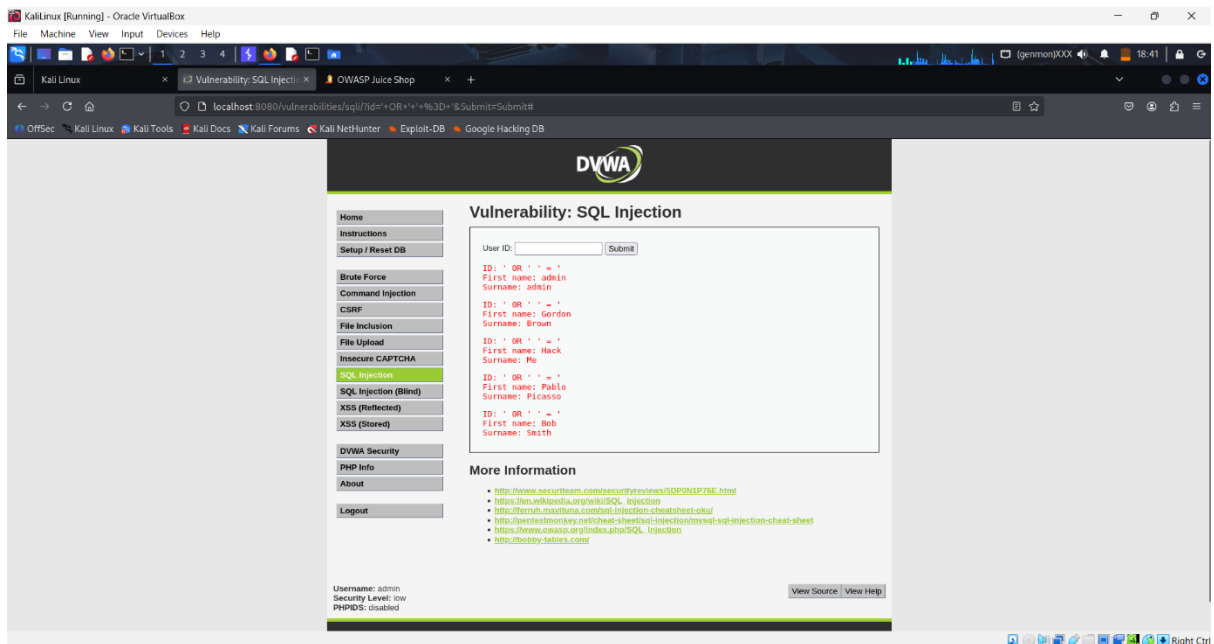
1. Open target web page.



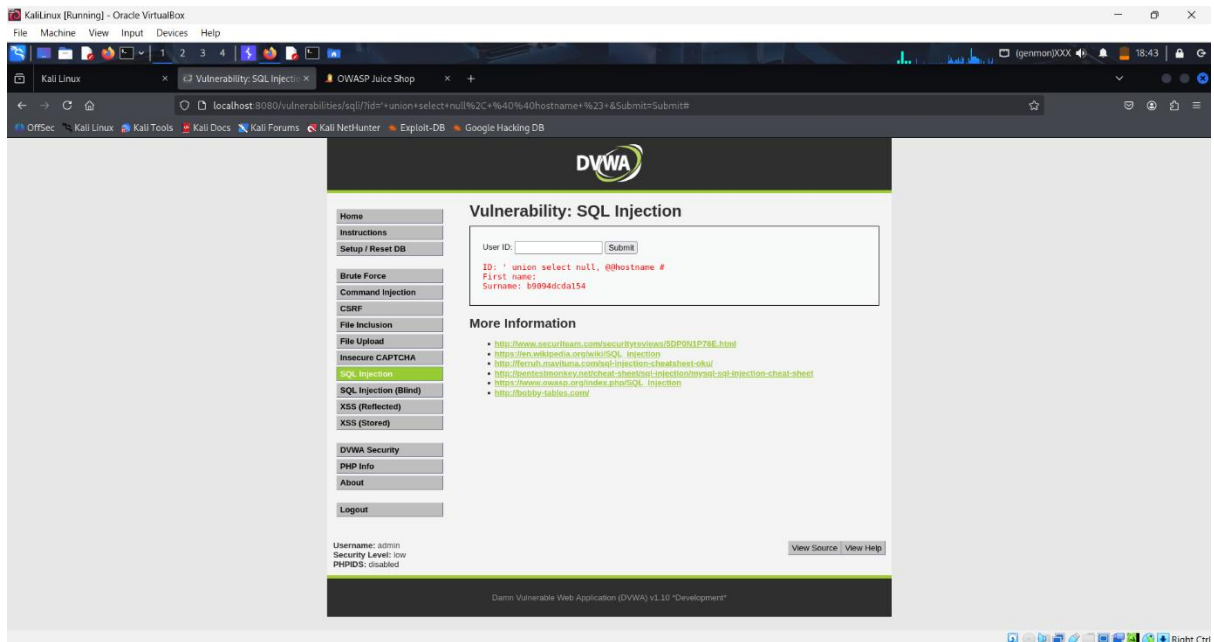
2. View source code for analysis.



