A NOVEL TECHNIQUE TO DETECT AND PREVENT SQL INJECTION ATTACKS USING BITAP STRING MATCHING ALGORITHM

N.KARTHIKEYAN,
Assistant Professor in CSE,
Government College of Engineering,
Thanjavur,

R.VIVEKANANDAN,
Assistant Professor in Civil Engineering,
Government College of Engineering,
Thanjavur,

M.SAKTHIVEL
B.E Computer Science and
Engineering (Final Year),
Government College of Engineering,
Thanjavur,

N.DINESH
B.E Computer Science and
Engineering (Final Year),
Government College of Engineering,
Thanjavur.

Abstract

SQL injection attack (SQLIA) is among the most common security threats to web-based services that are deployed on cloud. By exploiting web software vulnerabilities, SQL injection attackers can run arbitrary malicious code on target databases to acquire or compromise sensitive data. Although web application firewalls (WAFs) are offered by most cloud service providers, tenants are reluctant to pay for them. Therefore, measures must be put in place to curtail the growing threats of SQL injection and XSS attacks. This study presents a technique for detecting and preventing these threats using Bitap string matching algorithm. The algorithm was used to match user's input string with the stored pattern of the injection string in order to detect any malicious code. The implementation was carried out by using Flask Framework. The security cloud service will be provided to the tenants. The tenant will use this cloud service and block the SQL injection attack. The security level of the technique was measured using different test cases of SQL injection, cross-site scripting (XSS). Results obtained revealed that the proposed technique was able to successfully detect and prevent the attacks, log the attack entry in the database, send mail to the tenants and also generate a warning message. Therefore, the proposed technique proved to be more effective in detecting and preventing SQL injection and XSS attacks.

Keywords --SQL injection, Cross-site scripting, Information security, Web application vulnerability, Bitap string matching algorithm

1. INTRODUCTION

Due to the outstanding merits (e.g., continuous availability, easy accessibility, flexibility) of world wide web, an increasing number of enterprises and individuals from different business sectors, such as online shopping, e-banking, healthcare, e-government, and social media have made their services available on the web. This shift will be further accelerated due to the maturity of the latest web technologies (e.g., HTML5), which enable richer web applications and more enhanced user experiences. Along with the prosperity of web applications, inevitably they are becoming the main targets of malicious attackers. It is reported that in the third quarter of 2017, a web application experiences on average 500–700 attacks per day.

As one of the most serious threats to web applications, SQL Injection Attacks (SQLIAs) are widely used by attackers to obtain unauthorized access to sensitive information. As an example, a forum of the popular multiplayer online game "Dota 2" was attacked by some SQL injection on July 10, 2016. It raises serious concerns about user privacy, since around 2 million player records were exposed, including email addresses, IP addresses, usernames, user IDs, and corresponding passwords. Although the user passwords of this forum were encrypted using the MD5 hashing algorithm, the original passwords can be easily figured out.

Web applications use data supplied by users in SQL queries, attackers can manipulate these data and insert SQL meta-characters into the input fields so as to access, modify, or delete the content of the database. For instance, the WHERE clause in the SQL query SELECT * FROM login WHERE password = admin could be manipulated when hackers supply inputs like 'anything' OR '7' = '7'; #. The WHERE clause now contains two conditions separated with the logical operator OR. The first condition might not be TRUE, but the second condition must be TRUE because 1 is always equals 1, and the logical operator "OR" returns TRUE if either or both of the conditions are TRUE. Hence, the attacker gains access without a need to know the password. Sometimes, wrong input values can also be supplied intentionally so that error messages that will help the attackers to understand the database schema will be revealed. Therefore, SQL injection is a serious threat for web application users.

1. 1. Web application architecture

Although a web application is simply recognized as a program running on a web browser, a web application generally has a three-tier construction as shown in Fig. 1. In Fig. 1, a presentation tier is sent to a web browser by request of the browser.

