Vulnerability Analysis of Content Management Systems to SQL Injection Using SQLMAP

Olajide Ojagbule
Dept. of Information Technology
Georgia Southern University
Statesboro, GA USA
oo00974@georgiasouthern.edu

Hayden Wimmer
Dept. of Information Technology
Georgia Southern University
Statesboro, GA USA
hwimmer@georgiasouthern.edu

Rami J. Haddad
Dept. of Electrical & Computer Engineering
Georgia Southern University
Statesboro, GA USA
rhaddad@georgiasouthern.edu

There are over 1 billion websites today, and most of them are designed using content management systems. Cybersecurity is one of the most discussed topics when it comes to a web application and protecting the confidentiality, integrity of data has become paramount. SQLi is one of the most commonly used techniques that hackers use to exploit a security vulnerability in a web application. In this paper, we compared SQLi vulnerabilities found on the three most commonly used content management systems using a vulnerability scanner called Nikto, then SQLMAP for penetration testing. This was carried on default WordPress, Drupal and Joomla website pages installed on a LAMP server (localhost). Results showed that each of the content management systems was not susceptible to SQLi attacks but gave warnings about other vulnerabilities that could be exploited. Also, we suggested practices that could be implemented to prevent SQL injections.

Keywords-- SQLi, web applications, vulnerability, SQL injection

I. Introduction

This paper aims to compare the SQL injection vulnerabilities found in the three most commonly used content management systems, why they are susceptible and ways on how to mitigate them. According to online statistics on the usage of content management systems, WordPress is found to be the most used followed by Drupal and Joomla. SQL injection vulnerabilities are among the most common vulnerabilities known to hackers and consistently appear at the top of the list of security vulnerability for the past couple of years. A computer security firm Imperva was quoted to have mentioned that SQL injection has one of the most severe vulnerabilities in human history due to the damage that can be caused using this technique. It is said to have accounted for over 83 percent of data breaches between the years 2005 to 2011 [1].

In this paper, we used Kali Linux as it is very versatile and user-friendly operating system that is used to carry out seamless and effective penetration testing and security auditing tasks. This fully open source operating system makes it quite flexible for users to carry out exploits intimately. LAMP stack was installed as a web platform for running the content management systems. The three content management systems used in this paper were all installed using default configuration with no extra security feature implemented or plugin installed. Nikto, a web scanner that tests web servers, web applications for dangerous files/CGIs, outdated server software and other problems was used to scan each system for various types of problems ranging from poorly configured files to SQL injection vulnerabilities while SQLMAP which automates the process of exploiting SQLi was used as the penetration tool. After carrying out the penetration test, the content management systems were all found to be protected by a type of Intrusion Prevention System (IPS) or Web application firewall (WAF) which made them not vulnerable to SQL injection attacks.

II. Literature Review

SQL Injection can be used in different ways to cause serious damage to a web application database. By exploiting an SQLi vulnerability, an attacker can use it to bypass both authentication and authorization mechanisms of a web application to retrieve vital information from the database. This can be very dangerous because once an attacker gains access to a backend database, the attacker would be able to view the contents of the database, modify existing records and go as far as deleting stored information. Several types of hackers exist, but one type poses explicitly a significant threat that is black hat hackers. This type of hackers uses such attacks to steal data and vital information for their financial gain. That is why it is essential for web applications to be properly tested for all kind of security vulnerabilities. Two important characteristics of SQL injection attacks are the injection mechanism and different kind of attacks that can be carried out. Below are some examples of SQL injection attacks.

A. In-band SQLi

This is arguably the easiest to exploit of all SQL Injection attacks. In this type of SQL injection attack, an attack is launched when an attacker posts malicious codes into a web application and gathers results from the database using the same communication channel, i.e., the input (malicious codes) and the output (database results) uses the same communication channel. The results are usually returned to the attacker's screen. Error-based SQLi is an example of an In-band SQLi

B. Error-based SQLi

This SQL Injection technique relies on error messages returned by the database server after some reserved characters or malicious codes are posted on a login form of a web application. The error messages give the user/attacker some information about the structure of the database. Error messages features are used for testing during the phase of application development and should be disabled or logged to a file with restricted access when such applications Go-live.

C. Double Query-based SQLi

For this technique, the attacker combines two different queries into a single query with the sole purpose of confusing the backend database causing it to return error messages. The responses from the backend database usually contain information the attacker is trying to extract.

D. Blind SQLi

Blind SQL Injection, requires an attacker to create a well-constructed logical query to be injected into a web application to observe the way the backend database reacts to the input. It takes a longer time to exploit because the parameters are injected blindly into the application, and the results from the database are not displayed to the attackers, thereby getting its name Blind SQL injection because the results are not dumped on the screen of the user or visible to the attacker. There are two types of Blind SQL Injection, the Boolean-based blind SQLi, and time-based Blind SQLi.

1. *Boolean-based Blind SQLi:* Every technique is based on two different inputs either TRUE or FALSE. In this technique for every query sent to the database, it returns different results or outputs for either a TRUE or FALSE query result.

2. *Time-based Blind SQLi:* This technique relies on sending SQL query to the database which causes a delay in the amount of time it takes the database to respond. To create this delay in time, the attacker must build a proper query to force the server to work. The response time indicates to the attacker whether the result of the query is TRUE or FALSE.

Detecting SQL injections and finding ways to mitigate them have become paramount amongst web developers and security analyst due to the growing number of attacks. According to the paper proposed by [2], a solution was proposed that predict unknown future occurrences of SQL injection attacks on web applications by using an expectation-based solution. The expectation is intuitively the long-run average value of repetitions of a random variable and in probability theory. This was achieved by Calculating the occurrence probability of the SQL injection attack on special characters dataset and typical dataset respectively, and in the second phase detect SQL injection attack based on expectation with the computed occurrence probability. The results from this research showed high detection of SQL injections, preventing future occurrences of SQL injections.

Alserhani et al. [3] also discussed the use of Alerts correlation techniques to provide intelligent and stateful detection methodologies. Majority of the systems proposed are based on rule mechanisms which are time-consuming and prone to errors. They proposed an improved Event alert correlation system that identifies the shortcomings of current alert correlation techniques. The MARS (Motivation, Abilities, Role perception & Situational factors) model was used to generate rules to correlate high-level alerts called Meta Alerts which are derived from the "provides/requires" model. In this paper, SQLI attack was used to measure the accuracy and performance of the MARS tool. The tool identified higher attack plans and reduced false positive results by using vulnerability knowledge from the results of the test carried out.