3. Enter input for testing functionality of web page.



4. Enter the payload to exploit vulnerabilities of page.



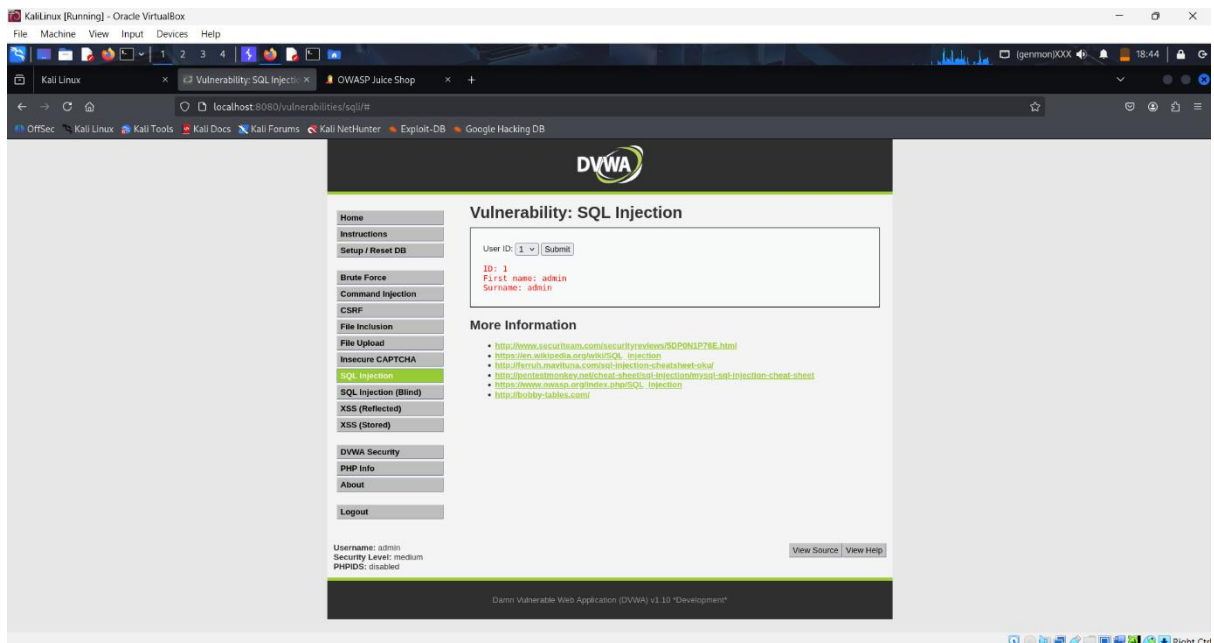
Medium Level Security

The medium level uses a form of SQL injection protection, with the function of "mysql_real_escape_string()". However due to the SQL query not having quotes around the parameter, this will not fully protect the query from being altered.