(1) Presentation Tier: This tier receives the user input and shows the result of the processing to the user. It can be thought of as the Graphical User Interface (GUI). Flash, HTML, Java script, etc. are all part of the presentation tier, which directly interacts with the user. This tier is analyzed by a web browser.

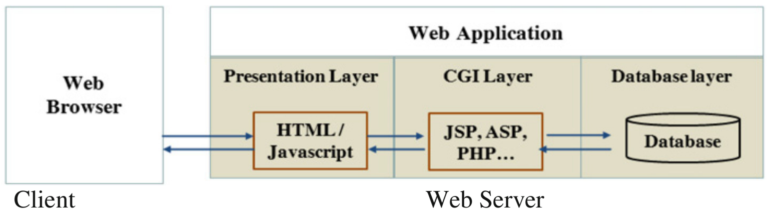


Fig. 1. Web application architecture

(2) CGI Tier: Also known as the Server Script Process, this is located in between the presentation and database tiers. The data inputted by the user is processed and the result is sent to the database tier. The database tier sends the stored data back to the CGI tier, and it is finally sent to the presentation tier to be viewed by the user. Therefore, data processing within the web application is performed at the CGI Tier and can be programmed in various server script languages such as JSP, PHP, ASP, etc.

(3) Database Tier: This tier only stores and retrieves all of the data. All sensitive web application data are stored and managed within the database. Since this tier is directly connected to the CGI tier without any security check, data in the database can be revealed and modified if an attack on the CGI tier succeeds.

1. 2. SQL injection attacks

The SQL injection attacks are:

1.2.1. Tautology or Boolean based attack:

Tautology is a formula which is based on True or False values. The malicious SQL query forces the web application to return a different result depending on whether the query returns a TRUE or FALSE. Tautology-based SQL injection attacks are usually bypass user authentication and extract data by inserting a tautology in the WHERE clause of a SQL query. The query transforms the original condition into a tautology, causes all the rows in the database table are open to an unauthorized user. A typical SQL tautology has the form "or <comparison expression>", where the comparison expression uses one or more relational operators to compare operands and generate an always true condition. If an unauthorized user input user id as 'admin' and password as 'anything' or 'x'='x' then the resulting query will be:

SELECT * FROM login WHERE username = 'admin' AND password = 'anything' or 'x'='x'

1.2.2. Like based attack:

This injection type is used by attackers to impersonate a particular user using the SQL keyword LIKE with a wildcard operator (%). For example, an attacker can inject input: 'anything' OR username LIKE ‘S%’; # instead of a username to have SQL query: SELECT * FROM login WHERE username =’ anything OR username LIKE ‘S%’; #”. The LIKE operator implements a pattern match comparison, that is, it matches a string value against a pattern string containing wildcard character. The query searches the user’s table and returns the records of the users whose username starts with letter S. The wildcard operator (%) means zero or more characters (S…), and it can be used before or after the pattern.

1.2.3. Comment based attack

Comments injected into an application through input can be used to compromise a system. As data is parsed, an injected/malformed comment may cause the process to take unexpected actions that result in an attack. If the attacker has the ability to manipulate queries which are sent to the database, then they’re able to inject a terminating character too. For example, **SELECT * FROM login WHERE username=’’- AND password=’’**

1.2.4. Union based attack

This type of attack can be done by inserting a UNION query into a vulnerable parameter which returns a dataset that is the union of the result of the original first query and the results of the injected query. The SQL UNION operator combines the results of two or more queries and makes a result set which includes fetched rows from the participating queries in the UNION. The attacker who tries to use this method must have solid knowledge of DB schema. For example, in the SQL query "SELECT ∗ FROM customers WHERE password = 123 UNION SELECT creditCardNo, pin FROM customers" the attacker injects the SQL statement "123 UNION SELECT creditCardNo, pin FROM customers" instead of the required password. The query therefore exposes all the credit card numbers with their PINs from the customer’s table.