Techniques are available to deal with QLI attacks including black box testing where vulnerability scanners are used to detect the SQLI inside the web application. Existing vulnerability scanners do provide full coverage and often provide fast positives. Various vulnerability scanners are used in web applications before live deployment, but most of them provide lots of false positives. *Singh and Roy [4]* presented a Network-based Vulnerability scanner which consisted of 3 major stages, scanning the web application Attack Simulation and, creating an ad-hoc network. This method was found to identify far more vulnerabilities with less false positives over a shorter period compared to other widely used and popular web applications vulnerability scanners. Automated SQL injection can be used to exploit web applications and compromise databases. SQLMAP is a penetration tool used to automate the process of detecting and exploiting SQL injection vulnerabilities to take control of database servers. All poorly designed web applications are susceptible to SQLi attacks; hence all web developers are encouraged to follow standard security principles that prevent SQLi on their sites as mentioned by [5].

Cross-site scripting (XSS) is another form of attack carried out on vulnerable web applications. This occurs on the client side of a web application where an attacker can execute malicious scripts into a web application or website. XSS is amongst the most rampant of web application vulnerabilities and occurs when a web application uses input from a user within the output generated without validating and encoding it. Kieyzun et al. [6] in their paper presented a technique and an automated tool that can be used for detecting security vulnerabilities in Web applications. The tool focused on finding the two most common types of web application vulnerabilities which are SQL injections and XSS (cross-site scripting) attacks. Unlike other previously proposed approaches, their technique works on existing codes which are not modified and creating concrete inputs that expose vulnerability. It analyses the internal components of an application to find vulnerable codes. An automated tool was created called Ardilla which creates SQLi and XSS attacks in PHP/MySQL applications. This tool generates sample inputs, changes the state of the inputs to create genuine exploits and tracks taints throughout the execution process. This tool was used to evaluate five PHP/MySQL applications and found 68 previously unknown vulnerabilities (23 SQLI, 33 first-order XSS, and 12 second-order XSS) with few false positives. This speaks on another technique and automated tool for detecting vulnerabilities (SQL Injection and XSS) in web applications

It is still an ongoing discussion among IT experts on what the best way of detecting SQL injection vulnerabilities in web services and applications. Antunes and Vieira [7] in their paper made a comparison between two well-known techniques the penetration testing and static code analysis often used for the detection of security vulnerabilities. This was done to evaluate how effective they are in detection SQL injections vulnerabilities in web services. For their study, 8 Web services providing a total of 25 operations, 3 well known commercial penetration-testing tools (HP WebInspect, IBM Rational AppScan and Acunetix Web Vulnerability Scanner) and Three vastly used static code analyzers that detect vulnerabilities in Java applications (FindBugs, YASCA, and IntelliJ IDEA).

The results from penetration testing tools and static code analysis tools were analyzed separately and then compared to understand the strengths and weaknesses of each approach better. Results showed that the coverage of static code analysis tools is typically much higher than of penetration testing tools.

According to a paper by [8] traditional penetration test methods have been found to be insufficient to test SQL injection vulnerabilities (SQLIVs) in web applications, so a test method was proposed called SMART which automatically scans and test SQL Injection Vulnerabilities in web applications.

SMART tests each input parameter of web applications, matches the SQL queries generated by both original HTTP request and injected HTTP request, and determines whether it has SQL injection vulnerability. It uses a structure matching validation mechanism to determine whether SQLIVs exist. In comparison with traditional security scanners, SMART proved to be more efficient. It used less time, found much more SQLIVs, and had an acceptable false positives rate.

Much work has gone into the improvement of systems, techniques, and tools for detecting SQL injections to make web applications more secure making it difficult for hackers to attack. An approach which uses attack signatures and interface monitoring to increase the visibility of the penetration testing process, without needing to access web service's framework. This was proposed in a paper by *[9]*, a prototype tool that targeted the detection of SQL injection vulnerabilities in SOAP web services was developed to test the feasibility of this approach. The prototype included an attack-load generation module able to analyze the web service and generate attacks containing signatures. During the attack process, the database traffic was monitored, and the signatures detected were reported as vulnerabilities. The evaluation of this experiment was carried out on 21 web services provided by a benchmark for vulnerability detection tools. After comprehensive testing, results showed that this approach achieved much higher detection coverage than three other commercial penetration testing tools (HP WebInspect, IBM Rational AppScan, and Acunetix Web Vulnerability Scanner) while avoiding false positives.

III. Methodology

The three content management systems to be used in this paper for penetration testing are WordPress, Drupal, and Joomla. A content management system is a software application used for creation and management of digital content and is typically used for enterprise content management (ECM) and web content management (WCM). Many notable big businesses and brands use one of these 3 CMS for their websites. For this to be achieved, a list of components will be required which are a virtual machine with a Linux environment, LAMP stack which would comprise of the web server and database to be used and the Penetration testing tool.

For the system setup, we will start by installing Kali Linux OS on a Virtual Machine; the VMware workstation 12 platform will be used for virtualization. Kali Linux is very versatile and user-friendly, it is used to carry out seamless and effective penetration testing and security auditing tasks on systems, networks, and web applications. Due to its open source nature, it is quite flexible for users to carry out exploits. After installing Kali Linux, we will then go on to installing the LAMP stack. LAMP stack is a popular open source web platform used to run dynamic websites and web applications. It includes Linux which has already been installed, Apache as the Web server which would host the content management systems, MySQL as the relational database management system and PHP or Python as the object-oriented scripting language. It is a highly recommended platform for development and deployment of high-performance web applications.

Next step will be to install the content management systems one after the other starting with WordPress, followed by Drupal and Joomla. Each system would have a database table created in MySQL with admin accounts and various other users with certain privileges. Once all these components have been set up, we will then move on to the Penetration testing phase. SQLMAP will be the penetration testing tool used in this paper to exploit possible vulnerabilities found. This tool automates the process of detecting and exploiting SQLi vulnerabilities by taking over database servers. SQL injection is one of the most common techniques utilized by hackers to take information from database servers of web applications. This attack is carried out by injecting malicious codes into a poorly designed application which is then passed to the database, the malicious data then produces query actions that should never have been executed.

The final phase which is the penetration testing, we will take a step by step approach that will give us the maximum amount of information required about the security posture of the system. In this case, we suggest breaking down the penetration testing phase into the following steps below:

- Information gathering: This step involves the gathering of information about the system and server to be targeted by the hacker.
  - Scanning: For this phase, a web scanner called Nikto will be used. Nikto is an Open Source web server scanner which performs comprehensive tests on web servers for multiple items, which includes checking for potentially dangerous files, outdated versions and version specific problems on servers. The web application is scanned for vulnerabilities, and all vulnerabilities found are documented.
- Exploitation: In this step, we will exploit some of the vulnerabilities found to demonstrate the threats they pose in the real world. It is in this phase when we use the penetration testing tool called SQLMAP.
- Mitigation: This step would comprise of ways of removing vulnerabilities found and how to prevent them from being exploited.