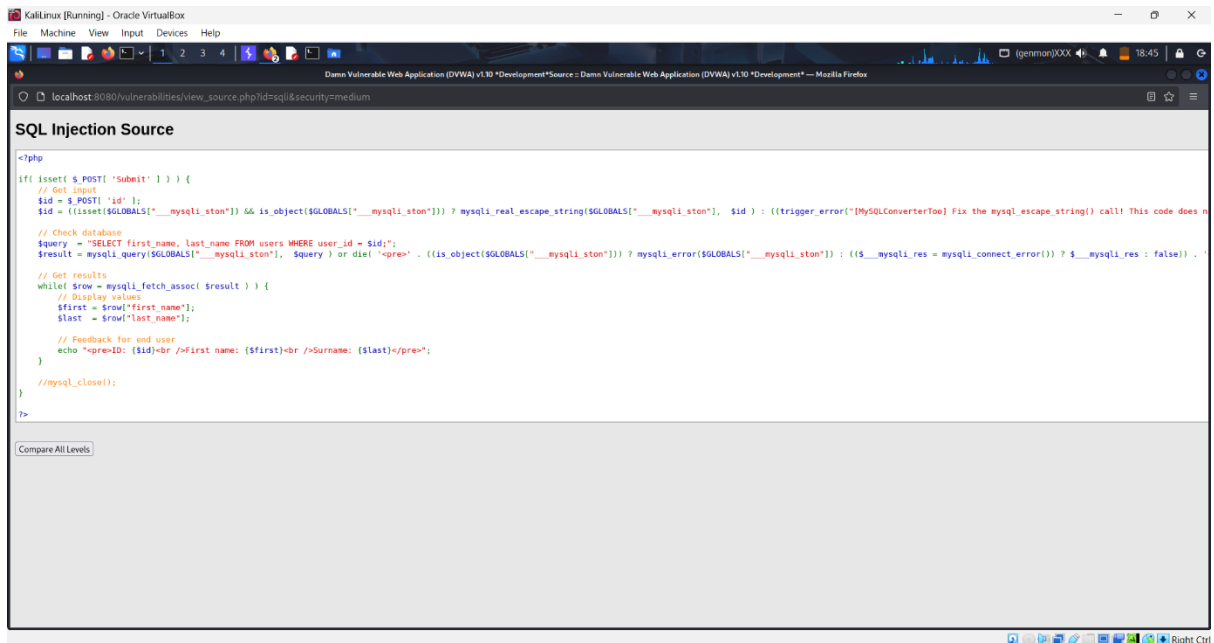
The text box has been replaced with a pre-defined dropdown list and uses POST to submit the form.

Steps:

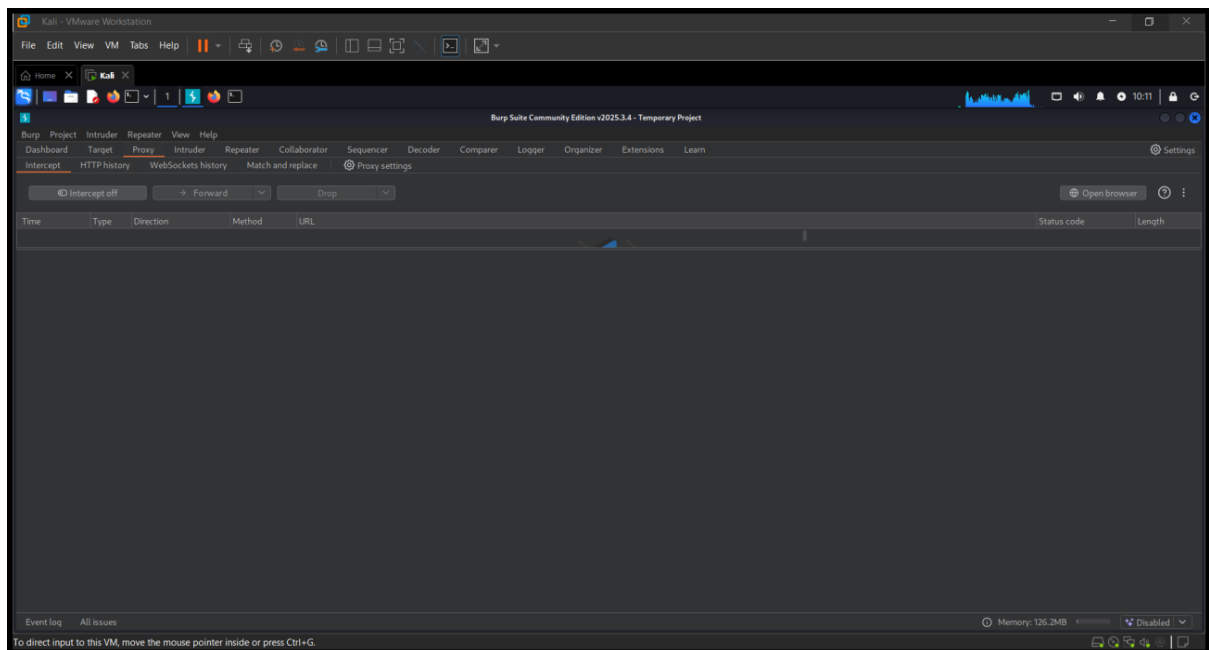
1. View target page for analysis functionality.



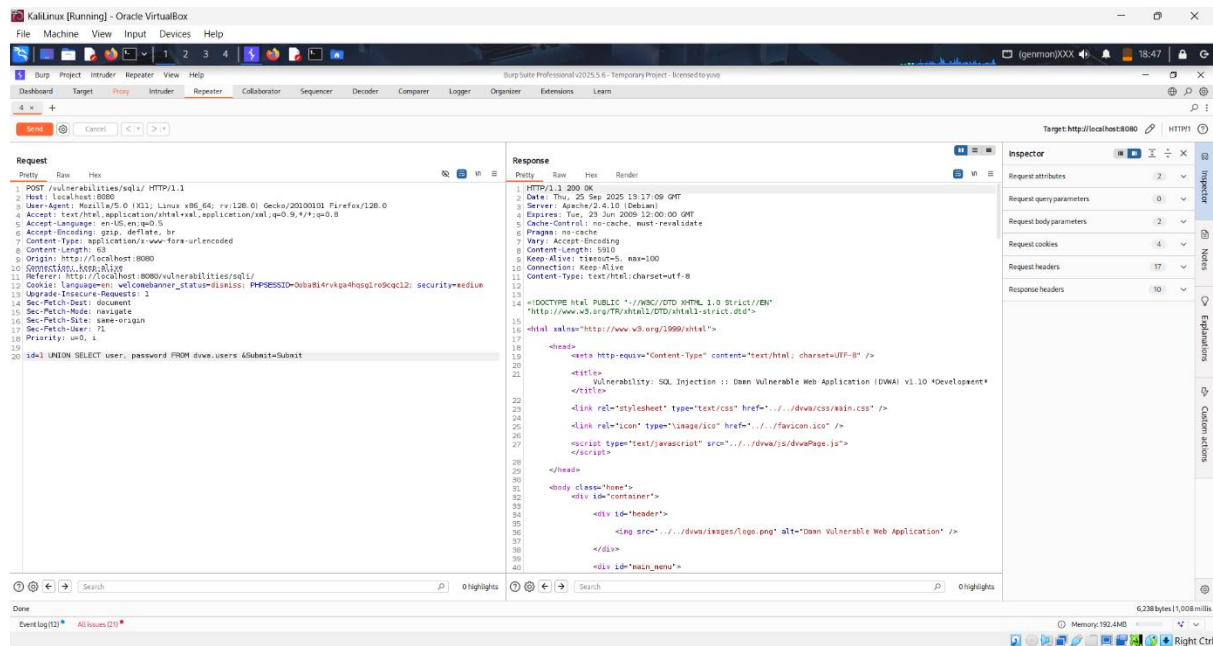
2. View source code for analysis.