1.3 Cross-site scripting (XSS) attacks

Cross site scripting (XSS) is a common attack vector that injects malicious code into a vulnerable web application. XSS differs from SQL injections in that it does not directly target the application itself. Instead, the users of the web application are the ones at risk. A successful cross site scripting attack can have devastating consequences for an online business’s reputation and its relationship with its clients. Depending on the severity of the attack, user accounts may be compromised, Trojan horse programs activated and page content modified, misleading users into willingly surrendering their private data. Finally, session cookies could be revealed, enabling a perpetrator to impersonate valid users and abuse their private accounts.

XSS vulnerabilities have been categorized into three categories which are reflected, stored, and Document Object Model (DOM)-based. DOM-based vulnerabilities occur when active contents on a web page (mostly JavaScript) accept user inputs which are malicious thereby causing the execution of injected code. Stored XSS vulnerabilities occur when inputs collected via web applications are malicious and stored in the database for immediate or future use. It is one of the most dangerous of all XSS vulnerabilities because in as much as it is in the database, the hacker can manipulate the contents of the database. Reflected XSS vulnerabilities are different from other XSS vulnerabilities because it attacks clients who accesses or loads a malicious URL. Though several techniques aimed at curtailing the growing hazards of these attacks have been reported in literature, many have not been able to fully address all scope of the problem. Several security techniques have been proposed towards preventing data and information from unauthorized attacks, and attackers continually devise new security vulnerabilities that could be exploited. Therefore, new techniques aimed at detecting and preventing these attacks are essential.

```
<input type=”search” value=”potatoes”/>

<input type=”search” value=” Attacker”/><script>StealCredentials()</script> “/>
```

2. REVIEW OF LITERATURE

As documented in literature, preventing SQL injection vulnerabilities as well as XSS attacks has been mostly achieved through the use of data encryption algorithms, PHP escaping functions, pattern matching algorithms, and through instruction set randomization. Authors in [1] employed a novel based approach to detect and prevent SQL injection and XSS attacks by using KMP string matching algorithm. The various types and patterns of the attacks were studied, then a parse tree was designed to represent the patterns. Based on the identified patterns, a filter () function was formulated using the KMP string matching algorithm. The formulated filter () function detects and prevents any form of SQL injection and XSS attacks. Every input string is expected to pass through this filter () function. If at least one function returns True, then, the filter () function will block that user, reset the HTTP request, and display a corresponding warning message. The proposed technique can successfully detect and prevent the attacks, log the attack entry in the database, block the system using its Mac Address to prevent further attack, and issue a blocked message.

Authors in [2] employed an adaptive algorithm to prevent SQL injection. The technique is based on comparing, at run time, the parse tree of the SQL statement before inclusion of user input with that resulting after inclusion of input. The proposed method consists of the best features of both parse tree validation technique and code conversion method. In this method, parse the user input and check whether its vulnerable, if there is any chance of vulnerability present then code conversion will be applied over that input. In this way, proposed technique can detect and prevent SQL Injection using a single code. In proposed technique, code conversion to each and every user input is more time consuming as well as the database size will also increase. Parse tree validation technique will raise false alarm even if legitimate user is having blank space in his/her input.

Authors in [3] employed a novel hybrid technique that detects and prevents all types of SQLIAs in different system categories regardless of the system development language or the database engine. The suggested hybrid technique is done in two main phases: runtime analysis, and static analysis. The first phase is a dynamic/runtime analysis method that depends on applying tracking methods to trace and monitor the execution processes of all received queries. The next phase is a static analysis phase that is performing a string comparison between the received SQL queries and previous expected SQL queries to prevent any query that is described as a suspicious query. The proposed simulation technique can detect and prevent all types of SQLIAs. The drawback of proposed technique is that it takes lot of time delay when the database recovery operation is performing after the SQLIA is detected.