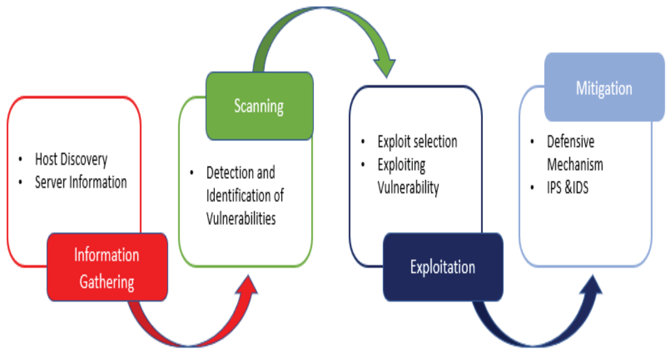


Fig 1. Penetration Testing Methodology.

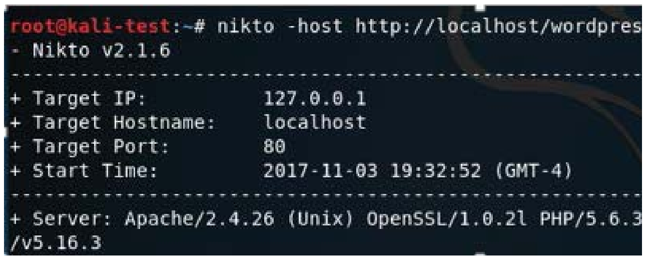
The test, illustrated in Fig. 1, would be carried out on each of the content management system (WordPress, Drupal, and Joomla), all vulnerabilities will be recorded and compared.

IV. Results

As explained in the methodology section, the penetration testing was divided into three stages; Scanning, Exploitation, and Mitigation.

A. Scanning

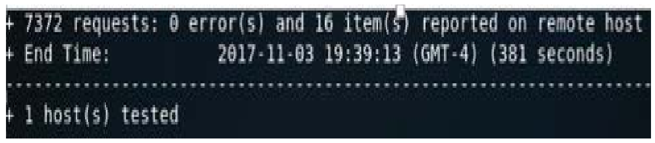
1. WordPress site: Using Nitko to scan the target http://localhost/wordpress, it provided the basic information about the target host server as illustrated in Fig. 2.



```
root@kali-test:~# nikto -host http://localhost/wordpres
- Nikto v2.1.6
---------------------------------------------------------------------------
+ Target IP:          127.0.0.1
+ Target Hostname:    localhost
+ Target Port:        80
+ Start Time:         2017-11-03 19:32:52 (GMT-4)
---------------------------------------------------------------------------
+ Server: Apache/2.4.26 (Unix) OpenSSL/1.0.2l PHP/5.6.3
/v5.16.3
```

Fig. 2. Command to gather information about the target (WordPress)

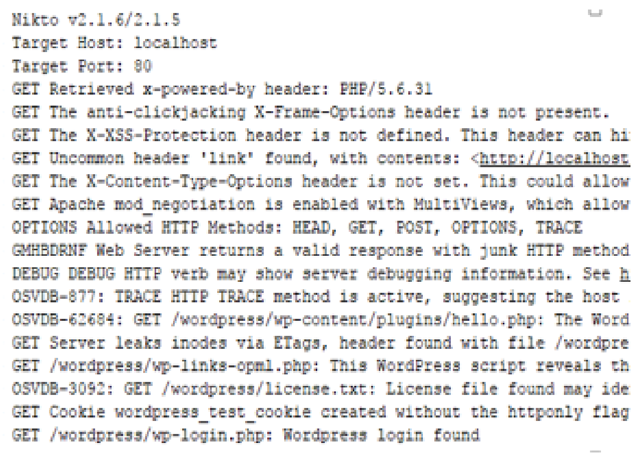
After successfully scanning the site, the results are illustrated in Fig. 3.



```
+ 7372 requests: 0 error(s) and 16 item(s) reported on remote host
+ End Time:           2017-11-03 19:39:13 (GMT-4) (381 seconds)
---------------------------------------------------------------------------
+ 1 host(s) tested
```

Fig. 3. Scan results

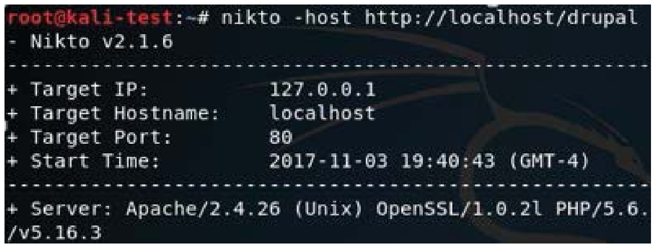
These results show that 7372 files were scanned for vulnerabilities and 16 items were reported to be susceptible to one form of attack. In Fig. 4, the detailed list of the scan result is provided. It shows various config files, pages that might be vulnerable to an attack.



```
Nikto v2.1.6/2.1.5
Target Host: localhost
Target Port: 80
GET Retrieved x-powered-by header: PHP/5.6.31
GET The anti-clickjacking X-Frame-Options header is not present.
GET The X-XSS-Protection header is not defined. This header can hi
GET Uncommon header 'link' found, with contents: <http://localhost
GET The X-Content-Type-Options header is not set. This could allow
GET Apache mod_negotiation is enabled with MultiViews, which allow
OPTIONS Allowed HTTP Methods: HEAD, GET, POST, OPTIONS, TRACE
GMHBDRNF Web Server returns a valid response with junk HTTP method
DEBUG DEBUG HTTP verb may show server debugging information. See h
OSVDB-877: TRACE HTTP TRACE method is active, suggesting the host
OSVDB-62684: GET /wordpress/wp-content/plugins/hello.php: The Word
GET Server leaks inodes via ETags, header found with file /wordpre
GET /wordpress/wp-links-opml.php: This WordPress script reveals th
OSVDB-3092: GET /wordpress/license.txt: License file found may ide
GET Cookie wordpress_test_cookie created without the httponly flag
GET /wordpress/wp-login.php: Wordpress login found
```

Fig. 4. Scan summary of WordPress site

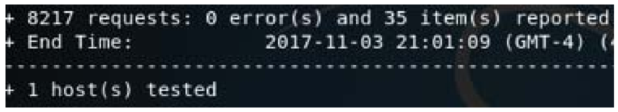
2. Drupal site: Using Nitko to scan the target http://localhost/drupal, it provided the basic information about the target host server as illustrated in Fig. 5.



```
root@kali-test:~# nikto -host http://localhost/drupal
- Nikto v2.1.6
---------------------------------------------------------------------------
+ Target IP:          127.0.0.1
+ Target Hostname:    localhost
+ Target Port:        80
+ Start Time:         2017-11-03 19:40:43 (GMT-4)
---------------------------------------------------------------------------
+ Server: Apache/2.4.26 (Unix) OpenSSL/1.0.2l PHP/5.6.
/v5.16.3
```