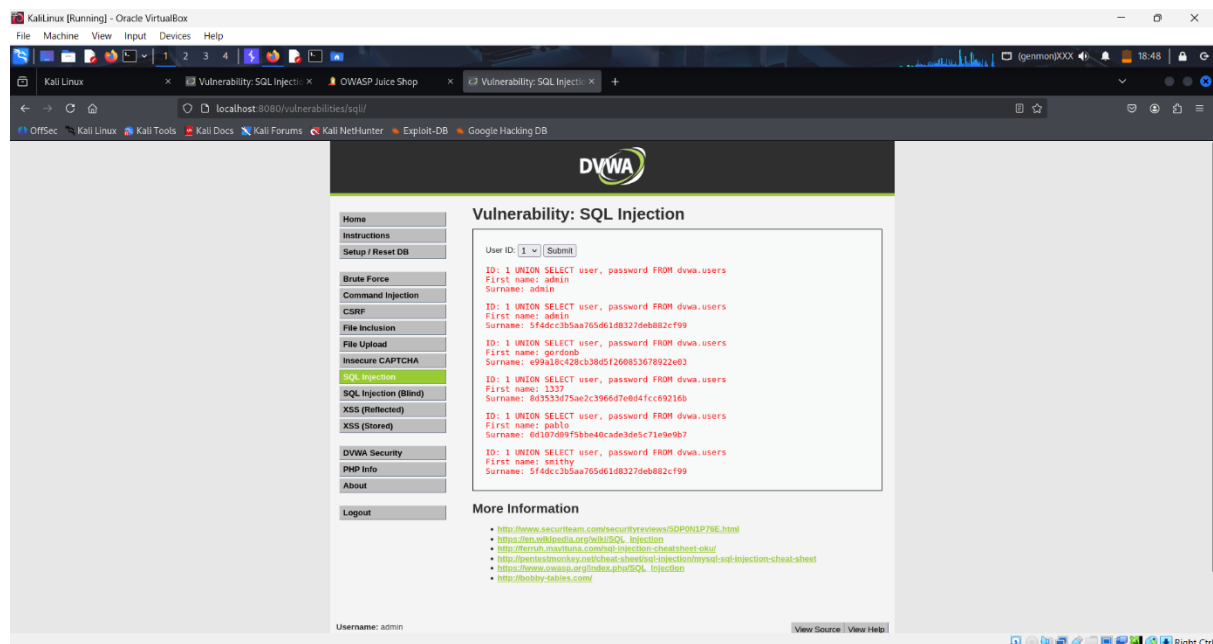
3. Turn on interception for capture traffic.



