Jada Young
CS370: Intro to Security
November 21, 2022
Damn Vulnerable Web Application

Introduction

This report provides an analysis of vulnerabilities present in Damn Vulnerable Web Application (DVWA)[1]. This document provides a summary of these findings and provides recommendations for addressing these vulnerabilities.

Deployment

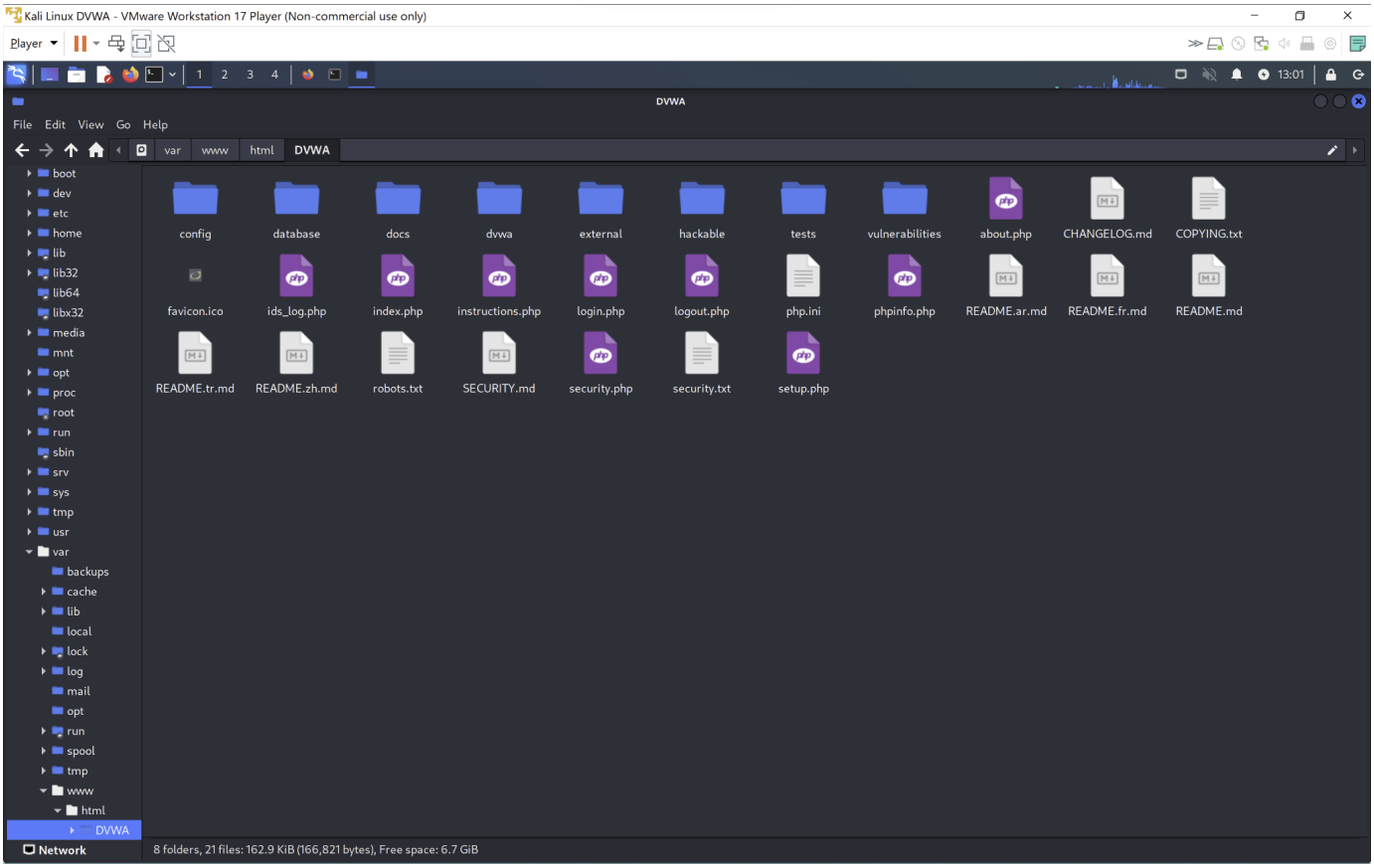
The DVWA Installation was completed using the following guide:

- https://nooblinux.com/how-to-install-dvwa/

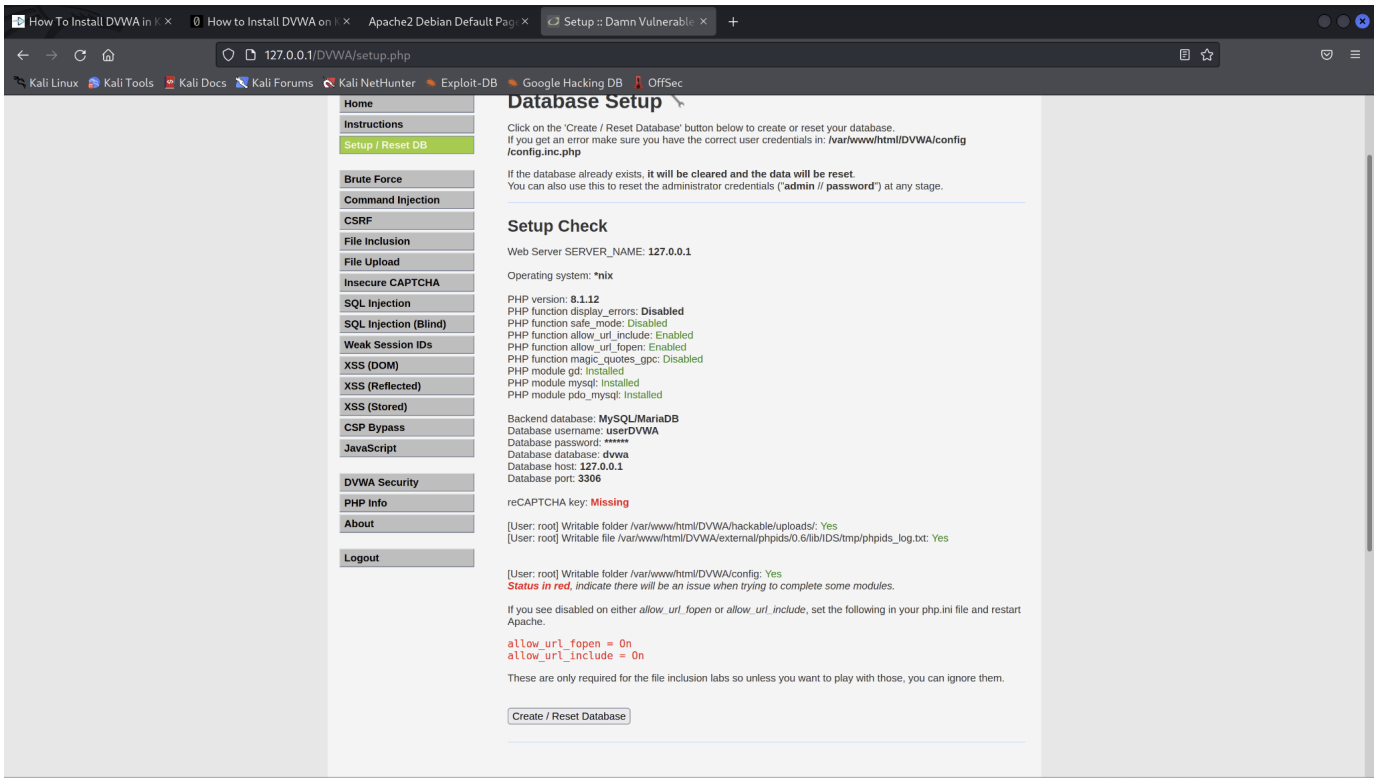
System Environment**:**

- Hypervisor: VMWare
- OS Distro: Kali Linux 2022.3
- Additional Software: Apache Web Server 2.4; PHP 8.1; MySQL 15.1

Installation Folder Structure



DVWA Database Setup



Vulnerabilities

Brute Force: Login

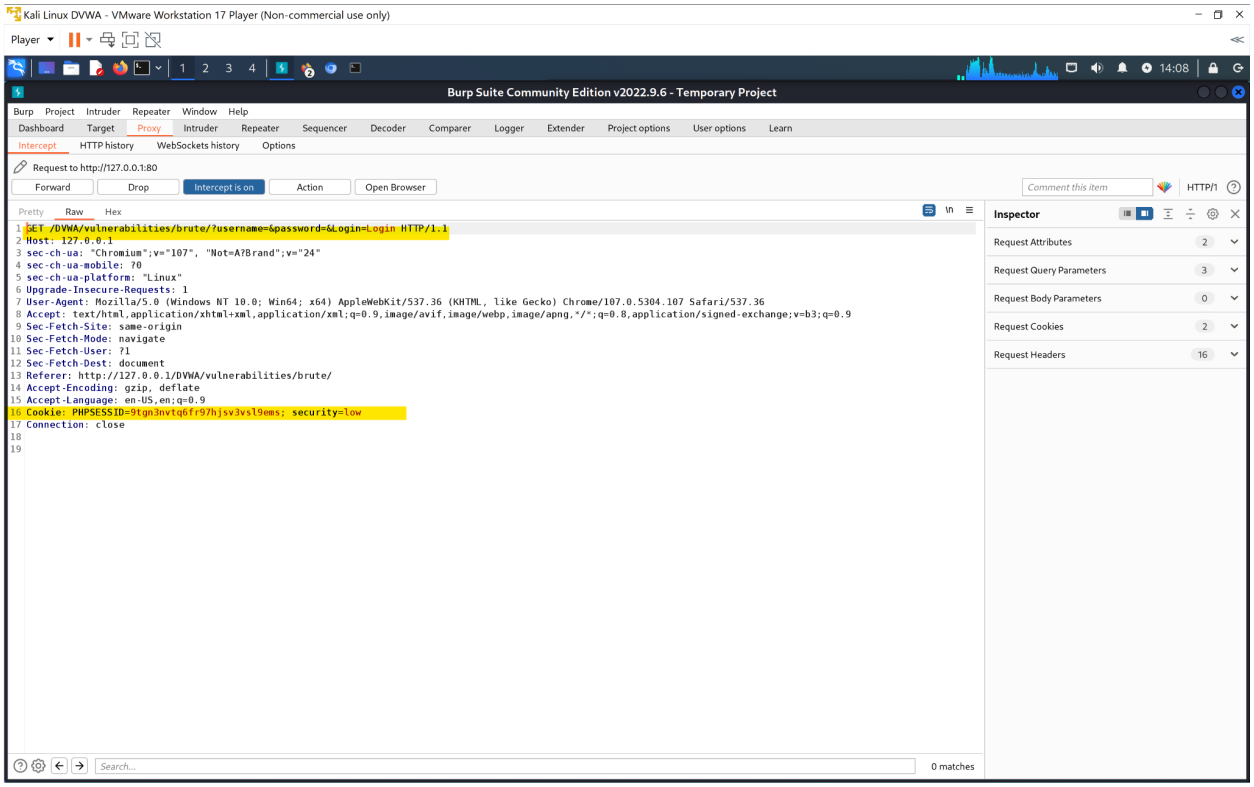
How does this feature normally work?

In order to log in, a typical user would enter their username and password (in this case, *admin* and *password*). The server then compares these credentials with valid credentials stored in its database. If there is a match, the user is able to successfully log in and is shown a success message. If the credentials are not found, an error message is displayed to the user.

What does it take to exercise the vulnerability?

Tools Used: BurpSuite Communiy Edition v2022.9.6, THC Hydra v9.3

Exploiting the Brute Force login vulnerability requires first analyzing the structure of the login attempt's HTTP request. In order to inspect the request, we can set up a proxy server that intercepts the browser's login request.



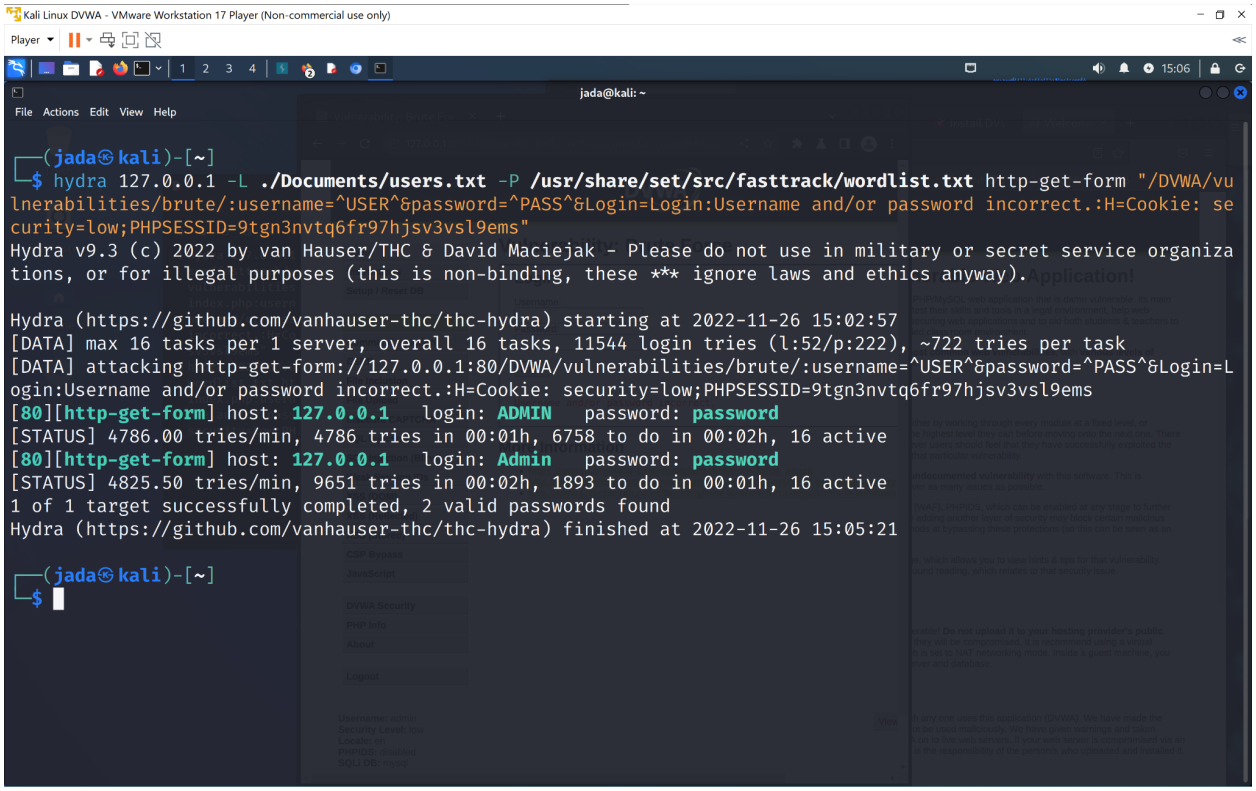
As shown in the previous figure, we can identify what information is required to recreate a login request:

- GET request
- Request parameters: username & password
- Session Cookie

Next, we can execute a brute force dictionary attack using THC Hydra. Using the information gathered during the inspection phase, we can test various HTTPS webform credential combinations against the authentication service. In the following figure, Hydra was provide with the following information for the complete command.