Fig. 5. Information on target site

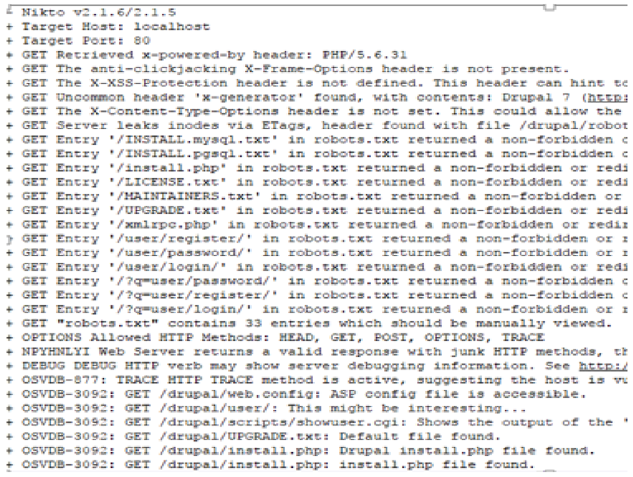
After successfully scanning the site, the results are illustrated in Fig. 6.



```
+ 8217 requests: 0 error(s) and 35 item(s) reported
+ End Time:           2017-11-03 21:01:09 (GMT-4) (
---------------------------------------------------------------------------
+ 1 host(s) tested
```

Fig. 6. Scan results

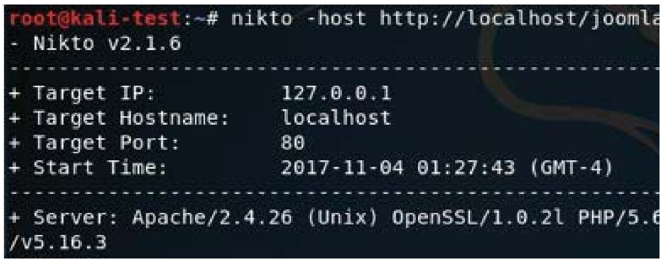
These results show that 8217 files were scanned for vulnerabilities and reported 35 items that might be vulnerable to attacks. In Fig. 7, the detailed list of items found to be vulnerable on our default Drupal site.



```
Nikto v2.1.6/2.1.5
+ Target Host: localhost
+ Target Port: 80
+ GET Retrieved x-powered-by header: PHP/5.6.31
+ GET The anti-clickjacking X-Frame-Options header is not present.
+ GET The X-XSS-Protection header is not defined. This header can hint tc
+ GET Uncommon header 'x-generator' found, with contents: Drupal 7 (http:
+ GET The X-Content-Type-Options header is not set. This could allow the
+ GET Server leaks inodes via ETags, header found with file /drupal/robot
+ GET Entry '/INSTALL.mysql.txt' in robots.txt returned a non-forbidden c
+ GET Entry '/INSTALL.pgsql.txt' in robots.txt returned a non-forbidden c
+ GET Entry '/install.php' in robots.txt returned a non-forbidden or redi
+ GET Entry '/LICENSE.txt' in robots.txt returned a non-forbidden or redi
+ GET Entry '/MAINTAINERS.txt' in robots.txt returned a non-forbidden or
+ GET Entry '/UPGRADE.txt' in robots.txt returned a non-forbidden or redi
+ GET Entry '/xmlrpc.php' in robots.txt returned a non-forbidden or redir
} GET Entry '/user/register/' in robots.txt returned a non-forbidden or r
+ GET Entry '/user/password/' in robots.txt returned a non-forbidden or r
+ GET Entry '/user/login/' in robots.txt returned a non-forbidden or redi
+ GET Entry '/?q=user/password/' in robots.txt returned a non-forbidden c
+ GET Entry '/?q=user/register/' in robots.txt returned a non-forbidden c
+ GET Entry '/?q=user/login/' in robots.txt returned a non-forbidden or r
+ GET "robots.txt" contains 33 entries which should be manually viewed.
+ OPTIONS Allowed HTTP Methods: HEAD, GET, POST, OPTIONS, TRACE
+ NPYHNLYI Web Server returns a valid response with junk HTTP methods, th
+ DEBUG DEBUG HTTP verb may show server debugging information. See http:/
+ OSVDB-877: TRACE HTTP TRACE method is active, suggesting the host is vu
+ OSVDB-3092: GET /drupal/web.config: ASP config file is accessible.
+ OSVDB-3092: GET /drupal/user/: This might be interesting...
+ OSVDB-3092: GET /drupal/scripts/showuser.cgi: Shows the output of the '
+ OSVDB-3092: GET /drupal/UPGRADE.txt: Default file found.
+ OSVDB-3092: GET /drupal/install.php: Drupal install.php file found.
+ OSVDB-3092: GET /drupal/install.php: install.php file found.
```

Fig. 7. Scan summary of Drupal site

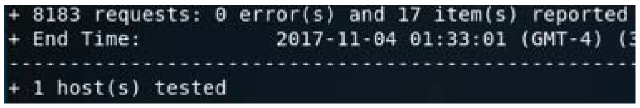
3. Joomla! Site: Using Nitko to scan the target http://localhost/joomla, it provided the basic information about the target host server as illustrated in Fig. 8.



```
root@kali-test:~# nikto -host http://localhost/joomla
- Nikto v2.1.6
---------------------------------------------------------
+ Target IP:          127.0.0.1
+ Target Hostname:    localhost
+ Target Port:        80
+ Start Time:         2017-11-04 01:27:43 (GMT-4)
---------------------------------------------------------
+ Server: Apache/2.4.26 (Unix) OpenSSL/1.0.2l PHP/5.6
/v5.16.3
```

Fig. 8. Joomla site information

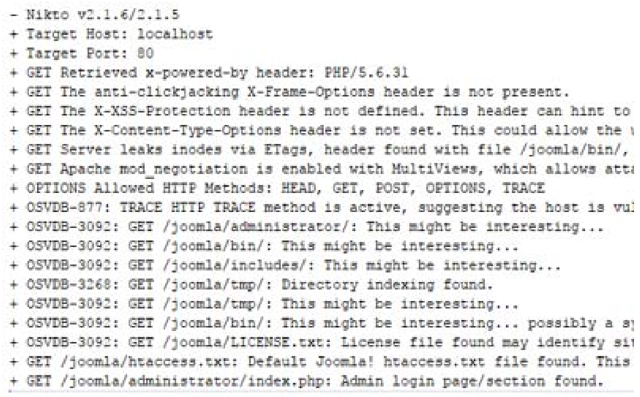
After successfully scanning the site, the results are illustrated in Fig. 9.



```
+ 8183 requests: 0 error(s) and 17 item(s) reported
+ End Time:          2017-11-04 01:33:01 (GMT-4) (3
---------------------------------------------------------
+ 1 host(s) tested
```