4. Intercept the request and Make changes in request for exploit vulnerabilities.



5. Successfully exploit it and gain sensitive information.

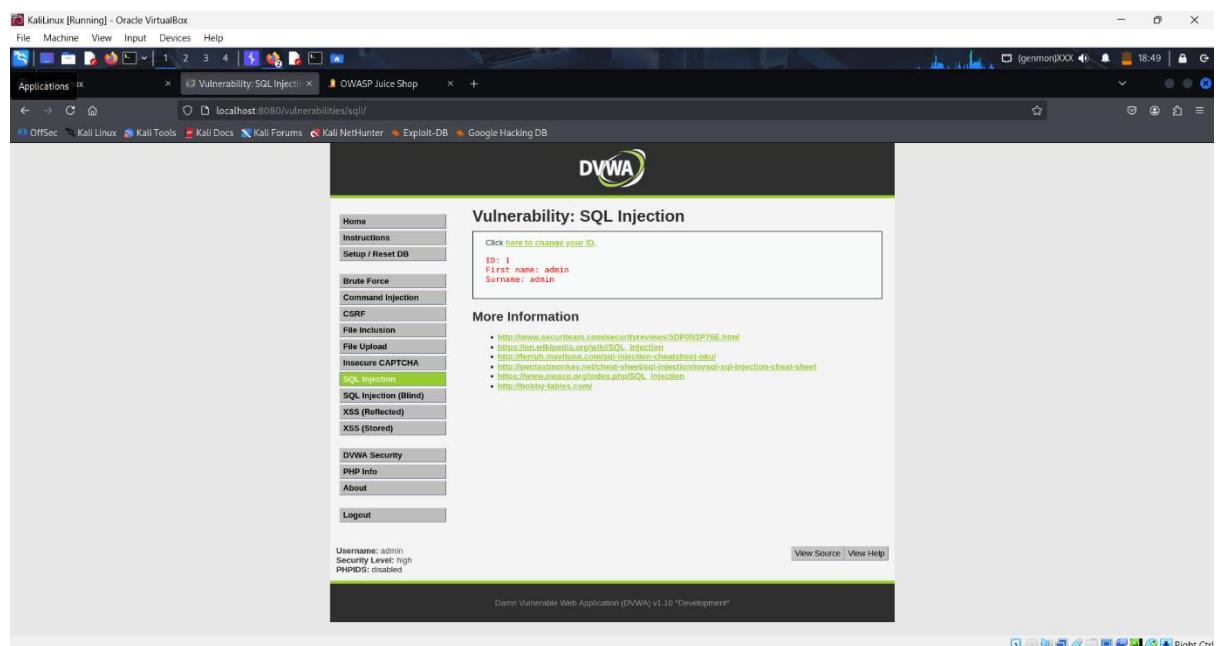


High Level Security

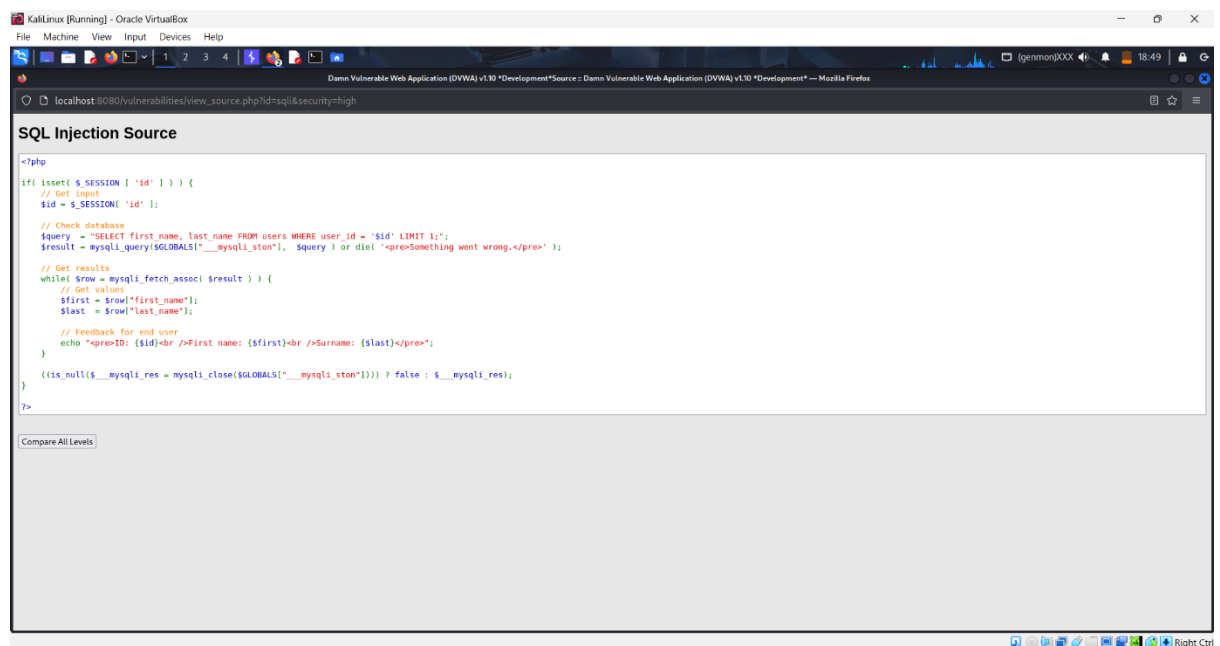
This is very similar to the low level, however this time the attacker is inputting the value in a different manner. The input values are being transferred to the vulnerable query via session variables using another page, rather than a direct GET request.

