International Journal of Engineering & Technology, 7 (2.32) (2018) 389-392

International Journal of Engineering & Technology

Website: www.sciencepubco.com/index.php/IJET

Research paper

Detecting SQL Injection Using Correlative Log Analysis

T. Sreeja[1], Dr. Manna Sheela Rani Chetty[2], Sekhar Babu Boddu[3]

[1]MTech (Cyber Security and Digital Forensics), CSE, KLEF, Vaddeswaram, Guntur Dt, India
[2]Professor, Assistant Professor[3], Dept of CSE, KLEF, Vaddeswaram, Guntur Dt, India
**Corresponding author E-mail: sreejacse333@gmail.com*

Abstract

The spiking landscape of cyber-attacks is reflecting its trend towards invoking vulnerabilities in a web application. The vulnerabilities seem to be over-growing second by second beside being over-coming time to time. The reason behind is, new attack vectors are often being deployed by the threat actors. The global cyber security market alone has brought a turnover of about $350 billion, which shows how wide the attack landscape is and how expensive it is to detect, protect and respond to the cyber issues. Most of the security experts have quoted that, the average cost of a data breach will exceed to $150million by 2020 and about 80 percent of the global demography were nowhere aware of such attacks. From the past few years, SQL injection is acting as a major vector in breaching the sensitive data. Detecting SQL injection through log correlation is the most effective methodology utilized under adaptive environments seeking no tool investigation. This paper exposes a detection methodology of an SQL injection attack without any mere concentration on automated tools. The paper goes with a motto of detection through configuring the available resources like web server,database,and an IDS in a way of creating adaptable environment that can bring the entire attacker information through log analysis. The paper would represent the attacker phases in a finite automata.

Keywords:*Detection, SQL injection, Web server configuration, database configuration, IDS configuration, log correlation, finite automata..*

1. Introduction

Most of the productive markets now are furnishing their services only through web applications, which shows how frequently, the users, the employees and the clients are interacting with web interfaces for acquiring their interested services. So, solely these web applications would act as a data asset to the hacker. As to pretend themselves to be secure, most of the production environments now always seek of deploying Web Application Firewalls (WAF) for overcoming the overwhelming devastating effects over the web applications. But those defensive infrastructures may lag as the attacker clock goes ahead. Because, the most significant task of any administrator is to configure the system to be secure, but it is also important to know it is secure. The only way to know that the system is secure is through informative and trustworthy log files. Detecting the attacker through manual log analysis would help one to be independent rather than relying on 3rd party applications and libraries. Such post-intrusion analysis would set a standard for it to act as prior analysis to the upcoming intrusion as well.

Over the last decade, SQL injection seemed to be the most catastrophic threat which made itself to occupy the 1st place among OWASP top 10 attacks. The principle of basic SQL injection is to take advantage of insecure code on a system connected to the internet in order to pass commands directly to the database and to then take advantage of a poorly secured system to leverage on attacker's access.[1] The attacker first encounters the data entry points across multiple pages of the web application. Then, he tries malicious patterns in each data entry point within an HTTP request of a web application. Finally, the HTTP response received on thereby will be scanned for several indications for the existence of vulnerability. Such a vulnerability would often be recognized through the detailed error messages displayed by the web application. This shows that there is utmost importance for the web server, IDS and databases to get configured securely, such that those logs could be correlated in determining the attacker's previous state and next state. Besides, performing automated log analysis, the manual way of log analysis through configuring web servers, IDS and databases as per our production environment is purely necessary. Because logs are the only source for detecting the suspicious behaviour of the attacker.

Generally, a web application information that is channeled to the web servers, would either go in the form of URL's, or user cookies, or form inputs (POST's and GET's). Because these are the only primary sources for the attacker to execute his malicious scripts, these parameters would count on as the attack vectors sought by the attacker. No matter which attack vector has been executed, but rather been apart as an administrator, it's prior to detect the attack vector and prevent it. Detection has the most vital role in today's cyber threats, because it's been a first initiative step taken by any incident responder at the crime scene. For example, let's suppose that a hosted website has been defaced by an anonymous hacker. Certainly, there might be tools and modules out there that can monitor the traffic flow and generate reports of data, which we could naturally get from the web server and database resources we have. It's just a matter of logging what kind of information we are actually in need for detecting the attacker. In the worst-case scenario of not having any detecting tools at the crime scene, then one could probably go through the web server and database logs for determining the actions performed by the hacker. In such a case, it's prior to have the knowledge regarding web server and database configuring.