Fig. 9. Scan Results

This result shows that 8183 files were scanned for vulnerabilities and reported 17 items that might be vulnerable to attacks. In Fig. 10, the detailed list of items found to be vulnerable on our default Drupal site.



```
- Nikto v2.1.6/2.1.5
+ Target Host: localhost
+ Target Port: 80
+ GET Retrieved x-powered-by header: PHP/5.6.31
+ GET The anti-clickjacking X-Frame-Options header is not present.
+ GET The X-XSS-Protection header is not defined. This header can hint to
+ GET The X-Content-Type-Options header is not set. This could allow the
+ GET Server leaks inodes via ETags, header found with file /joomla/bin/,
+ GET Apache mod_negotiation is enabled with MultiViews, which allows att
+ OPTIONS Allowed HTTP Methods: HEAD, GET, POST, OPTIONS, TRACE
+ OSVDB-877: TRACE HTTP TRACE method is active, suggesting the host is vul
+ OSVDB-3092: GET /joomla/administrator/: This might be interesting...
+ OSVDB-3092: GET /joomla/bin/: This might be interesting...
+ OSVDB-3092: GET /joomla/includes/: This might be interesting...
+ OSVDB-3268: GET /joomla/tmp/: Directory indexing found.
+ OSVDB-3092: GET /joomla/tmp/: This might be interesting...
+ OSVDB-3092: GET /joomla/bin/: This might be interesting... possibly a s
+ OSVDB-3092: GET /joomla/LICENSE.txt: License file found may identify si
+ GET /joomla/htaccess.txt: Default Joomla! htaccess.txt file found. This
+ GET /joomla/administrator/index.php: Admin login page/section found.
```

Fig. 10. Scan summary of Joomla site

Table I shows the comparison of the scan results of possible vulnerabilities found on the three content management systems installed for this research.

Table I. Summary of Nikto scan results.

Content Management System	Warnings	SQLI Vulnerabilities	Other Vulnerabilities (XSS, XST, Brute force)
WordPress	16	0	4
Drupal	35	0	4
Joomla!	17	0	4

B. Exploitation

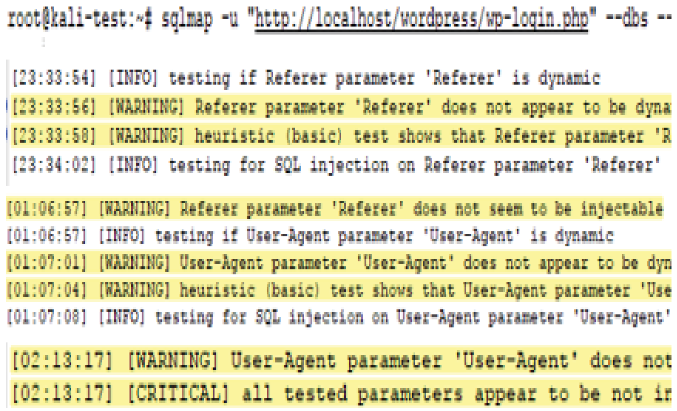
The information gathered for the vulnerability scan carried out on our three content management systems (WordPress, Joomla, and Drupal) shows that each default site template is not susceptible to SQL Injection attack. Nevertheless, SQLMAP is still going to be used to exploit each site using both a GET parameter and a POST parameter, these two methods are the most commonly identified types of SQLi and record what happens. By default, SQLMAP treats all requests as a GET method, so to specify a POST method, the parameters must be included in the data section of the URL.

For each test, we would specify some options in the request such as guessing the DBMS, increasing the risk and level, this way SQLMAP tries more clever stuff to find and exploit vulnerabilities. It means SQLMAP will try not only visible SQL injections but also blind SQLi which are not visible. Figs. 12 and 13 highlight some of the SQLMAP commands for SQL injection that we carried out on WordPress, Joomla and Drupal sites.

1. WordPress Site

GET method:

Command sqlmap -u "http://localhost/wordpress/wp-login.php --level=3" --risk=2 --dbs --dbms=MariaDB

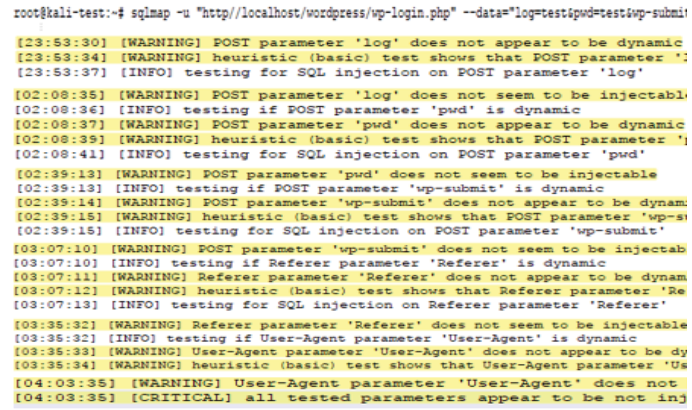


```
root@kali-test:~# sqlmap -u "http://localhost/wordpress/wp-login.php" --dbs --

[23:33:54] [INFO] testing if Referer parameter 'Referer' is dynamic
[23:33:56] [WARNING] Referer parameter 'Referer' does not appear to be dyna
[23:33:58] [WARNING] heuristic (basic) test shows that Referer parameter 'R
[23:34:02] [INFO] testing for SQL injection on Referer parameter 'Referer'

[01:06:57] [WARNING] Referer parameter 'Referer' does not seem to be injectable
[01:06:57] [INFO] testing if User-Agent parameter 'User-Agent' is dynamic
[01:07:01] [WARNING] User-Agent parameter 'User-Agent' does not appear to be dyn
[01:07:04] [WARNING] heuristic (basic) test shows that User-Agent parameter 'Use
[01:07:08] [INFO] testing for SQL injection on User-Agent parameter 'User-Agent'

[02:13:17] [WARNING] User-Agent parameter 'User-Agent' does not
[02:13:17] [CRITICAL] all tested parameters appear to be not ir
```

Fig. 12. SQLMAP results using GET method on WordPress login page

POST method:

sqlmap -u "http//localhost/wordpress/wp-login.php" --data="log=test&pwd=test&wp-submit=Log+in" --dbs --level=3 --risk=2 --dbms=MariaDB



```
root@kali-test:~# sqlmap -u "http//localhost/wordpress/wp-login.php" --data="log=test&pwd=test&wp-submit

[23:53:30] [WARNING] POST parameter 'log' does not appear to be dynamic
[23:53:34] [WARNING] heuristic (basic) test shows that POST parameter 'l
[23:53:37] [INFO] testing for SQL injection on POST parameter 'log'

[02:08:35] [WARNING] POST parameter 'log' does not seem to be injectabl
[02:08:36] [INFO] testing if POST parameter 'pwd' is dynamic
[02:08:37] [WARNING] POST parameter 'pwd' does not appear to be dynamic
[02:08:39] [WARNING] heuristic (basic) test shows that POST parameter 'p
[02:08:41] [INFO] testing for SQL injection on POST parameter 'pwd'

[02:39:13] [WARNING] POST parameter 'pwd' does not seem to be injectable
[02:39:13] [INFO] testing if POST parameter 'wp-submit' is dynamic
[02:39:14] [WARNING] POST parameter 'wp-submit' does not appear to be dynam
[02:39:15] [WARNING] heuristic (basic) test shows that POST parameter 'wp-su
[02:39:15] [INFO] testing for SQL injection on POST parameter 'wp-submit'

[03:07:10] [WARNING] POST parameter 'wp-submit' does not seem to be injectab
[03:07:10] [INFO] testing if Referer parameter 'Referer' is dynamic
[03:07:11] [WARNING] Referer parameter 'Referer' does not appear to be dynam
[03:07:12] [WARNING] heuristic (basic) test shows that Referer parameter 'Re
[03:07:13] [INFO] testing for SQL injection on Referer parameter 'Referer'

[03:35:32] [WARNING] Referer parameter 'Referer' does not seem to be injectable
[03:35:32] [INFO] testing if User-Agent parameter 'User-Agent' is dynamic
[03:35:33] [WARNING] User-Agent parameter 'User-Agent' does not appear to be dy
[03:35:34] [WARNING] heuristic (basic) test shows that User-Agent parameter 'Us

[04:03:35] [WARNING] User-Agent parameter 'User-Agent' does not
[04:03:35] [CRITICAL] all tested parameters appear to be not inj
```

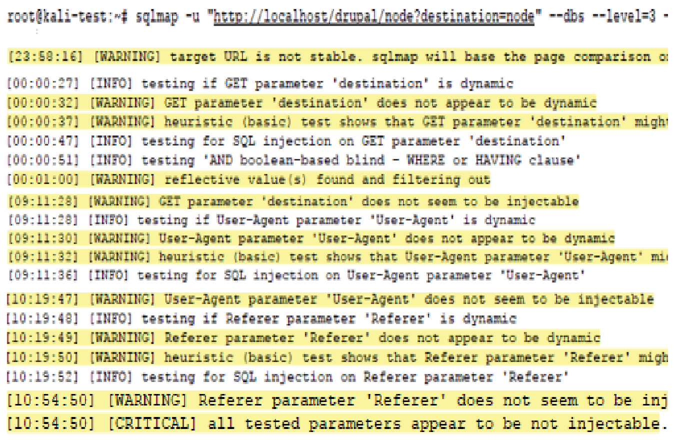
Fig. 13. SQLMAP results using POST method on WordPress

The lines highlighted in yellow display information about the parameters tested by both commands (GET and POST methods). It shows that all tested parameters appear to not be injectable.

2. *Drupal Site*

GET Method:

Command sqlmap -u "http://localhost/drupal/node?destination=node" --dbs --level=3 --risk=2 --dbms=MySQL.

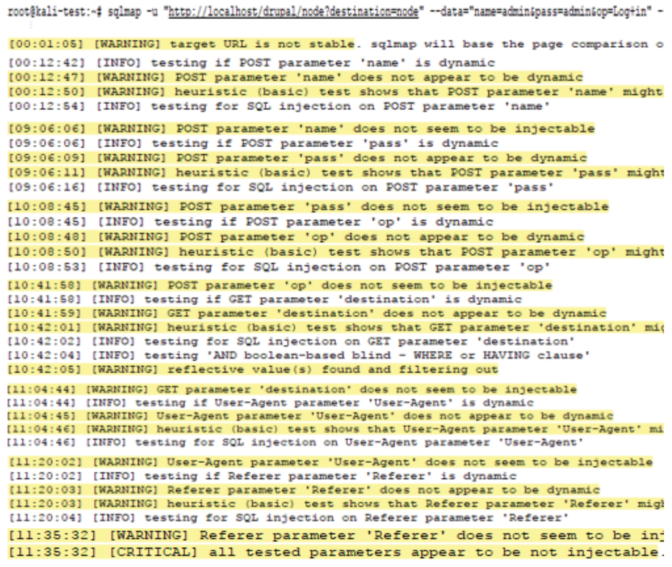


```
root@kali-test:~# sqlmap -u "http://localhost/drupal/node?destination=node" --dbs --level=3 -

[23:58:16] [WARNING] target URL is not stable. sqlmap will base the page comparison o

[00:00:27] [INFO] testing if GET parameter 'destination' is dynamic
[00:00:32] [WARNING] GET parameter 'destination' does not appear to be dynamic
[00:00:37] [WARNING] heuristic (basic) test shows that GET parameter 'destination' migh
[00:00:47] [INFO] testing for SQL injection on GET parameter 'destination'
[00:00:51] [INFO] testing 'AND boolean-based blind - WHERE or HAVING clause'
[00:01:00] [WARNING] reflective value(s) found and filtering out
[09:11:28] [WARNING] GET parameter 'destination' does not seem to be injectable
[09:11:28] [INFO] testing if User-Agent parameter 'User-Agent' is dynamic
[09:11:30] [WARNING] User-Agent parameter 'User-Agent' does not appear to be dynamic
[09:11:32] [WARNING] heuristic (basic) test shows that User-Agent parameter 'User-Agent' mi
[09:11:36] [INFO] testing for SQL injection on User-Agent parameter 'User-Agent'
[10:19:47] [WARNING] User-Agent parameter 'User-Agent' does not seem to be injectable
[10:19:48] [INFO] testing if Referer parameter 'Referer' is dynamic
[10:19:49] [WARNING] Referer parameter 'Referer' does not appear to be dynamic
[10:19:50] [WARNING] heuristic (basic) test shows that Referer parameter 'Referer' migh
[10:19:52] [INFO] testing for SQL injection on Referer parameter 'Referer'
[10:54:50] [WARNING] Referer parameter 'Referer' does not seem to be inj
[10:54:50] [CRITICAL] all tested parameters appear to be not injectable.
```

Fig. 14. SQLMAP results using GET method on Drupal site.