- target server: *127.0.0.1*
- URL path: *dvwa/vulnerabilities/brute/index.php:username=^USER^&password=^PASS^&Login=Login*
- Path to username dictionary
- Path to password dictionary
- Cookie: *PHPSESSID=9tgn3nvtq6fr97hjsv3vsl9ems; security=low*
- Failure message: *Username and/or password incorrect*

Hydra then attempts different credential combinations, reporting any successful matches.



Source: *DVWA Brute Force Tutorial (Low Security)*[2]

How and why did the feature work differently than normal use?

After using the applicable request information to construct the command, Hydra will perform a brute force attack on the authentication server by sending multiple GET requests with the username and password parameters containing potential username-password combinations. Hydra then evaluates the response received from the server. If the response contains the failure message, Hydra knows that the pair is invalid. If the response does not contain the failure message, Hydra recognizes this as a successful login attempt.

The adversarial approach differs from a normal use case because typically when logging into an account, users will enter their username and password. Typically, they only need to enter their credentials once but may try again if they incorrectly typed their credentials. In contrast, and adversary can try an unlimited combination of usernames and passwords in an attempt to identify a valid pair. In its current configuration, authentication server does not have a mechanism to distinguish between requests from a legitimate users attempting to access their account and requests from an adversary attempting to guess login information.

Implications and Recommendations

Currently, the user's credentials are the only roadblock between an adversary and the server's protected resources. If an attacker is able to identify the valid username and password, they will be able to access the password protected admin area of the site. The scope of the attack isn't just limited to the resources available to the compromised user, but could also impact other users on the network if the attacker has escalated privileges.

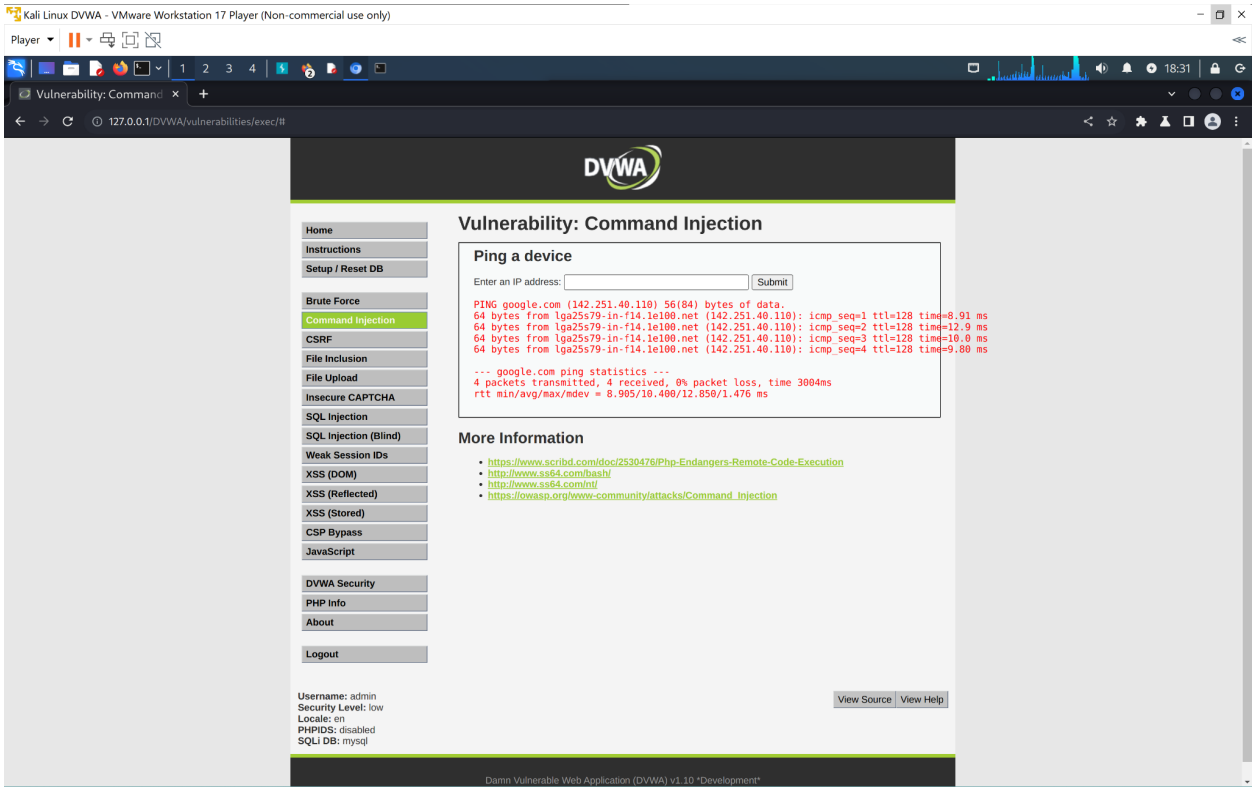
Limiting the effectiveness of a Brute Force attack could be accomplished in a variety of ways[3]:

- System administrators requiring users to utilize strong, unique password, decreasing the likelihood that an adversary can guess the password
- Using Captcha to distinguish between humans and computers.
- Limiting login attempts and implementing a time delay for multiple login attempts.
- Multi-Factor Authentication

Applicable OWASP Top 10: [A01:2021] Broken Access Control; [A04:2021] Insecure Design; [A05:2021] Security Misconfiguration; [A07:2021] – Identification and Authentication Failures; [A09:2021] – Security Logging and Monitoring Failures

Command Injection: Ping

How does this feature normally work?



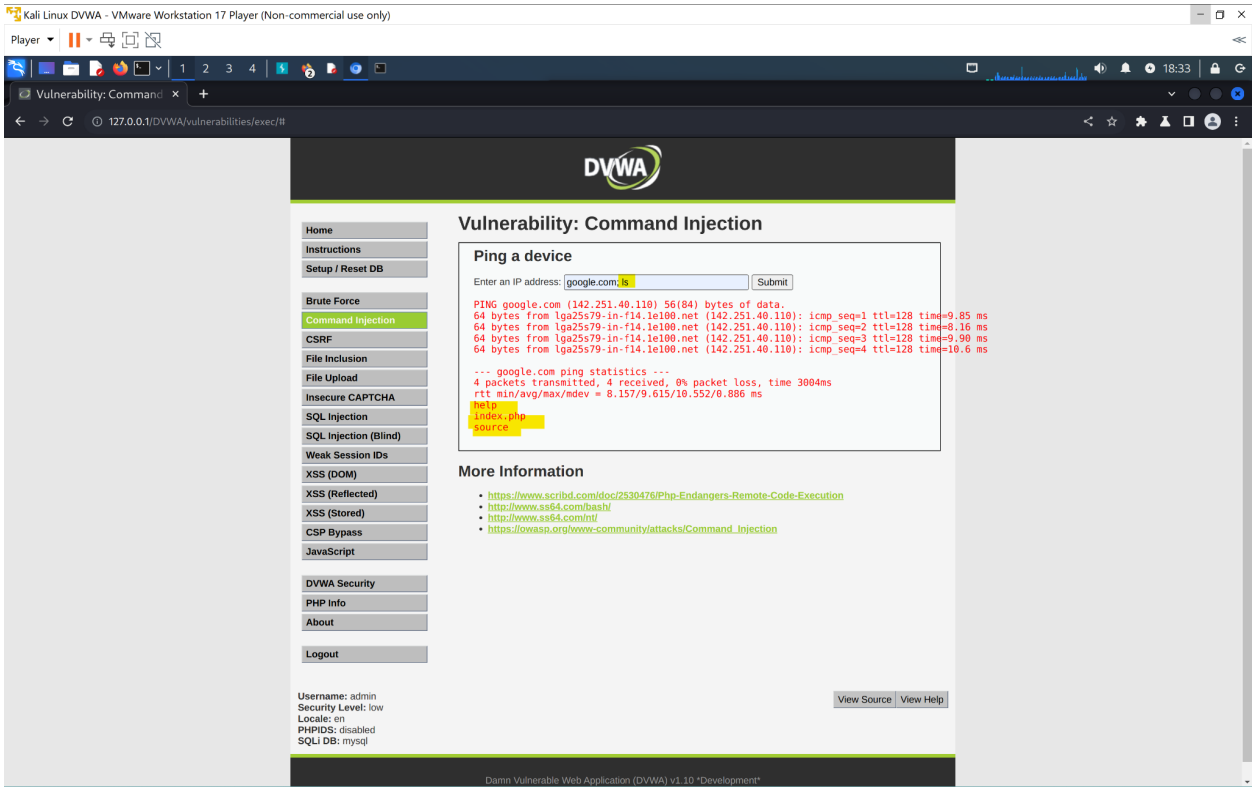
As seen in the above figure, this feature is intended to allow users to ping a provided target IP address in order to test if that network device is available. Once the user submits the target IP address, ping the server sends an echo request packet to the provided address.[4] When the target server receives the echo request, it responds with an echo reply packet. The browser then displays the results of the command.

Source: *How To Perform Command Injection Attacks (DVWA) For Aspiring Hackers!*[5]

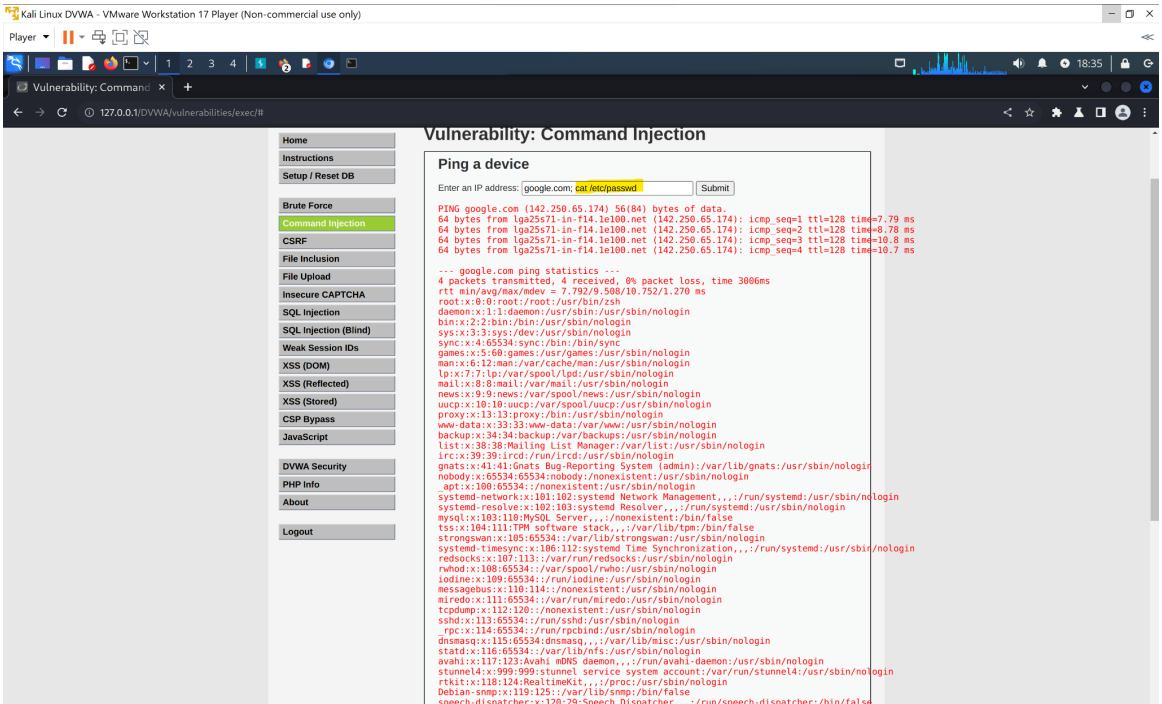
What does it take to exercise the vulnerability?

Typically, a system can be vulnerable to a Command Injection attack because it does not perform sanitization of a user's input. As demonstrated in the previous section, we can see that the server issues a command in the form of *ping <user input>*. We can use this assumption, to test the server's behaviour when additional commands are entered as user input.

As shown in the figure below, we provide the server with an expected input (google.com). In addition, we use the *;* operator to chain the expected input with an additional command. When the server receives the submission *`google.com; ls`*, the user is shown the results of ping and the folders present in the server's current working directory.



Since the server does not validate user-provided input, we can issue a variety of commands that allow us to gain more information about the system. As shown in the next example, we can execute the additional command *'cat /etc/passwrd'* to view users registered to the system.



How and why did the feature work differently than normal use?

In its current implementation, the server assumes that all input provided by the user is valid. As long as the server receives a valid URL or IP address for ping, a malicious user can also chain additional commands with their input.

The server then executes the additional command(s) and transmits all of the results in its response, which allows the malicious user to verify that command injection is possible.

Implications and Recommendations

Command Injection allows an adversary to execute system commands, usually with the same privilege level as the vulnerable application. As shown in the previous examples, an adversary can use this vulnerability to execute a variety of commands on the host system. Depending on the application privileges, attackers maybe able to view protected files, download malicious code, or explore the system environment.