2. Related Work

In [6] the author has presented different practical scenarios that can come under SQL injection attack where he analysed the suspicious strings using ZAP tool under the scenarios like injection through HTTP GET parameter, injection through HTTP POST parameter and injection through modifying cookie data.

In [5] the author has presented different unique and advanced techniques used by the attacker under SQL injection attack. The strings like ORDER BY is used for getting the first and last records of the database. The attacker may enter as:

Username: a' OR '1' = '1 Password : a' OR '1' = '1' ORDER BY 1 DESC

Constructed query: SELECT name from users where username='a' OR '1' = '1' AND password='a' OR '1' = '1' ORDER BY 1 DESC;

This would yield a result of showing the records of the database in the descending order and thereby knowing the number of records that the database is currently holding.

3. Proposed Architecture

For performing post-intrusion analysis, a vulnerable application is hosted in a local web server and SQL injection is performed on the application in different vectors. The Web server, IDS and database are configured to an adaptable environment for detecting SQL injection attack and thereby the logs are analysed to get the entire attacker information as shown in the fig1.

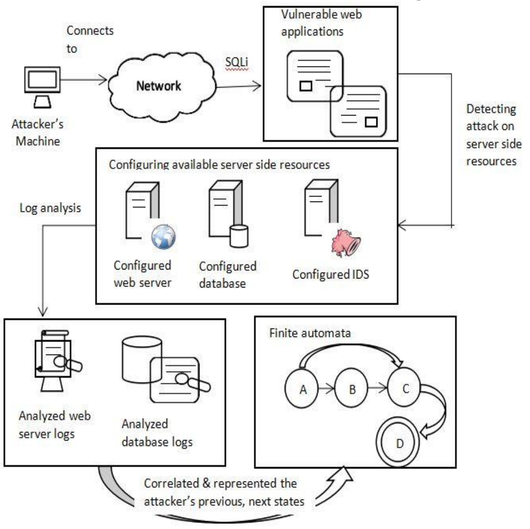

Connects to
SQLi
Network
Vulnerable web applications
Attacker's Machine
Detecting attack on server side resources
Configuring available server side resources
Log analysis
Configured web server
Configured database
Configured IDS
Finite automata
A
B
C
D
Analyzed web server logs
Analyzed database logs
Correlated & represented the attacker's previous, next states

Fig 1: Proposed architecture

4. Configuration Done on Web Server

Apache produces two essential logs access logs and error logs. Access logs would record all the clients requests whereas the error logs would log all the exceptional events that occurred during the period of server start and server stop and it even includes the debug information as well.

Apache provides a flexibility of logging custom logs where we can log all the information we need into a separate log file named as custom.log. The default log format of apache would only record the client request line. So, there is need to enable more information in order to determine the attacker. The httpd.conf file will be configured with the required log format of getting SQL injection attack information in the following way:

%a is included in the log format for logging the remote ip address of the attacker who performed SQL injection attack.
%A is added to log, server ip address. This would help in the scenario of hosting multiple servers where we could determine which server has been compromised.

%f is added to log, the name of the file that client is requesting. This information would help if the client is accessing a restricted file, upon leveraging root permissions.

The mod_forensic module of apache reveals the client request which made the server to crash. This module is made to record the logs into a separate log file. The functionality of this module enables a unique id for each client request. This unique id is prefixed with '+' and '-'signatures where the positive sign forensic id represents the start of client's session and the negative sign forensic id represents the end of the session. In between information that is holding would represent what all are the pages requested and accessed by the client. For supposing the attacker has taken the advantage of the web server and made it to crash then the negative signature forensic id will not be logged because the attack has happened before the session is timed out. So, this would largely help in investigating the attack process.