Steps:

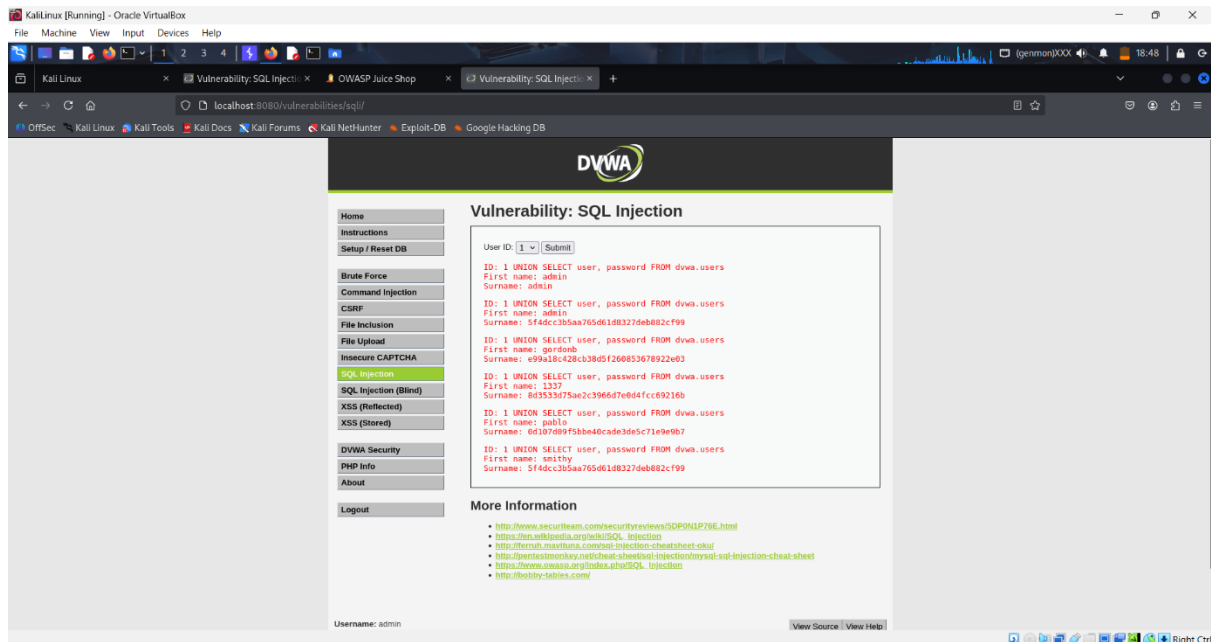
1. View target web for analysis of functionality.



2. View source page code for analysis.



3. Enter payload and exploit vulnerabilities.



Mitigation:

Use parameterized queries / prepared statements everywhere — never concatenate user input into SQL. Enforce strict input validation (whitelists, canonicalization) as a secondary layer, run the application with a least-privilege database account (no DDL/FILE/SUPER rights), and suppress detailed database error messages from end users while logging them securely. Prefer ORMs or safe query builders, and where possible replace dynamic SQL with stored procedures or a query allow-list.

Complement these fixes with defense-in-depth: deploy a WAF to block common payloads, add rate-limiting and monitoring/alerting for abnormal query patterns, remove dangerous DB functions (e.g., LOAD_FILE, INTO OUTFILE, xp_cmdshell), and include SQLi checks in CI/pen testing. For DVWA specifically, patch vulnerable challenge code to use prepared statements and reduce DB privileges, keeping vulnerable versions isolated in a VM for training.