Preventing Command Injection can be accomplished by[6]:

- Whitelisting of acceptable user inputs
- Using existing APIs or built-in library function as opposed to system calls
- Sanitizing user input for malicious characters

Applicable OWASP Top 10: [A03:2021] Injection

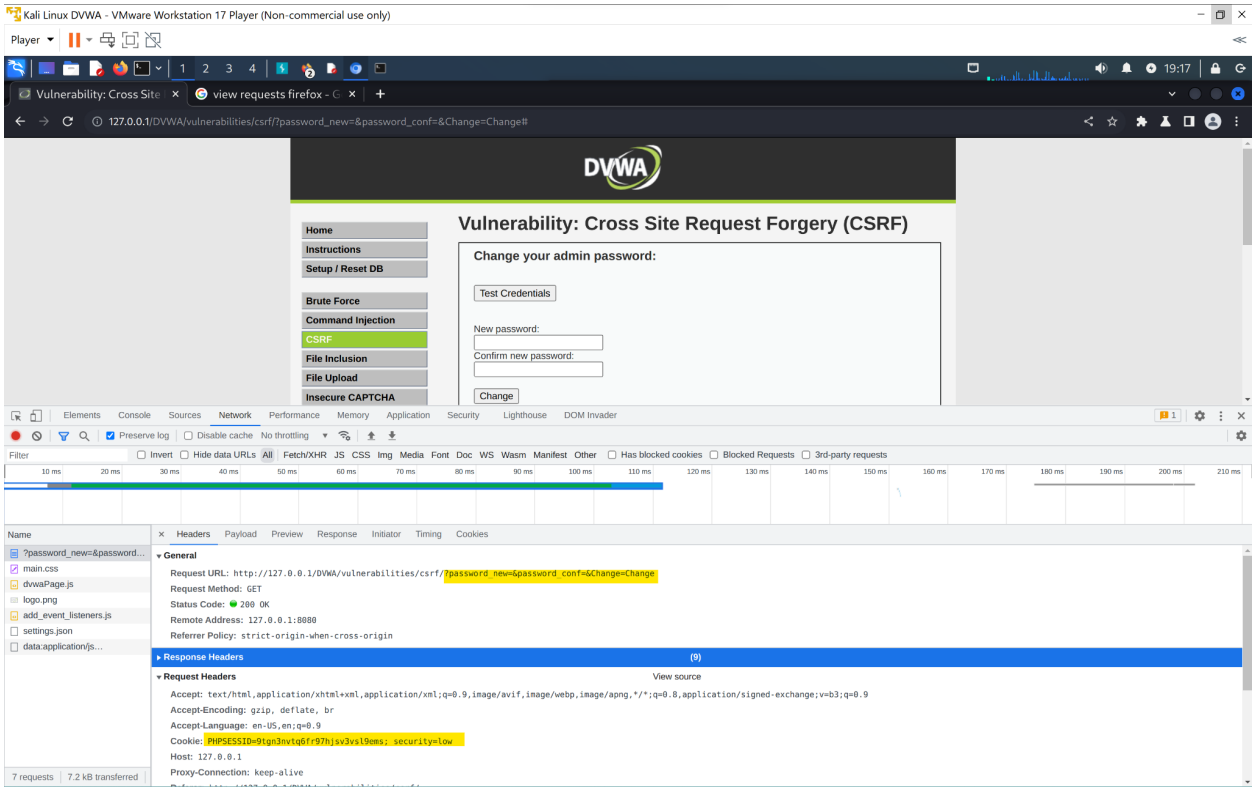
Cross Site Request Forgery (CSRF): Change Password

How does this feature normally work?

The Change Password page is intended to allow the admin user to change their password. The user enters their new password then confirms the password. The server then updates the admin user's password in its database.

What does it take to exercise the vulnerability?

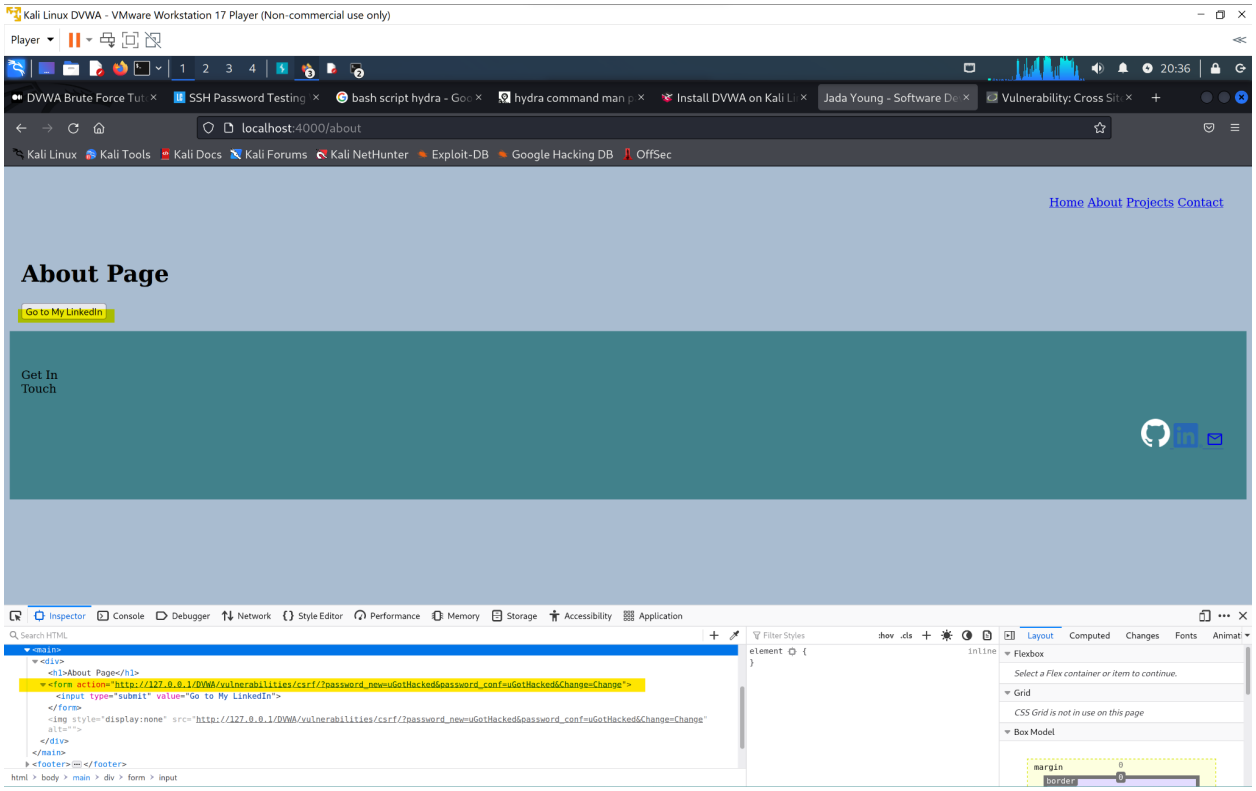
Cross Site Request Forgery (CSRF) allows an adversary to perform unwanted actions in a web application using an authenticated user's session. In order to execute a CSRF attack on the Change Password page, we first inspect the HTTP request sent when the change password form is submitted.



As shown in the above figure, we can note the user's session cookie. In addition, we can observe that the Change Password request sends a GET request with the following parameters:

- password_new
- password_conf
- Change=Change

Once we have gathered the structure of the URL, we can trick the user to visit a malicious site that sends a request to change password form, impersonating the original user.



If the user clicks the button, they will be redirected to the URL *http://127.0.0.1/DVWA/vulnerabilities/csrf/?password_new=uGotHacked&password_conf=uGotHacked&Change=Change* and their password will be changed to *uGotHacked*.

Source: *DVWA CSRF Tutorial (Low Security)*[7]

How and why did the feature work differently than normal use?

The password change service authenticates the user based on their session cookie. Since a password change is sent using a simple GET request, it is easy to manipulate the parameters to provide the server with the desired password. If a valid user is not careful, they can easily be socially engineered into clicking on the modified link, allowing the adversary to update their password.

Implications and Recommendations

The CSRF vulnerability allows an adversary to hijack a users account. Once the password is changed, the attacker can log into the user's account potentially exposing sensitive information. In addition, they may assume the role of the victim in the system.

If the compromised attack is a higher level admin account, they maybe able to view sensitive system information or compromise other users.

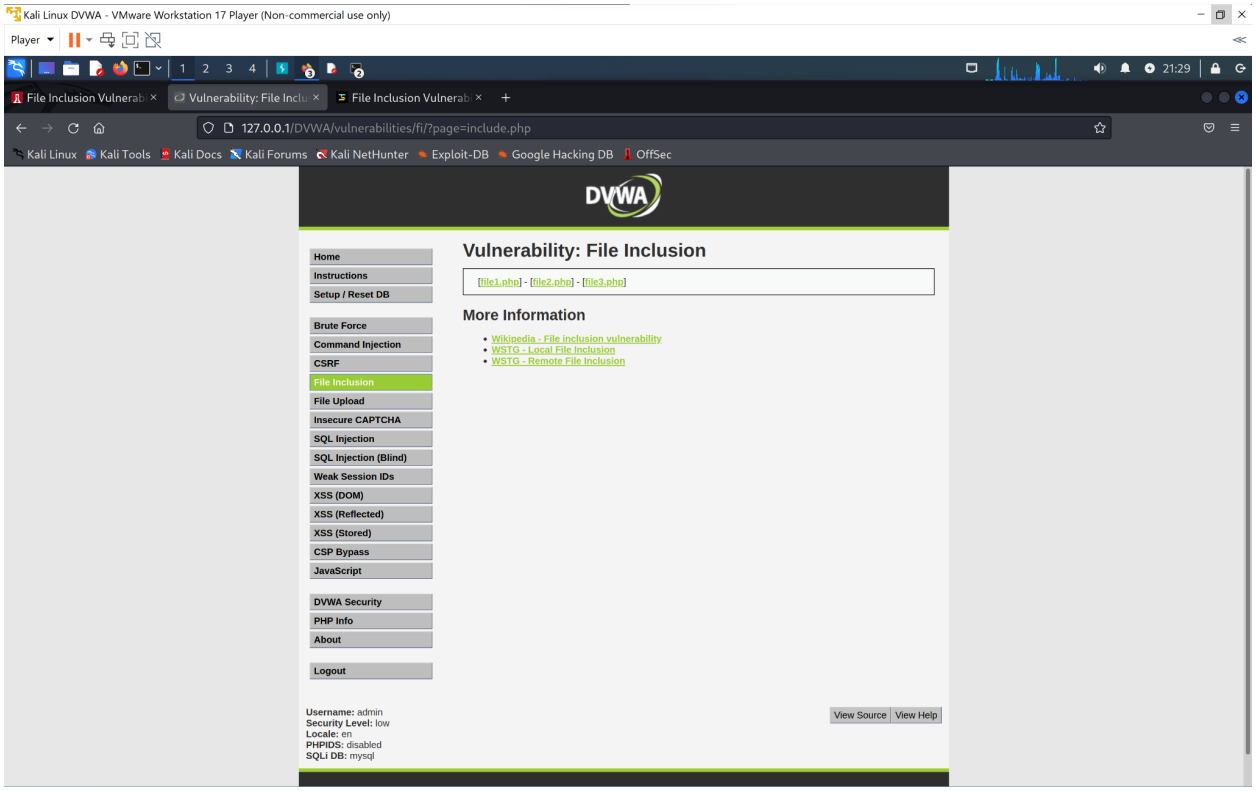
At the user level, CSRF attacks can be prevented by encouraging users to log off of sites before visiting others and encouraging them to clear their cookies. When designing web applications, CSRF vulnerabilities can be addressed by[8]:

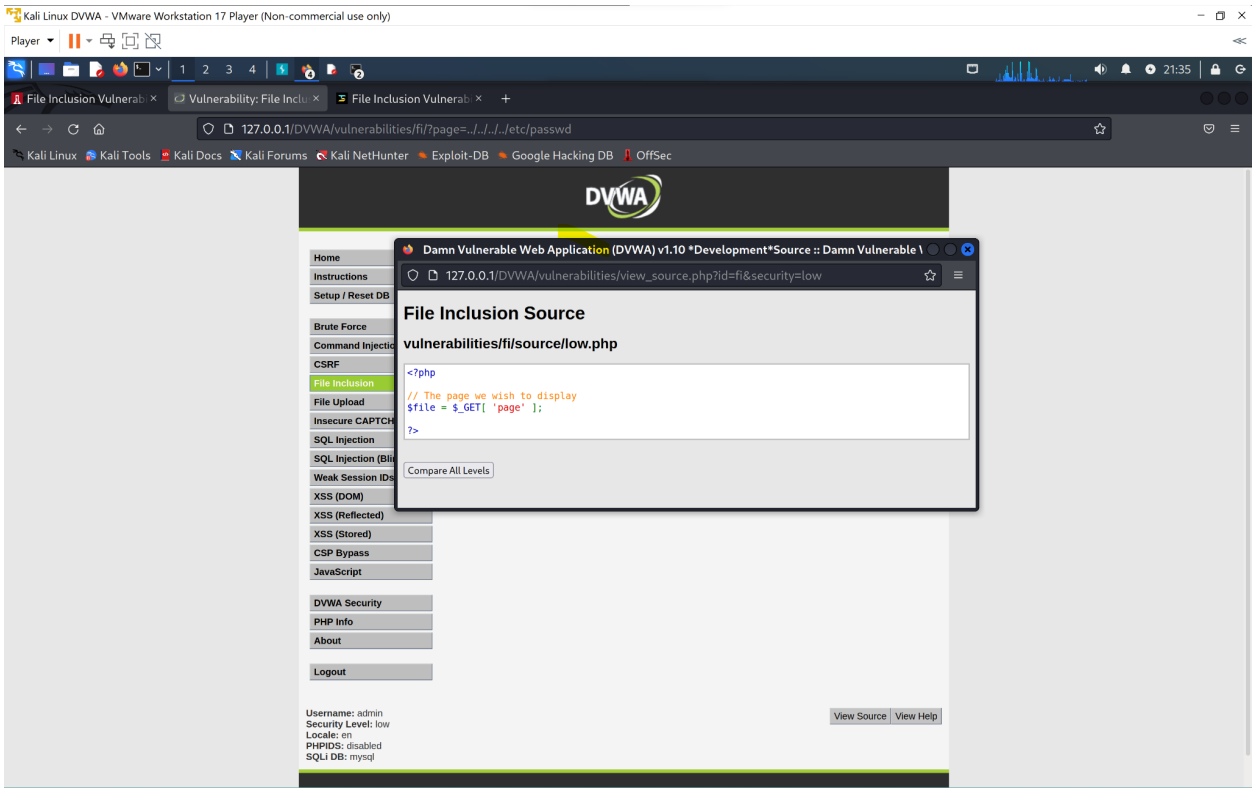
- Adding per-request form keys to the URL and all forms
- Adding a hash such as a session id to all forms
- Verifying and validating the origin of requests that alter server-side information
- Avoiding GET requests for state changing information

File Inclusion: User Information

How does this feature normally work?