Authors in [4] employed a novel method for detecting SQL injection attacks by comparing static SQL queries with dynamically generated queries after removing the attribute values. The proposed method is a novel based method to detect SQL injection attacks based on static and dynamic analysis. This method removes the attribute values of SQL queries at runtime (dynamic method) and compares them with the SQL queries analyzed in advance (static method). The proposed method simply removes the attribute values in SQL queries for analysis, which makes it independent of the DBMS. Complex operations such as parse trees or particular libraries are not needed in the proposed method.

Authors in [5] employed how to prevent the various types of SQLIAs using tools. The proposed method investigated SQL injection detection and prevention tools. After that it compared these tools in terms of their ability to stop SQLIA. In addition, these tools were compared based on deployment requirement (modifying source code, additional infrastructure and automation of detection or prevention) and common evaluation parameters (efficiency, effectiveness, stability, flexibility and performance). This emphasizes on tools not on techniques. Thirteen prevention tools were listed and analyzed.

3 PROPOSED SYSTEM

With a view to come up with a technique that could detect and prevent the various forms of SQLinjection and XSS attacks, the patterns for each attack were studied, and solutions were proffered based on these patterns. The methodology employed in this study is in six phases:formation of SQL injection and string patterns, compare user input with patterns by using Bitapalgorithm, detecting SQL-injection and XSS attacks, send mail to tenants, preventing SQL injection and XSS attacks using Bitap algorithm, and formulating the filter functions.