POST method:

Command sqlmap -u "http://localhost/drupal/node?destination=node" --data="name=admin&pass=admin&op=Log+in" --dbs --level=3 --risk=2 --dbms=MySQL



```
root@kali-test:~# sqlmap -u "http://localhost/drupal/node?destination=node" --data="name=admin&pass=admin&op=Log+in" -

[00:01:05] [WARNING] target URL is not stable. sqlmap will base the page comparison o
[00:12:42] [INFO] testing if POST parameter 'name' is dynamic
[00:12:47] [WARNING] POST parameter 'name' does not appear to be dynamic
[00:12:50] [WARNING] heuristic (basic) test shows that POST parameter 'name' might
[00:12:54] [INFO] testing for SQL injection on POST parameter 'name'
[09:06:06] [WARNING] POST parameter 'name' does not seem to be injectable
[09:06:06] [INFO] testing if POST parameter 'pass' is dynamic
[09:06:09] [WARNING] POST parameter 'pass' does not appear to be dynamic
[09:06:11] [WARNING] heuristic (basic) test shows that POST parameter 'pass' might
[09:06:16] [INFO] testing for SQL injection on POST parameter 'pass'
[10:08:45] [WARNING] POST parameter 'pass' does not seem to be injectable
[10:08:45] [INFO] testing if POST parameter 'op' is dynamic
[10:08:48] [WARNING] POST parameter 'op' does not appear to be dynamic
[10:08:50] [WARNING] heuristic (basic) test shows that POST parameter 'op' might
[10:08:53] [INFO] testing for SQL injection on POST parameter 'op'
[10:41:58] [WARNING] POST parameter 'op' does not seem to be injectable
[10:41:58] [INFO] testing if GET parameter 'destination' is dynamic
[10:41:59] [WARNING] GET parameter 'destination' does not appear to be dynamic
[10:42:01] [WARNING] heuristic (basic) test shows that GET parameter 'destination' mi
[10:42:02] [INFO] testing for SQL injection on GET parameter 'destination'
[10:42:04] [INFO] testing 'AND boolean-based blind - WHERE or HAVING clause'
[10:42:05] [WARNING] reflective value(s) found and filtering out
[11:04:44] [WARNING] GET parameter 'destination' does not seem to be injectable
[11:04:44] [INFO] testing if User-Agent parameter 'User-Agent' is dynamic
[11:04:45] [WARNING] User-Agent parameter 'User-Agent' does not appear to be dynamic
[11:04:46] [WARNING] heuristic (basic) test shows that User-Agent parameter 'User-Agent' mi
[11:04:46] [INFO] testing for SQL injection on User-Agent parameter 'User-Agent'
[11:20:02] [WARNING] User-Agent parameter 'User-Agent' does not seem to be injectable
[11:20:02] [INFO] testing if Referer parameter 'Referer' is dynamic
[11:20:03] [WARNING] Referer parameter 'Referer' does not appear to be dynamic
[11:20:03] [WARNING] heuristic (basic) test shows that Referer parameter 'Referer' migh
[11:20:04] [INFO] testing for SQL injection on Referer parameter 'Referer'
[11:35:32] [WARNING] Referer parameter 'Referer' does not seem to be inj
[11:35:32] [CRITICAL] all tested parameters appear to be not injectable.
```

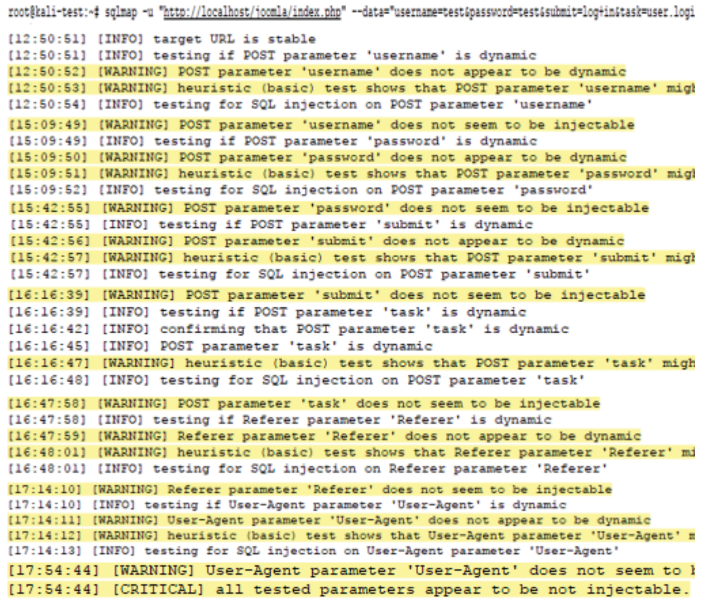
Fig. 15. SQLMAP results using POST method on Drupal site.

All parameters tested using both GET and POST method appear to be not injectable as illustrated in Figs. 14 and 15.

3. *Joomla Site*

POST method:

Command sqlmap -u "http://localhost/joomla/index.php" --data="username=test&password=test&submit=log+in&task=user.login" --dbs --level=3 --risk=2 --dbms=MySQL

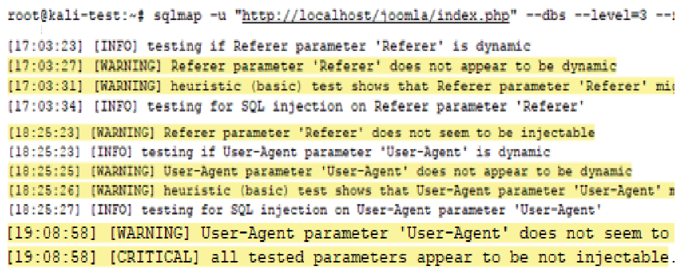


```
root@kali-test:~# sqlmap -u "http://localhost/joomla/index.php" --data="username=test&password=test&submit=log+in&task=user.logi
[12:50:51] [INFO] target URL is stable
[12:50:51] [INFO] testing if POST parameter 'username' is dynamic
[12:50:52] [WARNING] POST parameter 'username' does not appear to be dynamic
[12:50:53] [WARNING] heuristic (basic) test shows that POST parameter 'username' migh
[12:50:54] [INFO] testing for SQL injection on POST parameter 'username'
[15:09:49] [WARNING] POST parameter 'username' does not seem to be injectable
[15:09:49] [INFO] testing if POST parameter 'password' is dynamic
[15:09:50] [WARNING] POST parameter 'password' does not appear to be dynamic
[15:09:51] [WARNING] heuristic (basic) test shows that POST parameter 'password' migh
[15:09:52] [INFO] testing for SQL injection on POST parameter 'password'
[15:42:55] [WARNING] POST parameter 'password' does not seem to be injectable
[15:42:55] [INFO] testing if POST parameter 'submit' is dynamic
[15:42:56] [WARNING] POST parameter 'submit' does not appear to be dynamic
[15:42:57] [WARNING] heuristic (basic) test shows that POST parameter 'submit' migh
[15:42:57] [INFO] testing for SQL injection on POST parameter 'submit'
[16:16:39] [WARNING] POST parameter 'submit' does not seem to be injectable
[16:16:39] [INFO] testing if POST parameter 'task' is dynamic
[16:16:42] [INFO] confirming that POST parameter 'task' is dynamic
[16:16:45] [INFO] POST parameter 'task' is dynamic
[16:16:47] [WARNING] heuristic (basic) test shows that POST parameter 'task' migh
[16:16:48] [INFO] testing for SQL injection on POST parameter 'task'
[16:47:58] [WARNING] POST parameter 'task' does not seem to be injectable
[16:47:58] [INFO] testing if Referer parameter 'Referer' is dynamic
[16:47:59] [WARNING] Referer parameter 'Referer' does not appear to be dynamic
[16:48:01] [WARNING] heuristic (basic) test shows that Referer parameter 'Referer' mi
[16:48:01] [INFO] testing for SQL injection on Referer parameter 'Referer'
[17:14:10] [WARNING] Referer parameter 'Referer' does not seem to be injectable
[17:14:10] [INFO] testing if User-Agent parameter 'User-Agent' is dynamic
[17:14:11] [WARNING] User-Agent parameter 'User-Agent' does not appear to be dynamic
[17:14:12] [WARNING] heuristic (basic) test shows that User-Agent parameter 'User-Agent' m
[17:14:13] [INFO] testing for SQL injection on User-Agent parameter 'User-Agent'
[17:54:44] [WARNING] User-Agent parameter 'User-Agent' does not seem to
[17:54:44] [CRITICAL] all tested parameters appear to be not injectable.
```

Fig. 16. SQLMAP results using POST method on Joomla site.