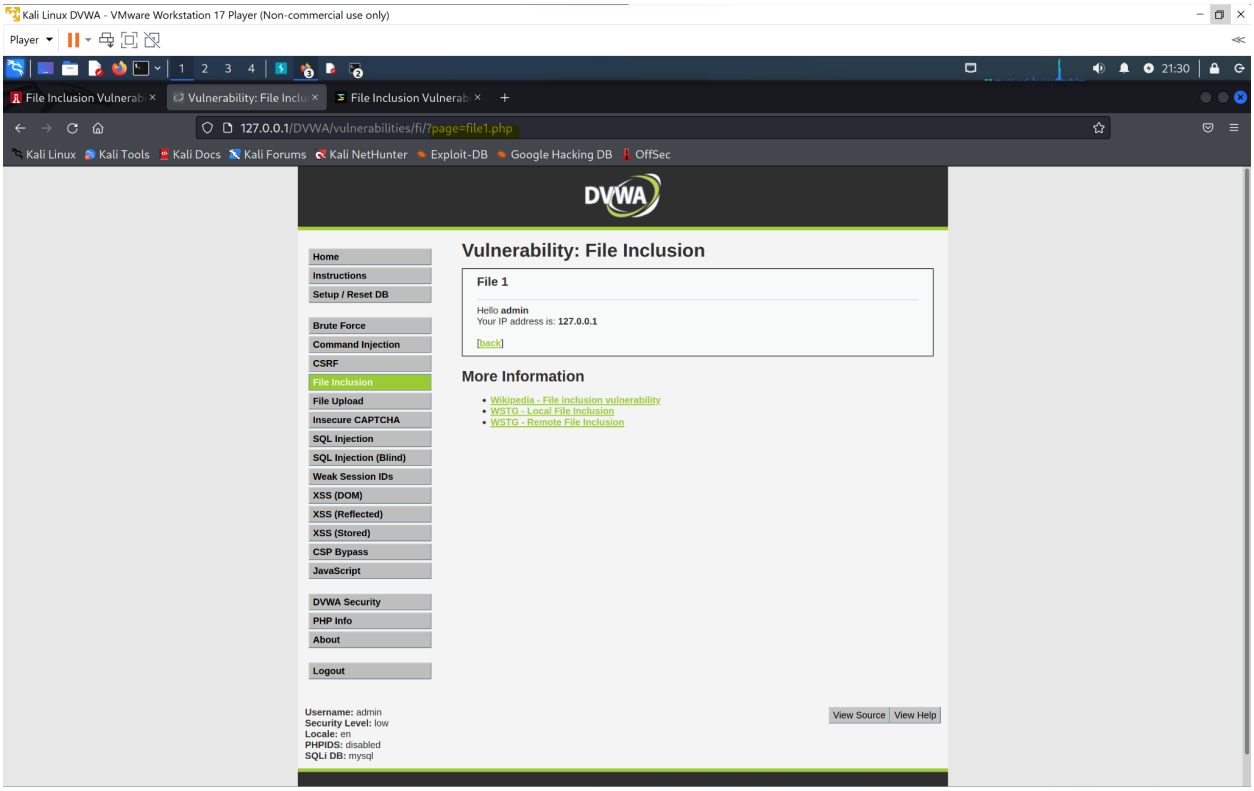
When users visit the File Inclusion page, they are provided with three links: file1.php, file2.php, file3.php. When a user clicks on the link, the server gets the contents of the file and displays them back to the user using a dynamic file retrieval mechanism.



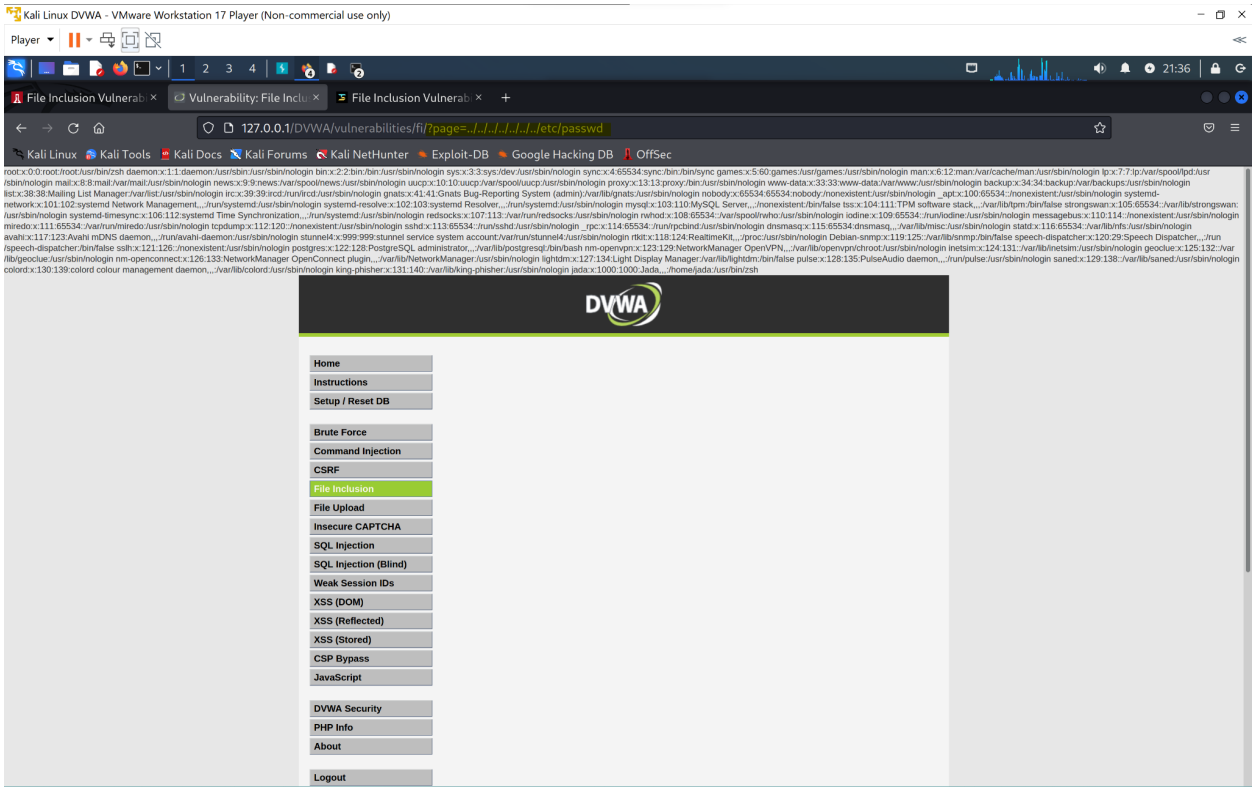


What does it take to exercise the vulnerability?

To exploit a local File Inclusion vulnerability, we can first inspect how the server retrieves the file to view.



After clicking on file1.php, we can see that the file is loaded using the page parameter (file1.php). We can then modify the value of the page parameter to attempt to access sensitive data from the web server. In the following example, we traverse the server's directories to access the contents of /etc/passwrd.



Source: *File Inclusion Vulnerability: (LFI & RFI) Full Guide*[9]

How and why did the feature work differently than normal use?

In its current implementation, the feature assumes that users will not attempt to access files on the server that aren't clearly available. However, if an adversary inspects the source code they will quickly notice that the server will attempt to retrieve any file specified by the page parameter.

Implications and Recommendations

The File Inclusion vulnerability allows an adversary to access the contents of system files. In severe cases, an attacker could also use this vulnerability to execute code on the client-side or server-side.[10]

Preventing a File Inclusion attack can be acheived using the following methods[11]:

- Avoiding passing user-submitted input to filesystem/framework APIs
- Whitelisting allowed files
- Using an identifier for allowed files, and rejecting all requests that do not contain a valid identifier
- Placing limits on which users may upload files, file size, and filename length

Applicable OWASP Top 10: [A01:2021] Broken Access Control; [A04:2021] Insecure Design; [A05:2021] Security Misconfiguration;

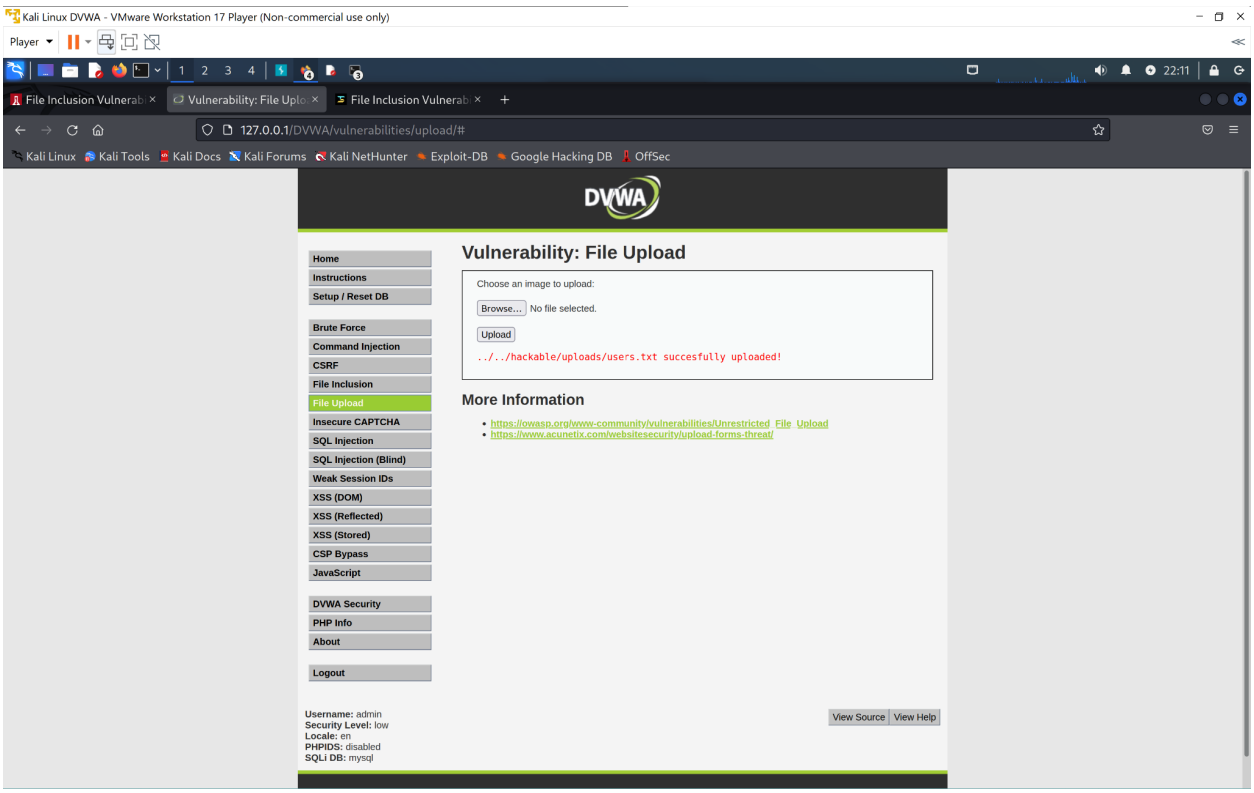
File Upload: Image

How does this feature normally work?

This feature allows users to upload images from their device to the server. If the file is successfully uploaded, users receive the following success message: *../../hackable/uploads/<filename> succesfully uploaded!*

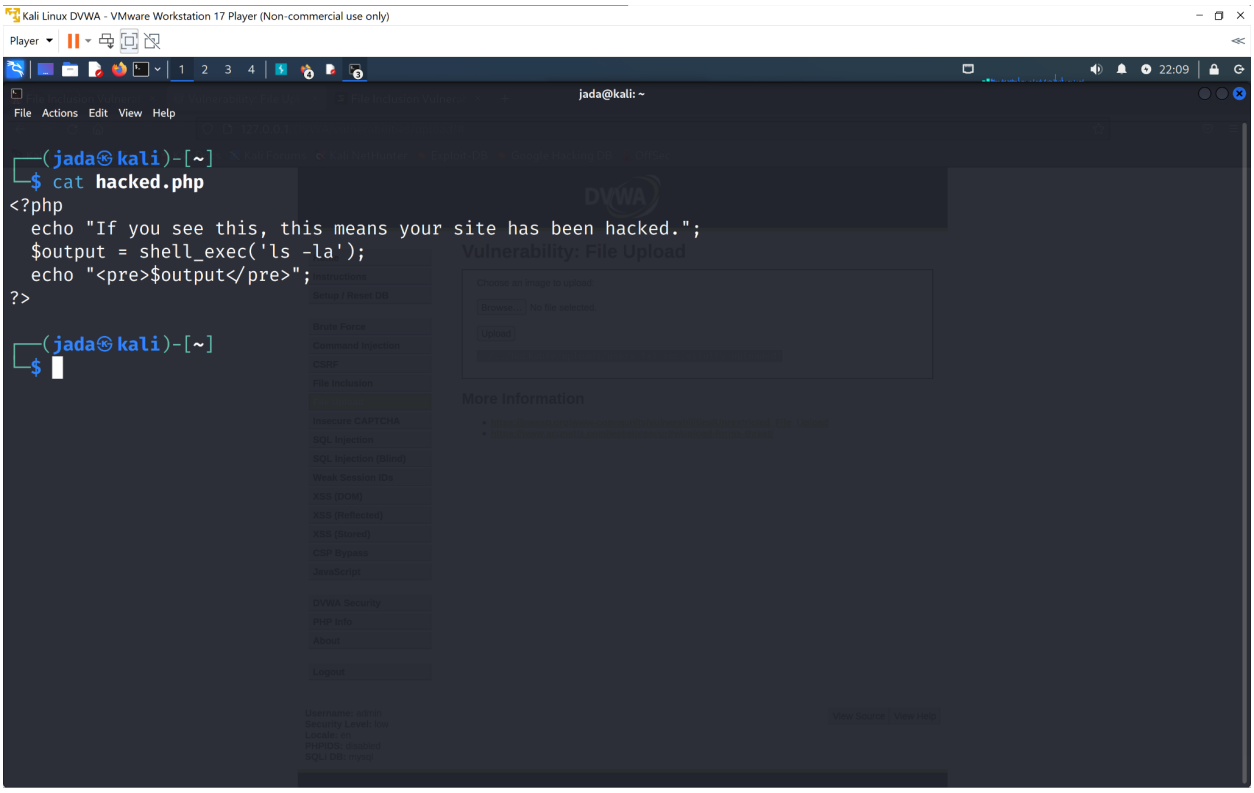
What does it take to exercise the vulnerability?