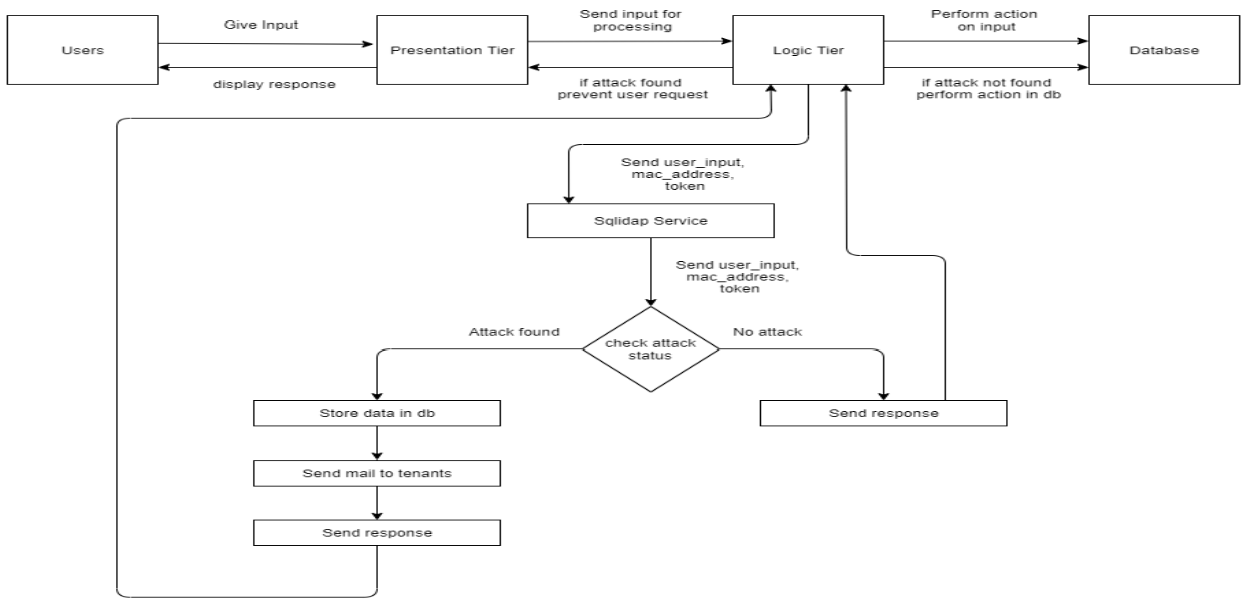


Fig. 2. Work Flow Diagram

Table 1 Different forms of injection code with their common patterns

S.NO.	Injection type	Common pattern	Example
1	Boolean-based	’ OR ‘…’ = | > | > =| < | < =|<>|! = ‘…’;#	‘ OR ‘1’ = ‘1’;# 123’ OR ‘a’ <> ‘b’ ;# ‘ OR ‘2 + 3’ < = ‘10’ ;#
2	Like-based	’OR … LIKE ‘…%’;#	‘ OR username LIKE ‘S%’#
3	Comment-based	’-	‘-
4	XSS	<script> …’…;</script>	<script>alert(‘Xss’);</script>
5	Union-based	‘ union select … from …;#	‘ union select * from users; # ‘ union select name from a;#

3.1 Formation of SQL injection string patterns

Every form of attacks has certain characters and keywords that hackers do manipulate to perpetuate their attacks. These characters and keywords are used to form malicious codes that are used to carry out the various forms of attacks. Identifying these injection codes will help in coming up with how to detect and prevent these attacks. The injection codes common to the various forms of attacks are provided in Table 3.

3.2 Compare user input with patterns by using Bitap Algorithm

The Bitap Algorithm is astring-matching algorithm. The algorithm tells whether a given text contains a substring which is “equal” to a given pattern.The algorithm begins by precomputing a set of bitmasks containing one bit for each element of the pattern. This does most of the bitwise operations, which are quick.The Bitap Algorithm is also known as shift-or, shift-and, or BYG Algorithm.

```
defmatch_string(self, text, pattern):
    # Length of pattern
    m = len(pattern)
    # Initialize ~1
    A = ~1
    # If length is 0 or exceeds 63 then return "No match"
    if m==0:
        return -1
    elif m>63:
        #print("Pattern is too long")
        return -1

    p_mask = [~0 for i in range(300)] # Preparing bit masks
    #Taking the pattern as index of PATTERN MASK then apply AND operator with complement
    of 1 by LEFT SHIFT by i times
    for i in range(m):
        p_mask[ord(pattern[i])] &= ~(1<<i)
    # Apply OR with pattern_mask
    for i in range(len(text)):
        A |= p_mask[ord(text[i])]
        A <<= 1
        if (A&(1<<m))==0:
            return i-m+1
    return -1
```

3.3 Detecting SQL injection and XSS attacks

The various types of SQL injection and XSS attacks were detected thus:

3.3.1. Boolean-based SQL-injection attack:

As presented in Table 1, it was deduced that most Boolean-Based SQL injection strings have a single quote (‘) followed by logical operator OR and a true statement such as ‘1’ = ‘1’;#, ‘a’ <> ‘b’ ;# , ‘2 + 3’ < = ‘10’ ;#

3.3.2. Like-based SQL injection attack:from Table 1, it shows the different forms of like-based SQL injection attack, and it is detected when the input string contains a single quote (‘) followed by the logical operator OR, followed by one or more identifiers, followed by the SQL keyword LIKE, followed by a single quote (‘), followed by the wildcard operator (%), followed by a single quote (‘), followed by semicolon with hash. Example include ‘OR username LIKE ‘S%’# and ‘OR password LIKE ‘%2%’;#

3.3.3. Comment based attack: As in Table 1, it shows the comment based attack, and it is detected when the input string contains quotation character (‘ ’ ’)comment character (‘--‘) followed by any string. For example, ‘-

3.3.4. XSS attack:this can be detected when a JavaScript open tag “<script>” is encountered from the input string, followed by zero or more characters and/or a single quote (‘), followed by a JavaScript closing tag “</script>” as in <script>alert(‘XSS’);</script>. If it were to be encoded XSS attack, such will have a JavaScript open tag “<script>” followed by one or more ASCII code, hexadecimal number, HTML name, or HTML number of a character and/or a single quote (‘), followed by a JavaScript closing tag “</script>” as in <script>alert(&#34; XSS &#34;);</script>.