Should enable the following modules to get forensic information:

LoadModule log_forensic_module modules/mod_log_forensic.so

LoadModule unique_id_module modules/mod_unique_id.so

Added this line in the main configuration section of the configuration file. This would create a new configuration file under logs folder.

ForensicLog logs/forensic_log

Added %{forensic_id}n string in the log format part of mod_log_config module

The mod_usertrack module of apache logs client cookie information. For enabling cookie information, the following lines must be added in the main configuration file:

CookieTracking on
CookieName ABC
CookieExpires "6 weeks"

Enabled the module mod_usertrack and add the string %{cookie}i in the log format of log_config_module.

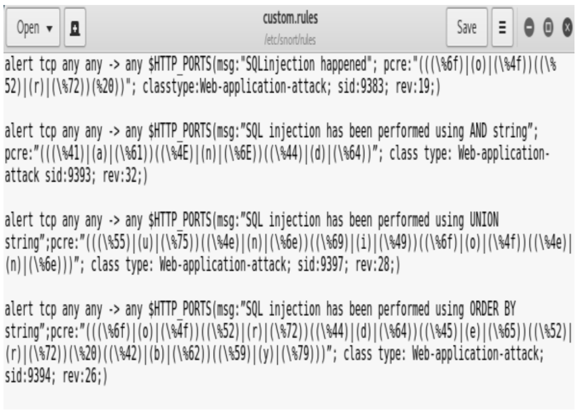
Putting it together:
LogFormat "%A %a (%{forensic-id}n) %f ie}i) %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-Agent}i\"" combined

5. Configuration Done on Snort Ids

SNORT is an effective rule based Network Intrusion Detection System (NIDS) tool to identify intrusion attacks[4]. Basically, snort goes through various phases for detecting the intrusion where it would initially capture the network packet and then it searches for the suspicious connection attempts to TCP/UDP ports. Then after the packet is sent to the detection engine (bearing preloaded and customised rules) for detecting the malicious patterns against the applied rules.

Snort by default have certain signatures for detecting SQL injection attacks but they can easily be bypassed by the attacker. This paper proposes a Perl's compatible regular expressions(pcre) for detecting SQL injection attacks.

The advanced SQL injection strings proposed in [5] is considered for writing the snort rules. The following are the rules written in regular expression format as shown in fig 2:



```
alert tcp any any -> any $HTTP_PORTS(msg:"SQLinjection happened"; pcre:"(((\%6f)|(o)|(\%4f))((\%
52)|(r)|(\%72))(%20))"; classtype:Web-application-attack; sid:9383; rev:19;)

alert tcp any any -> any $HTTP_PORTS(msg:"SQL injection has been performed using AND string";
pcre:"(((\%41)|(a)|(\%61))((\%4E)|(n)|(\%6E))((\%44)|(d)|(\%64))"; class type: Web-application-
attack sid:9393; rev:32;)

alert tcp any any -> any $HTTP_PORTS(msg:"SQL injection has been performed using UNION
string";pcre:"(((\%55)|(u)|(\%75))((\%4e)|(n)|(\%6e))((\%69)|(i)|(\%49))((\%6f)|(o)|(\%4f))((\%4e)|
(n)|(\%6e)))"; class type: Web-application-attack; sid:9397; rev:28;)

alert tcp any any -> any $HTTP_PORTS(msg:"SQL injection has been performed using ORDER BY
string";pcre:"(((\%6f)|(o)|(\%4f))((\%52)|(r)|(\%72))((\%44)|(d)|(\%64))((\%45)|(e)|(\%65))((\%52)|
(r)|(\%72))(\%20)((\%42)|(b)|(\%62))((\%59)|(y)|(\%79)))"; class type: Web-application-attack;
sid:9394; rev:26;)
```

Fig 2: Regular expression based snort rules for SQL injection detection

All these rules are written into a separate rule file named custom.rules and saved under rules folder of the snort application.