To test for a File Upload vulnerability, we will first test the behavior of the website when uploading a normal file. In the following figure, we upload the file “users.txt.”

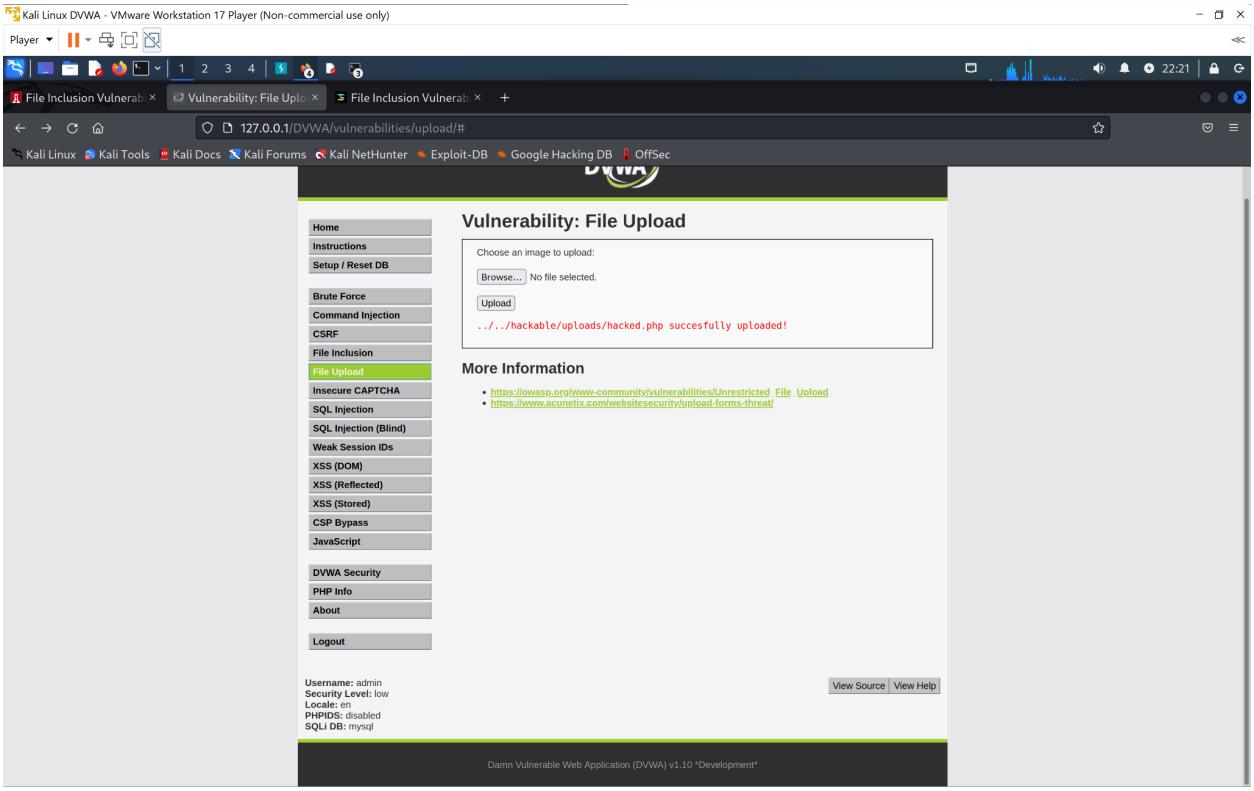


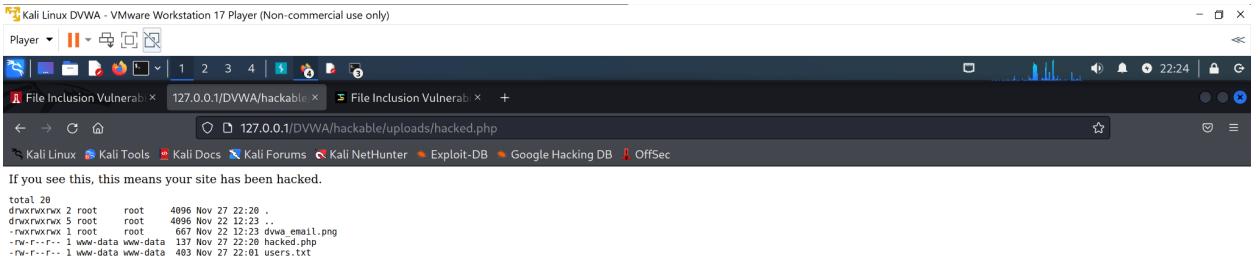
The server informs us that the file was successfully uploaded, despite .txt not being a image file extension.

Next, we can attempt to upload malicious code to the system. In the figure below, “hacked.php” contains code that displays a message then issues the *ls* system command.



Once the file is uploaded, the server provides us with a success message indicating the filepath that contains the malicious code. If we navigate to the provided path (http://127.0.0.1/DVWA/hackable/uploads/hacked.php), we will see that our script executes successfully.





Source: *DVWA File Upload*[12]

How and why did the feature work differently than normal use?

In its current implementation, the server assumes that the user will only upload a valid image file. The file uploaded is not tested prior to being made available to access through the server. Because the server informs the user of the location of the uploaded file, an adversary can very easily chain it with another vulnerability such as Local File Inclusion to attack the system.

Implications and Recommendations

Insecure and unvalidated file upload can allow a malicious actor to overwhelm server resources by uploading a large file as part of a DOS attack. Additionally, since the feature does not validate the file extension an attacker could upload and execute malicious code or commands. As demonstrated in the example, the File Upload vulnerability can be combined with other vulnerabilities to execute malicious code on the server, access secure files, or manipulate server data[13].

Developers can prevent File Upload vulnerabilities by utilizing the following practices:

- Whitelisting allowed extension and performing input validation prior to validating the extensions.
- Validating the file type and not relying on the Content-Type header.
- Restricting file name length, file size, and users authorized to upload files.
- Changing the file name to something generated by the server.

Applicable OWASP Top 10: [A01:2021] Broken Access Control; [A04:2021] Insecure Design; [A05:2021] Security Misconfiguration; [A09:2021] – Security Logging and Monitoring Failures, [A10:2021] Server-Side Request Forgery (SSRF)

SQL Injection: Find User

How does this feature normally work?

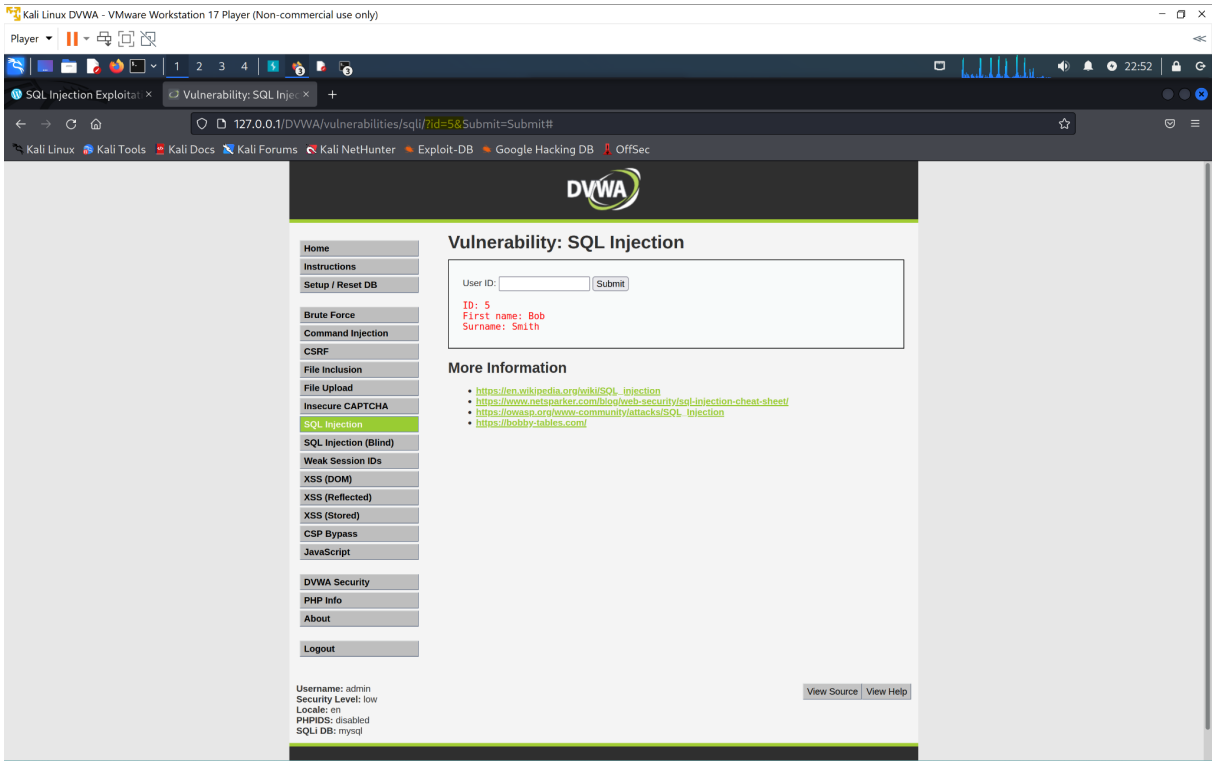
This feature allows users to enter a user ID. The server then parses the SQL databases, searching for the amtching entity. If the user is found, the server responds with the user's first name and surname are displayed. If they aren't found, nothing is displayed.

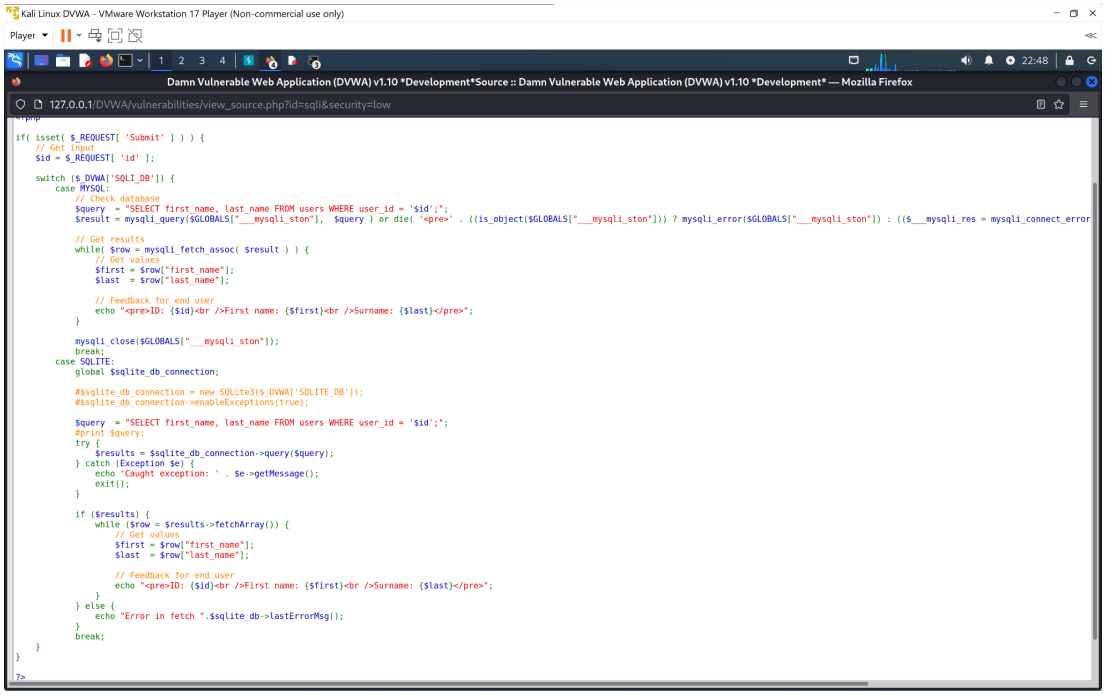
What does it take to exercise the vulnerability?