3.3.5 Union-based SQL injection attack: Also, most union-based SQL injection strings have a single quote (‘) followed by a UNION keyword, the SQL keyword SELECT, one or more identifiers, the SQL keyword FROM, one or more identifiers then a semicolon (;) with hash (#). Example includes ‘union select * from users; # or ‘ union select name from a; #.

3.4. Preventing SQL-injection and XSS attacks using Bitap algorithm

Bitap string matching algorithm was used to compare user’s input string with different SQL injection and XSS attacks patterns that have been formulated. The algorithm goes thus:

```
boolean_attack = boolean()
comment_attack = comment()
union_attack = union()
like_attack = like()
xss_attack = xss()

if(boolean_attack.check_boolean_based_attack(args['user_input'])):
        self.store_attack(args['mac_address'], "Boolean Based Attack", args['user_input'])
        returnreturn_response().get_response(True, "Boolean Based Attack", args['user_input'])

elif(union_attack.check_union_based_attack(args['user_input'])):

        self.store_attack(args['mac_address'], "Union Query Attack", args['user_input'])
        returnreturn_response().get_response(True, "Batch Query Attack", args['user_input'])

elif(like_attack.check_like_based_attack(args['user_input'])):
        self.store_attack(args['mac_address'], "Like Based Attack", args['user_input'])
        returnreturn_response().get_response(True, "Like Based Attack", args['user_input'])

elif(xss_attack.check_xss_based_attack(args['user_input'])):
        self.store_attack(args['mac_address'], "XSS Attack", args['user_input'])
        returnreturn_response().get_response(True, "XSS Attack", args['user_input'])
elif(comment_attack.check_comment_based_attack(args['user_input'])):

        self.store_attack(args['mac_address'], "Comment Based Attack", args['user_input'])
        returnreturn_response().get_response(True, "Comment Based Attack", args['user_input'])

else:
        returnreturn_response().get_response(False, "No attack")
```

3.5 Formulating the filter functions

The filter () function was formulated to prevent SQL injection and XSS attacks. This function contains other functions that have been written each to detect a particular form of attack. If at least one function returns True, then, the filter () will block that user, and display a corresponding warning message.

3.5.1 Boolean-based SQL-injection attacks

Formulating the check_boolean_based_attack () function: this was used to prevent Boolean-based SQL injection attack:

```
defcheck_boolean_based_attack(self, input):
inj_pattern = ["'", "'", "'", "'", "'", "#"]
logical_operators = ["or", "||"]
relational_operators = ["=", ">", ">=", "<", "<=", "<>", "!="]
bitap = bitap_algorithm()
        for i in range(len(inj_pattern)):
ifbitap.match_string(input.lower(), inj_pattern[i])>-1 :


              if i==0:
                  counter = 0
                  for j in range(len(logical_operators)):
ifbitap.match_string(input.lower(), logical_operators[j])>-1 :
                          counter = counter + 1
                  if counter==0:
                      result = False
                      break
             if i==2:
                  counter = 0
for k in range(len(relational_operators)):
ifbitap.match_string(input.lower(), relational_operators[k])>-1 :
                          counter = counter + 1
                  if counter==0:
                      result = False
                      break
               if i+1 == len(inj_pattern):
                  result = True
           else:
               result = False
               break
```

3.5.2. Like-based SQL injection attack

Formulating the check_like_based_attack ( ) function: this was used to prevent Like -based SQL injection attack:

```
defcheck_like_based_attack(self, input):
inj_pattern = ["'", "like", "'", "%", "#"]
```

```
logical_operators = ["or", "||"]
bitap = bitap_algorithm()
        for i in range(len(inj_pattern)):
ifbitap.match_string(input.lower(), inj_pattern[i])>-1:
                if i==0:
                    counter = 0
                    for j in range(len(logical_operators)):
if  bitap.match_string(input.lower(), logical_operators[j])>-1:
                            counter += 1
                    if counter ==0:
                        result = False
                        break
                if i+1==len(inj_pattern):
                    result = True
            else:
                result = False
                break
```