6. Configuration Done on Database

The MYSQL database has the flexibility of logging several logs which by default it enables only error logs. Apart from error logs, the other informative logs which needs to be enabled are query logs, and binary logs. The error logs would record diagnostic messages such as errors, warnings and notes that are encountered during the server startup, shutdown and while the server is running. The query logs would record the connection establishments and statements received from clients.

For enabling such query logs, the database configuration file will be included with the following lines:

```
general_log = 1
general_log_file = "mysql_query_log"
```

Another most significant log file which needs to be enabled are the binary logs. These binary logs would hold the information regarding the events that made changes to the database such as insert, delete, alter and modify. Enabling these binary logs would slow the server, but these logs would purely be helpful during the attack investigation process.

For enabling binary logs the configuration file will be configured with this line:

```
log-bin = "C:/xampp/mysql/data/binlog/bin-log"
```

Upon enabling the binary logs, two files will be created in the above specified path. The file with bin-log.000001 extension is the actual file used for log analysis. This binary file would be helpful during data recovery operations that is if suppose after a backup has been restored, the events in the binary log that were recorded after the backup was made are re-executed. These events bring databases up to date from the point of the backup.

The general nature of the attacker performing SQL Injection can be predicted from the above enabled logs.

Most of the regular expressions in the javascript code of web application don't validate all the user input and are thus susceptible to SQLinjection. If a Web application uses cookie's contents to build SQL queries, then attackers can take this opportunity to modify cookies and submit to the database engine. [2]

7. Results

Performing SQL injection on a vulnerable web application:

The userid field of session-input page is given with the following string by the attacker, as shown in fig 3.
1' and 1=1 union select database(), users()#

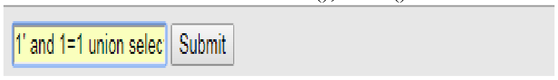
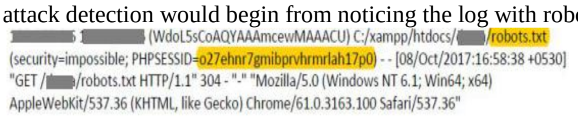


Fig 3: Malicious string execution on vulnerable input field.

Upon execution of such malicious query by the attacker, the log analysis will be performed in the following way
The configuration done on snort would detect the kind of malicious string the attacker actually used. A pcap file of SQLinjection attack is sniffed through wireshark and is given as input to the snort application. Then the file bearing the newly configured rules will be applied for detecting the malicious SQL strings used by the attacker. Based on the attacker's usage of malicious string the web server log analysis is performed.
The source and destination ip addresses, and the name of the hosted website were being darkened for security purposes.

7.1 Web Server Log Analysis

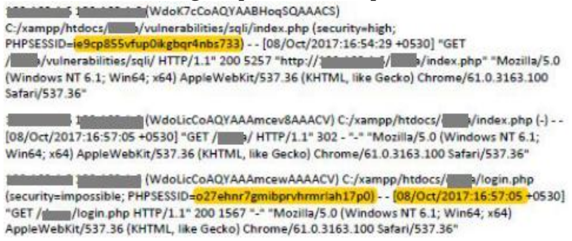
Firstly, the malicious activity would be detected if the attacker unauthorizedly accesses any file that is listed in robots.txt. This is the common file which will be maintained by every web application which would contain a list of disallowable files to the user. So, the attacker analysis begins from this point. As shown in fig 4 the attack detection would begin from noticing the log with robots.txt.



```
[redacted] [redacted] (WdoL5sCoAQYAAAmcewMAAACU) C:/xampp/htdocs/[redacted]/robots.txt
(security=impossible; PHPSESSID=o27ehnr7gmibprvhrmrlah17p0) - - [08/Oct/2017:16:58:38 +0530]
"GET /[redacted]/robots.txt HTTP/1.1" 304 - "-" "Mozilla/5.0 (Windows NT 6.1; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/61.0.3163.100 Safari/537.36"
```

Fig 4: Identifying the log bearing robots.txt file

Based on the log of robots.txt, the session id should be extracted. As the session id will be unique for every user so it is considered to know the time of the session creation. So, from the time of its creation till its expiration the whole activity of the attacker can be predicted. So, as shown in fig 5 the session id that should be tracked is o27ehnr7gmibprvhrmrlah17p0.