To exercise this vulnerability, we can analyze the source code and note that the SQL database is queried using the following:

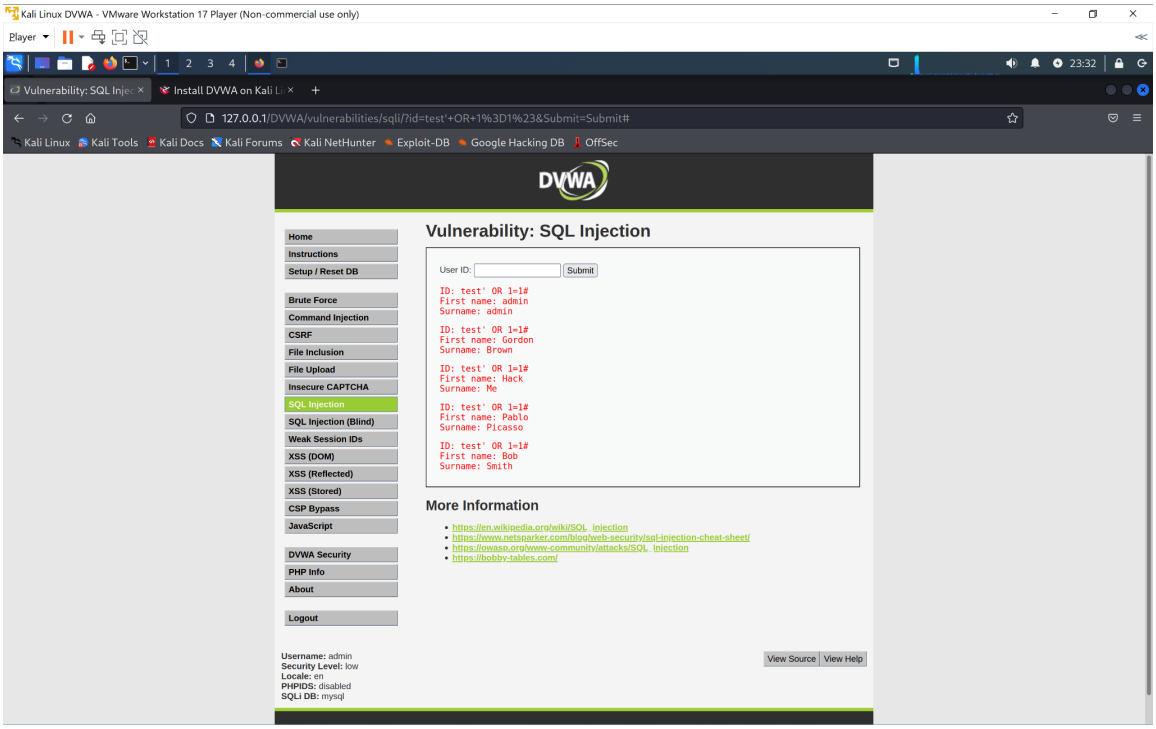
"SELECT first_name, last_name FROM users WHERE user_id = '$id';";

We can also note that when searching for a user, the injectable parameter is contained in the URL's parameter, id.

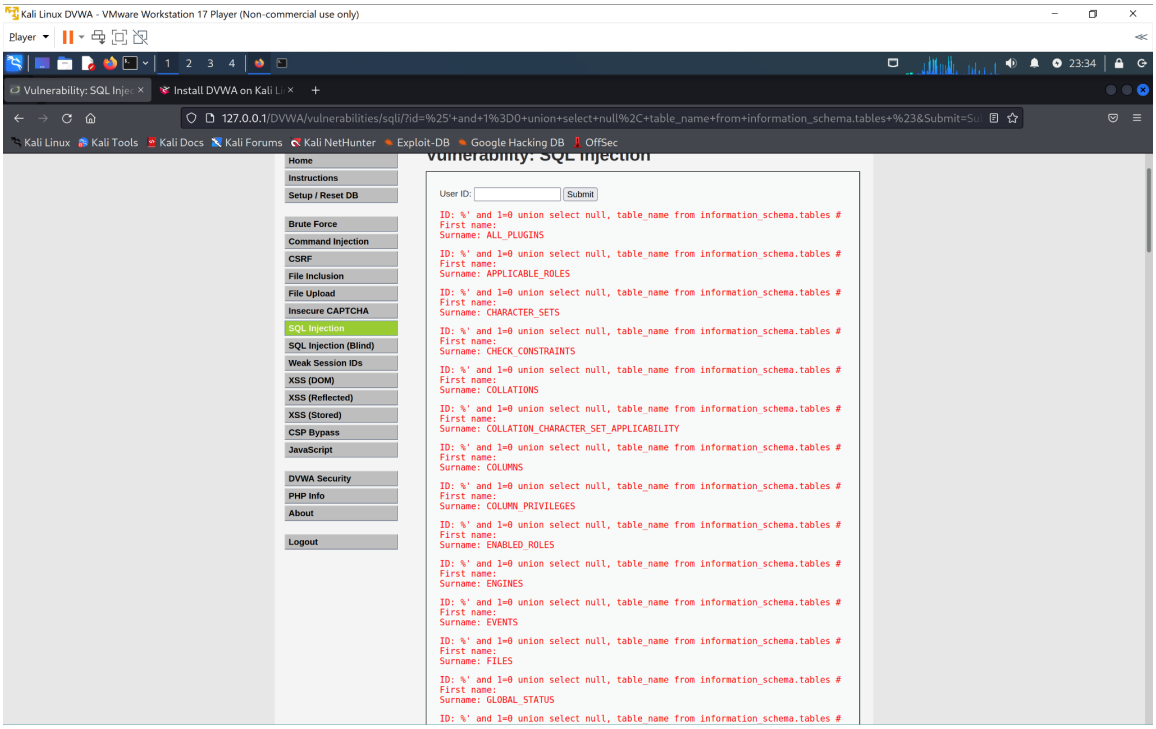




Because the server does not preform input validation on the id input, we can execute modified queries to the database. For example, we can use *test' OR 1=1#* to extract all first names and surnames from the database.



We can also use the vulnerable input field to learn more about the database schema. The following example uses *%' and 1=0 union select null, table_name from information_schema.tables #* to display all tables in the information schema.



Source: *DVWA SQL Injection Exploitation Explained (Step-by-Step)*[14]

How and why did the feature work differently than normal use?

This feature expects users to enter a valid user ID. Since the server accepts raw user input, a malicious actor could inject malicious SQL statements revealing sensitive database information. The server does not validate the database's response so a malicious actor is able to view the results of the executed statement.

Implications and Recommendations

SQL Injection can allow adversaries to spoof identities, access sensitive information stored in the database system, and tamper with, manipulate, and delete existing data on the server. This could lead to the disclosure of confidential information, the compromising of user accounts, or cause repudiation issues when the validity of the information is compromised.

SQL Injection attacks can be prevented using the following techniques[15]:

- Encouraging developers to use language-specific prepared statements with paramerized queries.
- Escaping all user-supplied input.
- Maintaining an allowed list of valid user input.

Applicable OWASP Top 10: [A03:2021] Injection

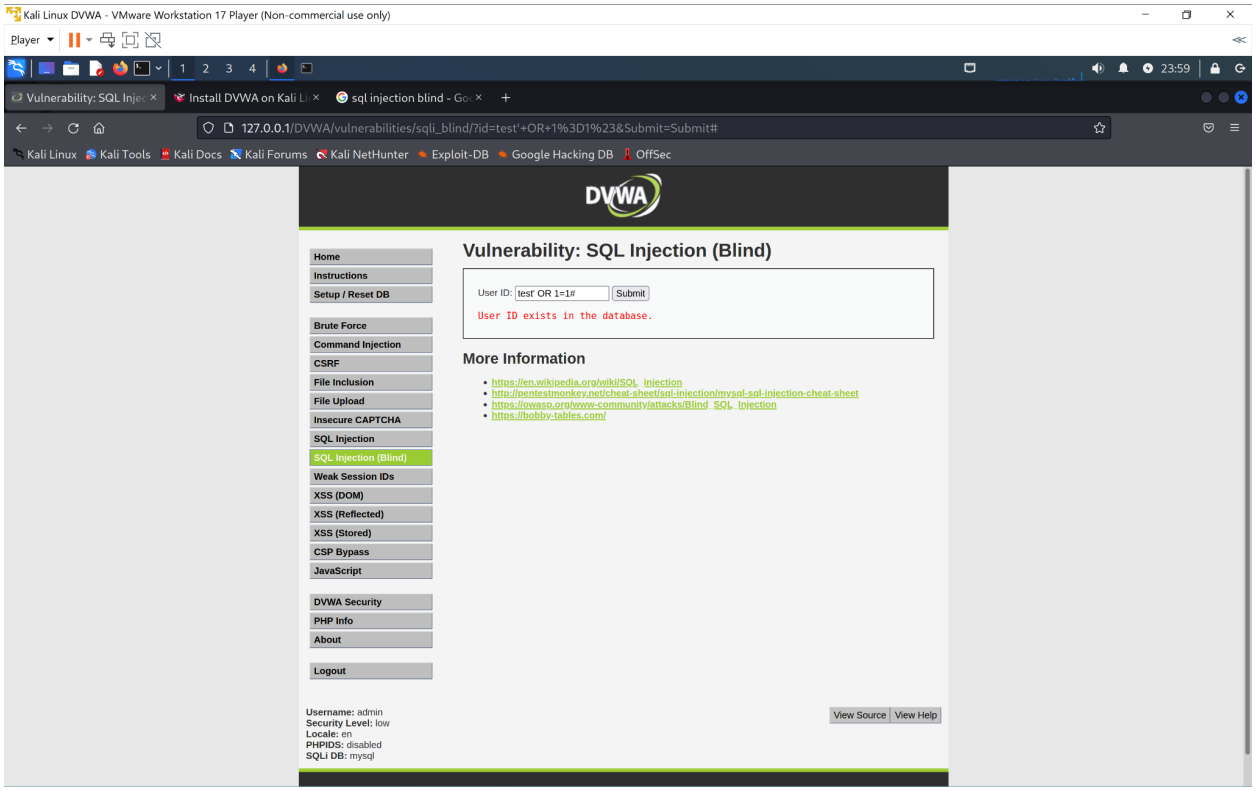
SQL Injection (Blind): Find User

How does this feature normally work?

This feature works similarly to the feature discussed in SQL Injection: Find User. Instead of providing the user's first name and surname, this feature simply confirms the existence of a record in the database. If the ID is found, the user will receive a message indicating that the "User ID exists in the database". Otherwise, the user receives a message indicating that the "User ID is MISSING from the database".

What does it take to exercise the vulnerability?

When exercising the Blind SQL Injection vulnerability, the server does not provide the result of the query. However, we can view the success and failure messages to determine whether or not the provided input was a valid SQL command. For example, using *test' OR 1=1#* generates a success message.



How and why did the feature work differently than normal use?

Similar to the previous SQL Injection vulnerability, this feature expects users to enter a valid user ID and since the server accepts raw user input, a malicious actor could inject malicious SQL statements. The server's success/failure messages confirm whether or not a SQL statement was successfully executed, not necessarily if the user is found.

Implications and Recommendations

At first glance, this vulnerability may not seem as critical as a non-blind SQL Injection. However, an adversary can still access all data stored in the database using a process called Enumeration[16]. Adversaries can use application such as sqlmap to extract information from the database.

Recommendations for preventing a Blind SQL Injection attack are comparable to preventing a normal SQL Injection attack.

Applicable OWASP Top 10: [A03:2021] Injection

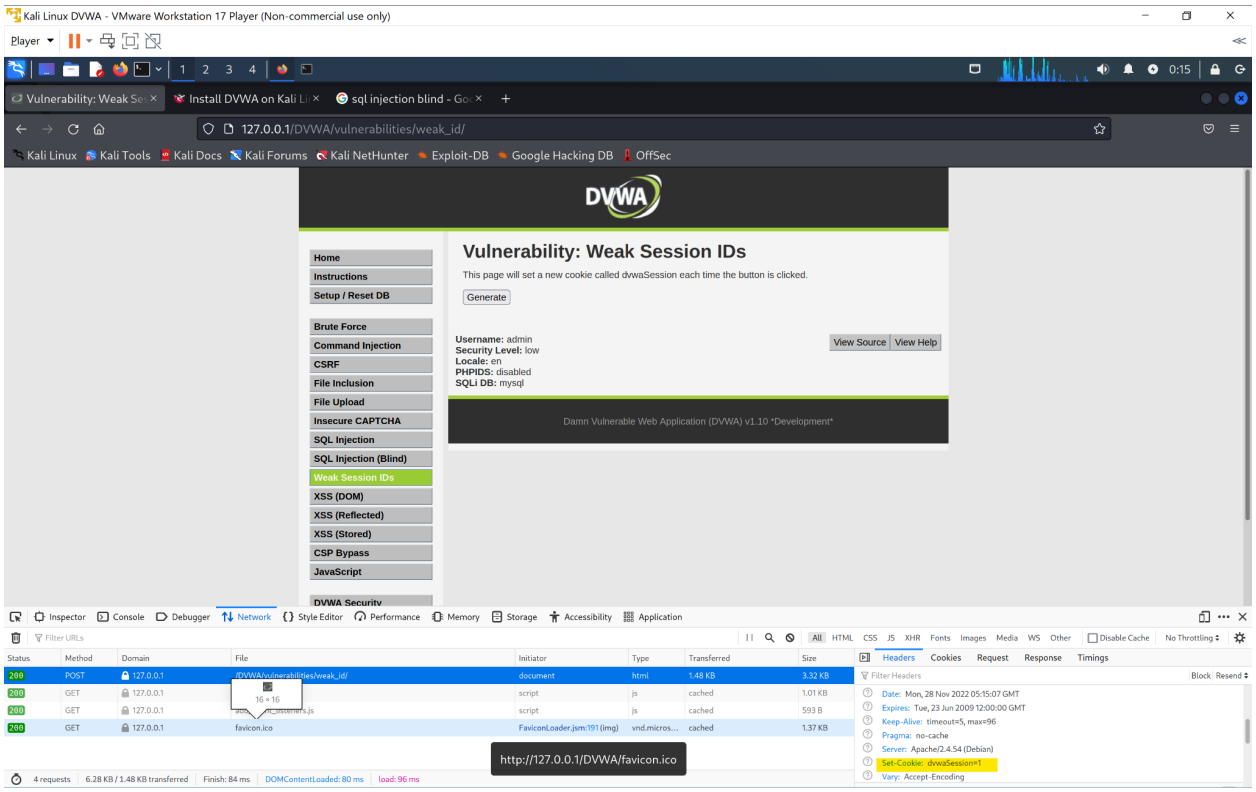
Weak Session IDs: Generate New Session Cookie

How does this feature normally work?