GET method:

Command sqlmap -u "http://localhost/joomla/index.php" --dbs --level=3 --risk=2 --dbms=MySQL



```
root@kali-test:~# sqlmap -u "http://localhost/joomla/index.php" --dbs --level=3 --

[17:03:23] [INFO] testing if Referer parameter 'Referer' is dynamic
[17:03:27] [WARNING] Referer parameter 'Referer' does not appear to be dynamic
[17:03:31] [WARNING] heuristic (basic) test shows that Referer parameter 'Referer' mi
[17:03:34] [INFO] testing for SQL injection on Referer parameter 'Referer'
[18:25:23] [WARNING] Referer parameter 'Referer' does not seem to be injectable
[18:25:23] [INFO] testing if User-Agent parameter 'User-Agent' is dynamic
[18:25:25] [WARNING] User-Agent parameter 'User-Agent' does not appear to be dynamic
[18:25:26] [WARNING] heuristic (basic) test shows that User-Agent parameter 'User-Agent' m
[18:25:27] [INFO] testing for SQL injection on User-Agent parameter 'User-Agent'
[19:08:58] [WARNING] User-Agent parameter 'User-Agent' does not seem to
[19:08:58] [CRITICAL] all tested parameters appear to be not injectable.
```

Fig. 17. SQLMAP results using POST method on Joomla site.

After a couple of tests trying to exploit different possible vulnerabilities on the default login pages of each site we were unable to gain access to the back-end database or successfully carry out a SQL injection attack. By Default, all the sites were enabled with WAF (web application firewall) or IPS /IDS which protects web applications from the most common types of SQL injection attacks and because the sites were static pages with no third-party plugins or themes that also made it very difficult to hack.

C. Implications for Practice

As discovered during the penetration test carried out on the sites (WordPress, Drupal, and Joomla!), we found that the sites were all protected by an Intrusion Protection system (IPS) or Web Application Firewall (WAF) which helps filter queries sent to the database for malicious codes and scripts. Content management systems are vulnerable by nature because they are built on open source frameworks. Web developers are left with the task of monitoring and maintaining their sites. There is also the issue of plugins and themes created by third-party developers which may introduce an additional set of vulnerabilities. So, in the real world, these content management systems would be exposed to various kind security risks, and the only way to avoid and protect the systems would be to adhere to best practices for web security. To mitigate against such security risks, we suggest site owners should take the following actions:

- Create a regular schedule to update or patch their CMS, and all installed plugins and themes. This will ensure that all components are up-to-date and prevent any security loophole that might arise from using an outdated plugin or patch.
- Perform a regular backup of the CMS and its underlying database on a weekly basis. So, in the case of a data breach, the user can always revert to previous configurations with not sensitive data lost.
- Subscribe to a regularly-updated list of vulnerabilities for the specific CMS being used (e.g., Joomla!).
- Plugins and themes from third-party developers should be well vetted and validated before being used.
- Delete default admin usernames and use strong passwords. This makes it difficult for a brute force attack to be carried out on the site.
- Implement a two-factor authentication (2FA) for an additional layer of protection.
- Enable HTTPS (HyperText Transfer Protocol Secure) on their site. This protects the privacy and integrity of the exchanged data.

V. Conclusions

As of today, OWASP (Open Web Application Security Project) ranks SQL injection as number one on its list of top 10 most critical web application security risks. Hackers carry out numerous attacks using vulnerabilities found on websites compromising thousands daily. CMS (content management system) being a significant player in the world of website development, it is paramount that extra attention is paid to possible security vulnerabilities that could be found in them. These research paper results show that the commonly used CMSes, WordPress, Drupal, and Joomla offers some level of security called Web Application Firewall (WAF) which prevents against some SQL injections but as the site grows in contents and size, they become more vulnerable to SQLi attacks. For these, we recommended some implications for practice that should be put in place to avoid attacks. For future works, more penetration tests should be done to test for other types of attacks such as Brute Force attack, Cross Site Scripting, etc. Doing this on a CMS site with dynamic pages hosted on the web would help create a better real-life scenario.

References

[1] "SQL Injection: By The Numbers – Blog | Imperva," 2011-09-20 2011.

[2] L. Xiao, S. Matsumoto, T. Ishikawa, and K. Sakurai, "SQL Injection Attack Detection Method using Expectation Criterion," in *Computing and Networking (CANDAR), 2016 Fourth International Symposium on*, 2016, pp. 649-654.

[3] F. Alserhani, M. Akhlaq, I. U. Awan, and A. J. Cullen, "Event-based alert correlation system to detect SQLI activities," in *Advanced Information Networking and Applications (AINA), 2011 IEEE International Conference on*, 2011, pp. 175-182.

[4] A. K. Singh and S. Roy, "A network-based vulnerability scanner for detecting SQLI attacks in web applications," in *Recent Advances in Information Technology (RAIT), 2012 1st International Conference on*, 2012, pp. 585-590.

[5] V. K. Gudipati, T. Venna, S. Subburaj, and O. Abuzaghleh, "Advanced automated SQL injection attacks and defensive mechanisms," in *Industrial Electronics, Technology & Automation (CT-IETA), Annual Connecticut Conference on*, 2016, pp. 1-6.

[6] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst, "Automatic creation of SQL injection and cross-site scripting attacks," in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, 2009, pp. 199-209.

[7] N. Antunes and M. Vieira, "Comparing the effectiveness of penetration testing and static code analysis on the detection of sql injection vulnerabilities in web services," in *Dependable Computing, 2009. PRDC'09. 15th IEEE Pacific Rim International Symposium on*, 2009, pp. 301-306.

[8] H. Wu and G. Gao, "Test SQL injection vulnerabilities in web applications based on structure matching," in *Computer Science and Network Technology (ICCSNT), 2011 International Conference on*, 2011, pp. 935-938.

[9] N. Antunes and M. Vieira, "Enhancing penetration testing with attack signatures and interface monitoring for the detection of injection vulnerabilities in web services," in *Services Computing (SCC), 2011 IEEE International Conference on*, 2011, pp. 104-111.