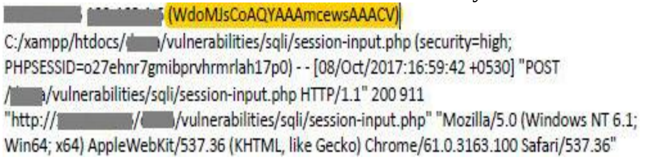


```
[redacted] [redacted](WdoK7cCoAQYAABHoqSQAAACS)
C:/xampp/htdocs/[redacted]/vulnerabilities/sqli/index.php (security=high;
PHPSESSID=ie9cp855vfup0ikgbqr4nbs733) - - [08/Oct/2017:16:54:29 +0530] "GET
/[redacted]/vulnerabilities/sqli/ HTTP/1.1" 200 5257 "http://[redacted]/[redacted]/index.php" "Mozilla/5.0
(Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/61.0.3163.100
Safari/537.36"

[redacted] [redacted](WdoLicCoAQYAAAmcev8AAACV) C:/xampp/htdocs/[redacted]/index.php (-) - -
[08/Oct/2017:16:57:05 +0530] "GET /[redacted]/ HTTP/1.1" 302 - "-" "Mozilla/5.0 (Windows NT 6.1;
Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/61.0.3163.100 Safari/537.36"

[redacted] [redacted] (WdoLicCoAQYAAAmcewAAAACV) C:/xampp/htdocs/[redacted]/login.php
(security=impossible; PHPSESSID=o27ehnr7gmibprvhrmrlah17p0) - - [08/Oct/2017:16:57:05 +0530]
"GET /[redacted]/login.php HTTP/1.1" 200 1567 "-" "Mozilla/5.0 (Windows NT 6.1; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/61.0.3163.100 Safari/537.36"
```

Fig 5: Extracting the attacker's session id

So, the log having the session id o27ehnr7gmibprvhrmrlah17p0 is found with its existence and came to know that the session is created at the time 08/Oct/2017:16:57:05.

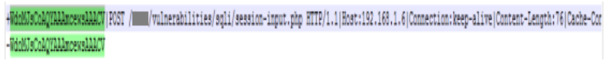
In case of a server crash by the attacker, even that can be analysed through the forensic logs of apache where every action performed by the attacker will be having a unique forensic id which have '+' and '-'signs. The positive sign forensic id represents the start of the action and the negative sign forensic id represents the end of the action. From the fig 5 the forensic id enabled is in the third field. As shown in fig 6, log bearing the POST method is extracted so that the forensic id is recorded for further analysis.



```
[redacted] [redacted] (WdoMJsCoAQYAAAmcewsAAACV)
C:/xampp/htdocs/[redacted]/vulnerabilities/sqli/session-input.php (security=high;
PHPSESSID=o27ehnr7gmibprvhrmrlah17p0) - - [08/Oct/2017:16:59:42 +0530] "POST
/[redacted]/vulnerabilities/sqli/session-input.php HTTP/1.1" 200 911
"http://[redacted]/[redacted]/vulnerabilities/sqli/session-input.php" "Mozilla/5.0 (Windows NT 6.1;
Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/61.0.3163.100 Safari/537.36"
```

Fig 6: Extracting forensic id from web server logs.

As shown in fig 7 the forensic log file is checked with the id bearing WdoMJsCoAQYAAAmcewsAAACV.



```
+WdoMJsCoAQYAAAmcewsAAACV|POST /[redacted]/vulnerabilities/sqli/session-input.php HTTP/1.1|Host:192.168.1.6|Connection:keep-alive|Content-Length:76|Cache-Cor
-WdoMJsCoAQYAAAmcewsAAACV
```

Fig 7: Extracting respective forensic id from forensic log file.

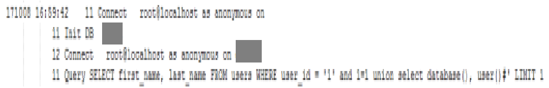
Since negative sign is also logged then it implies that the attacker action has completed without any server crash.