3.5.3. Comment based attack

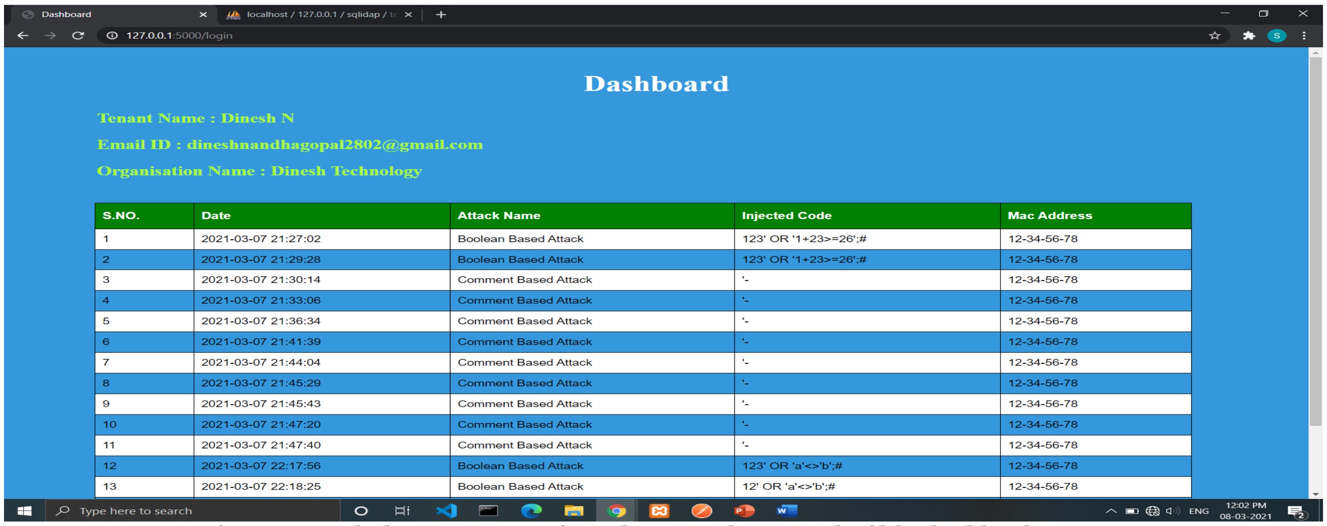
Formulating the check_comment_based_attack () function: this was used to prevent Comment based SQL injection attack:

```
defcheck_comment_based_attack(self, input):
inj_pattern = ["'", "-"]
bitap = bitap_algorithm()
        for i in range(len(inj_pattern)):
ifbitap.match_string(input, inj_pattern[i])>-1 :
                if i+1 == len(inj_pattern):
                    result = True
            else:
                result = False
                break
```

3.5.4. XSS attack

Formulating the check_xss_based_attack( ) function: this was used to prevent XSS SQL injection attack:

```
defcheck_xss_based_attack(self, input):
inj_pattern = ["<script>", "'", ";","</script>"]
bitap = bitap_algorithm()
        for i in range(len(inj_pattern)):
ifbitap.match_string(input.lower(), inj_pattern[i])>-1:
                if i+1==len(inj_pattern):
                    result = True
            else:
                result = False
                break
```