This feature allows the user to set a new cookie called dvwaSession by clicking the button.

What does it take to exercise the vulnerability?

To exercise the Weak Session IDs vulnerability, let's first observe how the feature generates a new session ID. As shown in the below figure, the first dvwaSession Cookie is set to 1.



If we continue to generate new session cookies, we can observe that each new cookies is just incremented by 1 (e.g. 1, 2, 3, etc.). Because the cookie is not randomly generated and easily predictable, this allows us to execute a Man in the Middle Attack.

An adversary can continue to generate session cookies until a gap arises (for example, the current cookie is 3 and the next cookie is 4). The adversary can then safely assume that another user has the missing cookie.[17]

Source: *Weak Session IDs (Low - Security) | DVWA Writeup*

How and why did the feature work differently than normal use?

The session ID serves as access control for an authenticated user in the system. Using the ID, the system is able to determine the user's permissions and privileges. Entropy is a critical component of generating a secure session ID, as unpredictability limits the effectiveness of random guessing attacks. However, since the ID is not randomly generated an adversary can easily guess a valid session ID. In addition, the cookie does not have the Secure attribute, leaving it susceptible to Man in the Middle Attacks.

Implications and Recommendations

By utilizing predictable, insecure session cookies, an attacker can predict a valid session ID and impersonate a user in the system. This session hijacking might not be detected by the user, and can potentially expose their information. Adversaries can also preform actions as a valid user in the system, compromising the system's data integrity.

Preventing exploitation of Weak Session IDs can be accomplished using the following Session Management approaches:

- Avoiding using a descriptive or default name used by the Session ID to prevent revealing the technologies used by the web application.
- Having Session IDs lengths of at least 16 bytes to prevent brute force attacks.
- Utilzing unique and meaningless Session IDs with at least 64 bits of entropy (randomness).
-

Applicable OWASP Top 10: [A01:2021] Broken Access Control; [A05:2021] – Security Misconfiguration; [A07:2021] Identification and Authentication Failures

Cross Site Scripting (DOM, Reflected, Stored): Change Language, Greet User

Note: Due to the similar attack methods for XSS, these have been included in the same section.

How does this feature normally work?

Change Language:

This feature allows users to select a language from a drop-down list. When the user submits the form with the desired language, the page redirects the URL. *http://127.0.0.1/DVWA/vulnerabilities/xss_d/?default=<selected_langauge>.* The user can select from the following options: English, French, Spanish, German.

Greet User:

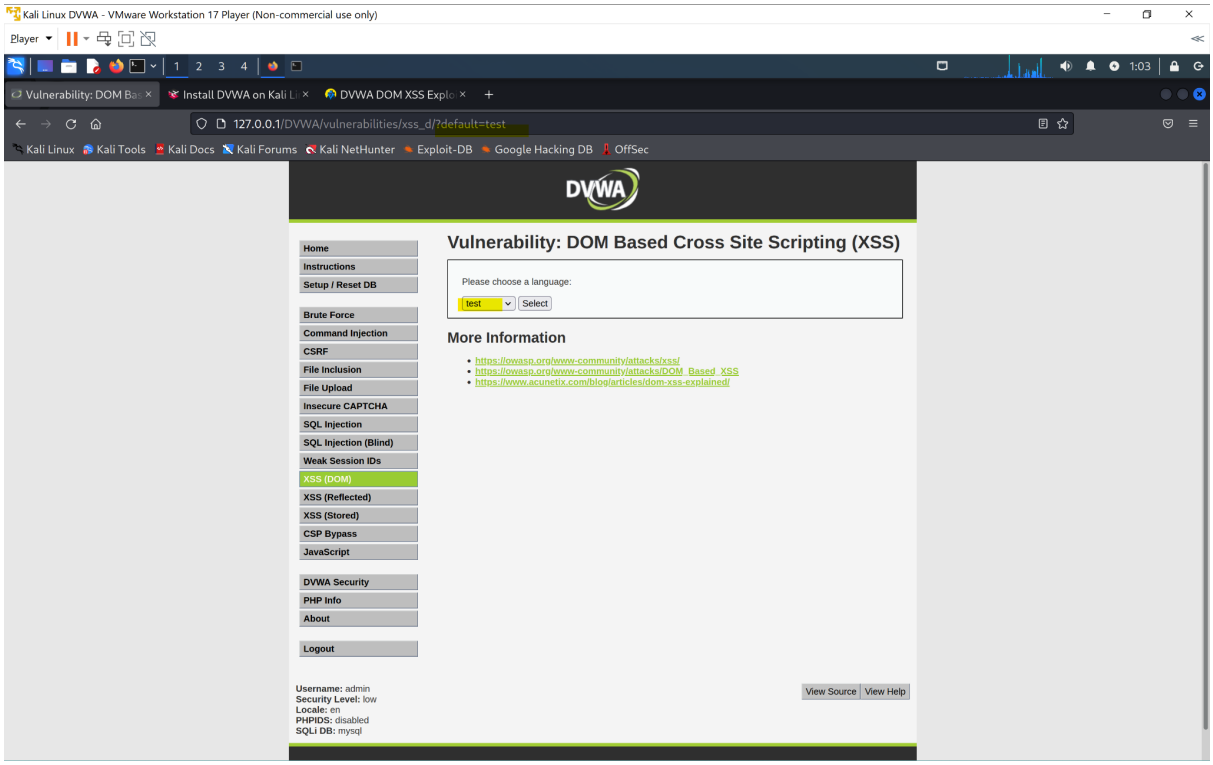
This feature allows users to enter their name or other text content into a text field. Once submitted, the website renders a greeting, *Hello <user_input>*.

Sign Guestbook:

This feature allows users to enter a name and message. The name and message is then stored in the database. Stored messages are reflected back to users when they visit the site.

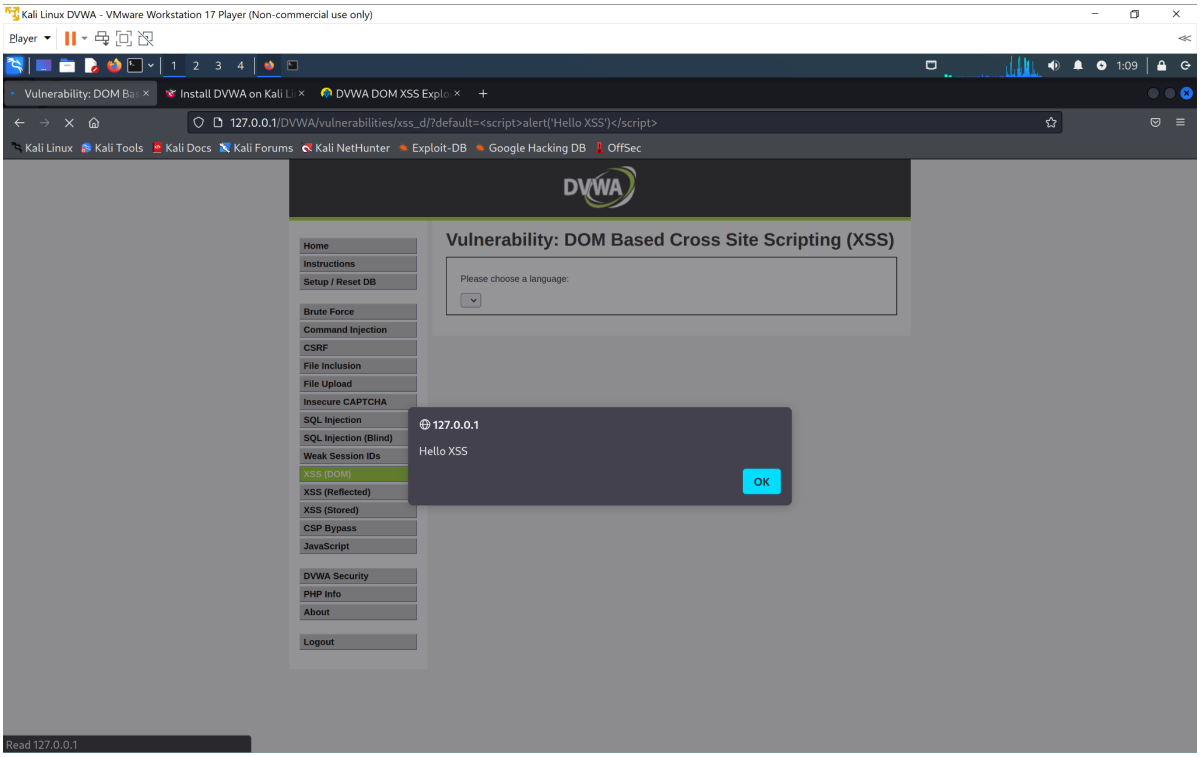
What does it take to exercise the vulnerability?

XSS DOM - Change Language



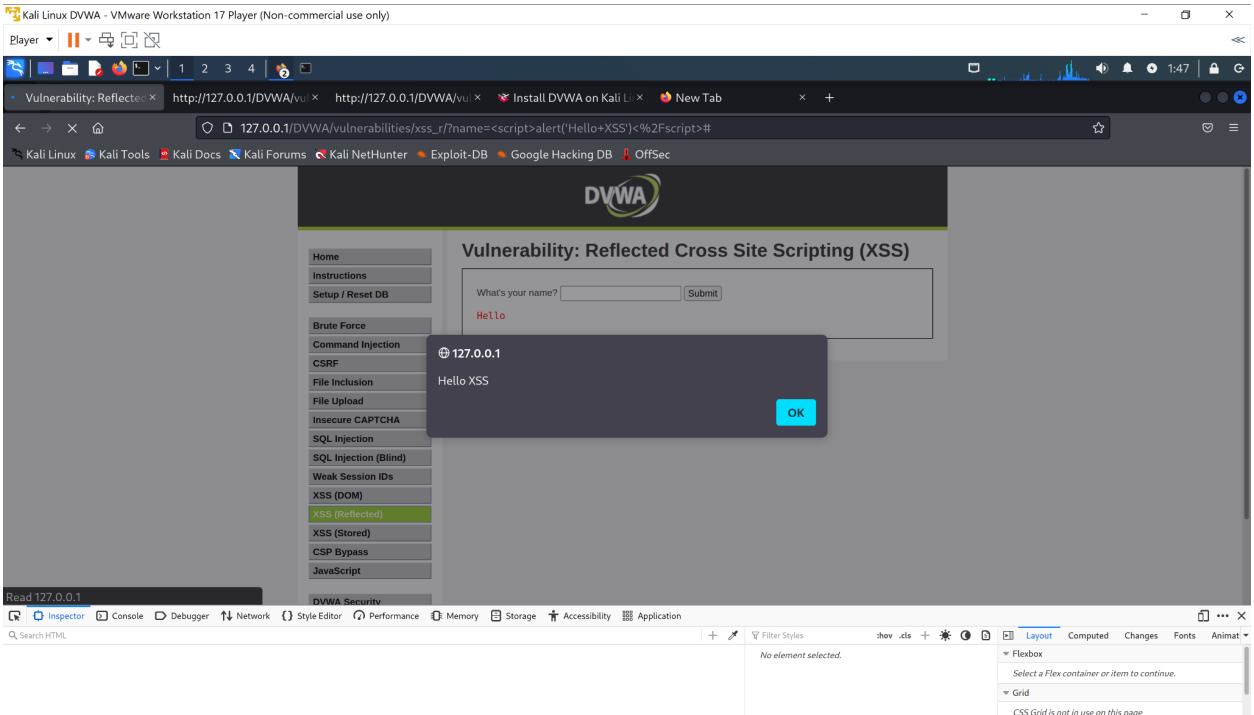
To exercise this vulnerability, we can observe what happens when altering the request parameters.

As shown in the above figure, when the parameter *test* in the URL it is reflected in the webpage's form. We can then attempt to run the JavaScript payload: *<script>alert('Hello XSS')</script>.* Passing the previous script to our URL parameter, we receive the following alert.



XSS Reflected - Greet User:

Similarly, we can pass the same payload to the Greet User feature of the site and receive a popup.



Analyzing the source code, we can see that our malicious script is reflected back to the user without validation.