If the attacker has used proxy with an intention of hiding his ip address then in such situation, the web server would log with CONNECT method indicating that the client has used proxy.

Even status codes of the web server log would be used to detect whether or not the application is vulnerable to SQL injection. If 200 response code comes back without any response body then it is likely that the application has processed the request without any authentication or authorization and so it is detected that the application is vulnerable to SQL injection.

7.2 Database Log Analysis

The malicious SQL strings posted by the attacker can be seen in database logs. The entire query of the attacker will be logged in query logs of MYSQL database as shown in fig 8.



```
171008 16:59:42    11 Connect   root@localhost as anonymous on
                   11 Init DB [redacted]
                   12 Connect   root@localhost as anonymous on [redacted]
                   11 Query SELECT first_name, last_name FROM users WHERE user_id = '1' and 1=1 union select database(), user()#' LIMIT 1
```

Fig 8: Extracting the malicious query log in database logs.

The time that is recorded from the web server logs will be checked for its existence in database logs. The above screenshot shows the time 16:59:42 and date, which is the exact time that is extracted from web server logs. The malicious query executed by the attacker can be seen from the above screenshot which is 1' and 1=1 select database(), user()#. So, the query executed at the backend will be of the form

Query SELECT first_name, last_name FROM users WHERE user_id = '1' and 1=1 union select database (), user()#'

8. Finite Automata

The sequence of states undergone by the attacker while performing the SQL injection attack is represented in the below fig 9. At the end of every malicious string # is included so that the rest of the query which is to be executed by the database will be made to comment and, so it will not be executed.

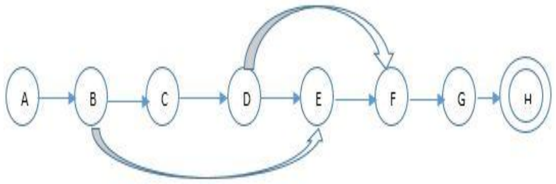


Figure 9: Finite Automata

A- Version of the database using the below string

$’ or 0=0 union select 2, version#

B- Database name using the below string

$’ or 0=0 union select database(), null#

C- Database user using the below string

$’ or 0=0 union select null, user()#

D- Number of columns in a table

-$’ or 0=0 order by 1#

-$’ or 0=0 order by 2#

-$’ or 0=0 order by 3#

This string must be executed continuously until the database throws an error as “unknown column n in order clause” which shows that there are only n-1 columns in the database.

The id value should be prefixed with a negative sign so that all the compromised records will display straight away on the web page.

E- Table name using the below string

$’ and 1=1 union select table_name, null from information_schema.tables#

information_schema is a metadata table containing a list of all the table names and column names by default in the database.

F- Column names using the below string

$’ and 1=1 union select column_name, null from information_schema.columns#

G- Using the table name and the column name we can find the password column and its table name. So, the column containing password is retrieved using the below string

$’ and 1=1 union select null, password from users#

H- The passwords displayed will be in hash format, where on, it will be decrypted either using MD5 or SHA algorithms.

9. Conclusion

The paper has presented the detection methodology followed when any SQL injection strings hits the server. The proposed relative approach and their respective log analysis would extremely be useful in detecting the SQL injection strings no matter how vast the size of attack log file is. This paper would suggest to perform the appropriate code review and let not such malicious SQL strings be used while coding a software.

References:

[1] SANS Institute Infosec Reading room
[2] Muhammad Saidu Aliero, Abdulhamid Aliyu Ardo, Imran Ghani, Mustapha Atiku
[3] Classification of SQL injection detection and prevention measure.
[4] Hussein Alnabulsi , Md Rafiqul Islam , Quazi Mamun
[5] Detecting SQL injection using snort IDS
[6] S. Eckmann, "Translating Snort rules to STATL scenarios", In Proc. Recent Advances in Intrusion Detection, pp. 1-13, October 2001
[7] Tautology based Advanced SQL Injection Technique A Peril to Web Application
[8] Kritarth Jhala Shukla Umang Chad Dougherty, “Practical identification of SQL injection vulnerabilities”