Reflected XSS Source

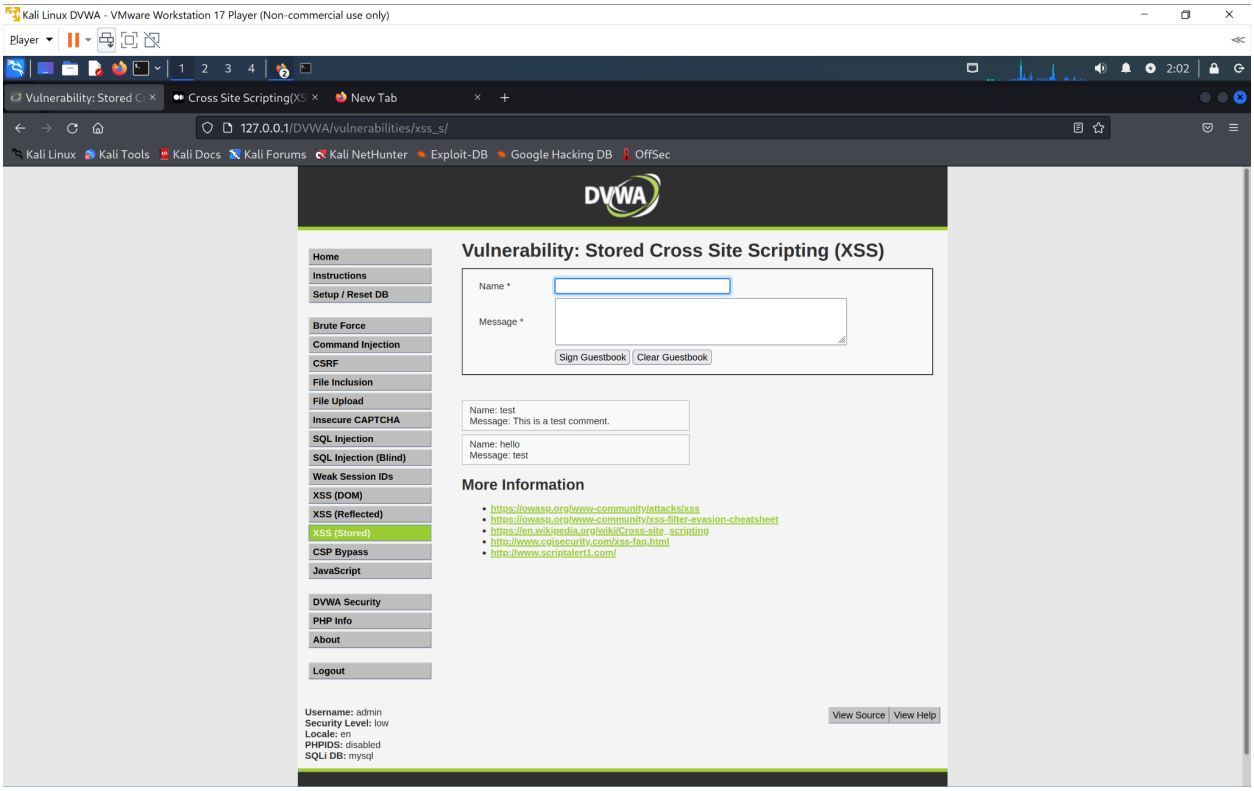
vulnerabilities/xss_r/source/low.php

```php
<?php

header ("X-XSS-Protection: 0");

// Is there any input?
if( array_key_exists( "name", $_GET ) && $_GET[ 'name' ] != NULL ) {
    // Feedback for end user
    echo '<pre>Hello ' . $_GET[ 'name' ] . '</pre>';
}

?>
```

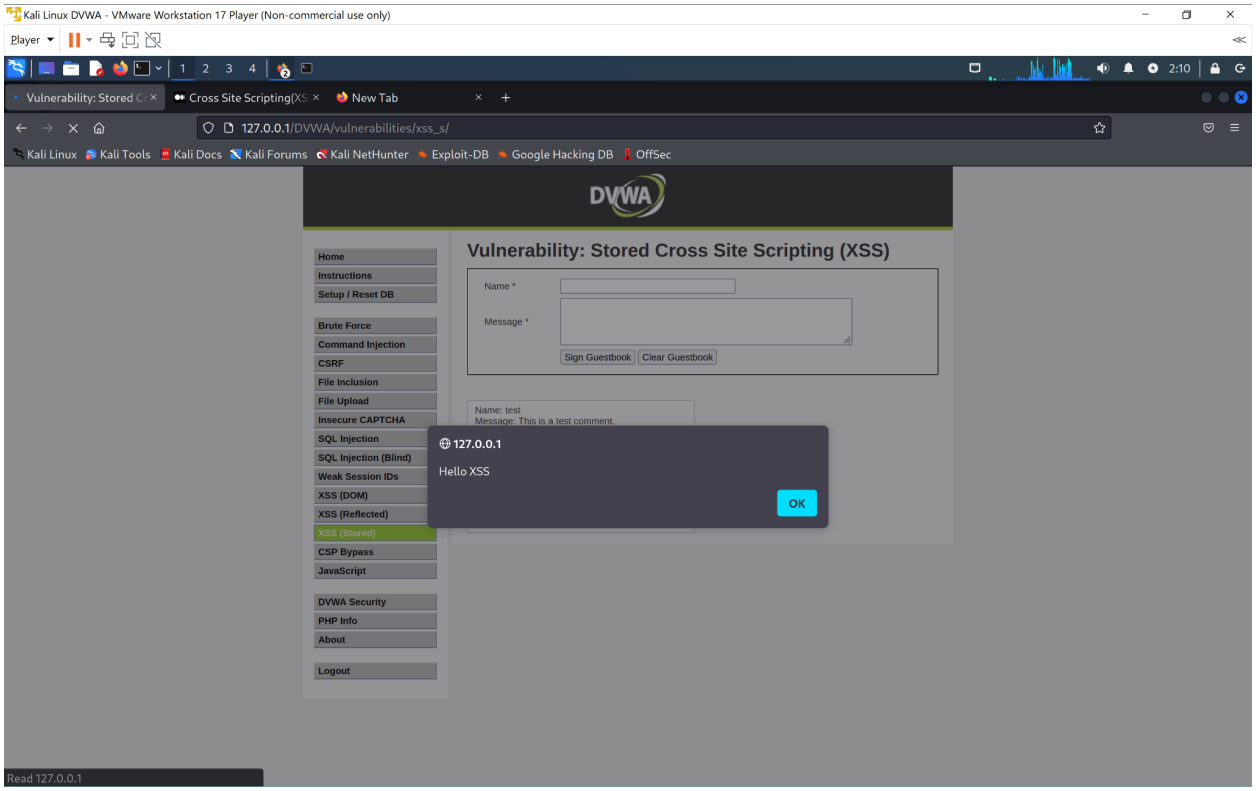
Compare All Levels

XSS Stored - Sign Guestbook

To test for XSS Stored vulnerabilities, we can start by sending a test message to the guestbook. As shown in the screenshot below, the message is stored and displayed back to the user.



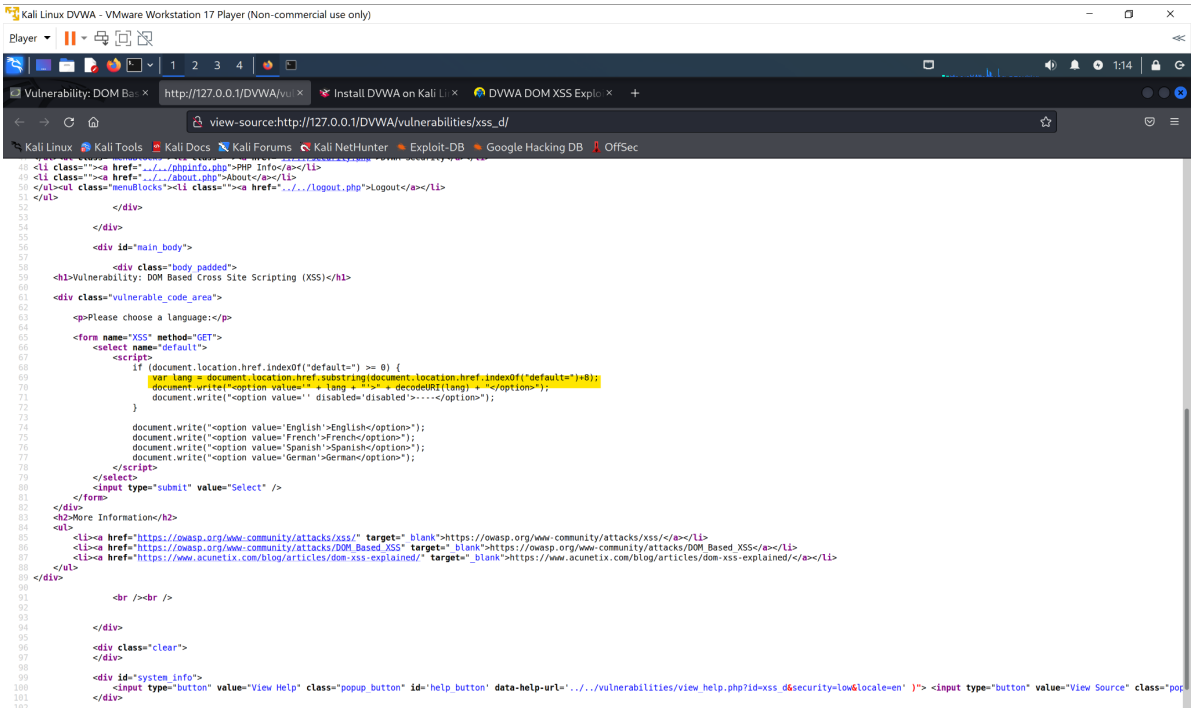
To exploit this vulnerability, we can pass malicious code in our guestbook message. As a result, our script is executed everytime the page reloads.



Source: *XSS: Cross Site Scripting*[18]

How and why did the feature work differently than normal use?

XSS DOM - Change Language



Reviewing this features source code exposes the cause of this vulnerability. As shown in the highlighted section, the URL parameter is used to dynamically update the DOM without validation. This allows a malicious actor to inject an attack into the DOM from the client side.

XSS Reflected - Greet User:

As mentioned in the previous section, once the server receives input from the user it does not undergo any validation or sanitization. As a result, a malicious actor could inject malicious code that will be reflected back to the client.

XSS Stored - Sign Guestbook

Similar to the previous examples, once the server receives input from the user it does not undergo proper validation or sanitization. As a result, a malicious actor could inject malicious code that will be stored in the database. Anytime a user visits the compromised, the script will execute.

Implications and Recommendations

XSS attacks can allow an adversary to manipulate elements of a site on both the client and server side, potentially altering or providing false information to other users of the site. In addition, DOM based XSS attacks can allow users to execute malicious code on the users' devices or compromise their accounts. This can potentially damage a users relationship with the web application's owner.

XSS attacks can be mitigated using the following techniques[19]:

- Sanitizing all untrusted data on the server-side *and* client-side.
- Using safe JavaScript functions such as *document.innerText.*
- If possible, avoiding using user-provided data that affects critical DOM elements such as *document.url*, *document.location*, and *document.referrer.*
- Always use user input in the text context, not as HTML tags.

Applicable OWASP Top 10: [A01:2021] Broken Access Control; [A04:2021] Insecure Design; [A05:2021] – Security Misconfiguration;

Automated Testing

As observed in the previous section, many of the vulnerabilities found in the DVWA can be exploited by manipulating parameters received from user input are sent during the application's HTTP requests. One testing method could target site routes that are

dynamically built using user input. For example, the test behave in the following manner:

For pages on website:
    If site allows user input:
        Test_cmds = [“ls”, “</script>alert('Hello XSS')</script>”, test' OR 1=1#]
        For each test_cmd:
            send_request(page, cur_cmd)
            If response is valid:
                display_reponse()

In the provided example, the testing suite could check all possible routes on the website. If the specified route allows user input, the script can then iterate through list of test commands utilized in the various attack schemes. If the server provides a valid response, the result is sent to the tester in some way for further inspection.

Resources

1. *Damn Vulnerable Web Application*, https://github.com/digininja/DVWA
2. Beton, Danny. 2016. “DVWA Brute Force Tutorial (Low Security) | by Danny Beton.” Medium. https://medium.com/@dannybeton/dvwa-brute-force-tutorial-low-security-463880d53e50#.d1xb364y2.
3. “Blocking Brute Force Attacks.” n.d. OWASP Foundation. Accessed November 26, 2022. https://owasp.org/www-community/controls/Blocking_Brute_Force_Attacks.
4. “Ping - Definition and details.” n.d. Paessler. Accessed November 27, 2022. https://www.paessler.com/it-explained/ping.
5. “How To Perform Command Injection Attacks (DVWA) For Aspiring Hackers! — StackZero.” n.d. InfoSec Write-ups. Accessed November 27, 2022. https://infosecwriteups.com/how-to-perform-command-injection-attacks-dvwa-for-aspiring-hackers-stackzero-c9d521c6f934.
6. Zhong, Weilin. n.d. “Command Injection.” OWASP Foundation. Accessed November 27, 2022. https://owasp.org/www-community/attacks/Command_Injection.
7. Beton, Danny. 2016. “DVWA CSRF Tutorial (Low Security) | by Danny Beton.” Medium. https://medium.com/@dannybeton/dvwa-csrf-tutorial-low-security-bc8474d2f49c.
8. “Cross-Site Request Forgery Prevention.” n.d. OWASP Cheat Sheet Series. Accessed November 27, 2022. https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html.
9. Nair, Rahul R. 2020. “File Inclusion Vulnerability: (LFI & RFI) Full Guide.” TechSphinx. https://techsphinx.com/hacking/file-inclusion-vulnerability-full-guide/.

10. Keary, Eoin. n.d. “WSTG - v4.2.” WSTG - v4.2 | OWASP Foundation. Accessed November 27, 2022. https://owasp.org/www-project-web-security-testing-guide/v42/4-Web_Application_Security_Testing/07-Input_Validation_Testing/11.1-Testing_for_Local_File_Inclusion.
11. “File Upload.” n.d. OWASP Cheat Sheet Series. Accessed November 27, 2022. https://cheatsheetseries.owasp.org/cheatsheets/File_Upload_Cheat_Sheet.html.
12. Maurya, Deepak K. n.d. “DVWA File Upload | (Bypass All Security).” Ethicalhacs.com. Accessed November 27, 2022. https://ethicalhacs.com/dvwa-file-upload/.
13. “File Upload.” n.d. OWASP Cheat Sheet Series. Accessed November 27, 2022. https://cheatsheetseries.owasp.org/cheatsheets/File_Upload_Cheat_Sheet.html.
14. Gidraph, Tonny. n.d. “DVWA SQL Injection Exploitation Explained (Step-by-Step).” GoLinuxCloud. Accessed November 27, 2022. https://www.golinuxcloud.com/dvwa-sql-injection/#Step_2_Basic_Injection.
15. “SQL Injection Prevention.” n.d. OWASP Cheat Sheet Series. Accessed November 27, 2022. https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html.
16. “DVWA SQL Blind Injection.” 2022. tkcyber.com. https://tkcyber.com/index.php/2022/04/10/dvwa-sql-blind-injection/#DVWA-SQL-Injection-Blind-Vulnerabilities-.
17. Maheshwari, Anuj. n.d. “Weak Session IDs (Low - Security) | DVWA Writeup | by Anuj Maheshwari.” System Weakness. Accessed November 28, 2022. https://systemweakness.com/weak-session-ids-low-security-dvwa-writeup-7b48c5cbb38f.
18. “XSS: Cross Site Scripting.” n.d. Whisper Lab. Accessed November 28, 2022. https://whisperlab.org/introduction-to-hacking/lectures/web-exploitation.
19. “How To Prevent DOM-based Cross-site Scripting.” 2019. Acunetix. https://www.acunetix.com/blog/web-security-zone/how-to-prevent-dom-based-cross-site-scripting/.