Dashboard

Tenant Name : Dinesh N

Email ID : dineshnandhagopal2802@gmail.com

Organisation Name : Dinesh Technology

S.NO.	Date	Attack Name	Injected Code	Mac Address
1	2021-03-07 21:27:02	Boolean Based Attack	123' OR '1+23>=26';#	12-34-56-78
2	2021-03-07 21:29:28	Boolean Based Attack	123' OR '1+23>=26';#	12-34-56-78
3	2021-03-07 21:30:14	Comment Based Attack	'-	12-34-56-78
4	2021-03-07 21:33:06	Comment Based Attack	'-	12-34-56-78
5	2021-03-07 21:36:34	Comment Based Attack	'-	12-34-56-78
6	2021-03-07 21:41:39	Comment Based Attack	'-	12-34-56-78
7	2021-03-07 21:44:04	Comment Based Attack	'-	12-34-56-78
8	2021-03-07 21:45:29	Comment Based Attack	'-	12-34-56-78
9	2021-03-07 21:45:43	Comment Based Attack	'-	12-34-56-78
10	2021-03-07 21:47:20	Comment Based Attack	'-	12-34-56-78
11	2021-03-07 21:47:40	Comment Based Attack	'-	12-34-56-78
12	2021-03-07 22:17:56	Boolean Based Attack	123' OR 'a'<>'b';#	12-34-56-78
13	2021-03-07 22:18:25	Boolean Based Attack	12' OR 'a'<>'b';#	12-34-56-78

Fig. 3. Attack detection interface showing the record of blocked hackers

Table 2 The test plan

S.NO.	Attack type	Sample injection code
1	Boolean-based SQLi	‘ OR ‘7’=’7’; #
2	Boolean-based SQLi	aaa‘ OR ‘1’=’1’; #
3	Boolean-based SQLi	‘ OR ‘a’<>’b’ ;#
4	Like-based SQLi	‘ OR password LIKE ‘%2%’;#
5	Like-based SQLi	‘ OR username LIKE ‘%e’;#
6	Like-based SQLi	a‘ OR username LIKE ‘S%’;#
7	Union-based SQLi	‘UNION select * from users; #
8	Union-based SQLi	‘UNION select cardNo, pin from customer; #
9	Union-based SQLi	‘UNION select * from login; #
10	Comment-based SQLi	‘-
11	Cross-site scripting	<script> alert(‘XSS‘) </script>
12	Cross-site scripting	<script>attackATM() </script>
13	Cross-site scripting	<script>attackUser() </script>

3.5.5. Union-based SQL injection attacks

Formulating the check_union_based_attack ( ) function: this was used to prevent Union-based SQL injection attack:

```
defcheck_union_based_attack(self, input):
inj_pattern = ["'", "union", "select", "from", "#"]
bitap = bitap_algorithm()
    for i in range(len(inj_pattern)):

ifbitap.match_string(input.lower(),inj_pattern[i])>-1 :
        if i+1 == len(inj_pattern):
            result = True
      else:
        result = False
        break
```

4. RESULTS AND DISCUSSION

The proposed technique was implemented using Flask Framework and Flask Server. The Python was selected as a scripting Language, because it is the efficient server-side scripting language in building database-driven web-based application. Flask Development Server was used to host the system during the test. The attack was launched on computers and mobile phones of different brand, processor speed, and RAM size using Windows, Linux, and Android operating systems. Opera (Version 74.0.3911.203), Google Chrome (version 89.0.4389.82), Mozilla Fire fox (Version 84.0) were used as browsers to launch the attack. The database to be targeted was stored on MySQL database of size 4.998 MB. The test was conducted on local WAMP/XAMPP server.

To test the proposed technique, a vulnerable web application shown in Fig. 3 was purposely developed. An attempt was made to submit various known SQL injection and XSS attack patterns using a "test plan" shown in Table 2. The test plan consists of test cases which contains input string for various attacks. The input strings were supplied via the input fields of the web application. When an attack was detected, the user would be blocked and The MAC address of the systems used to carry out the attack, types of attacks, the input strings supplied, time stamp, and hacking status was documented in a database. Based on the results obtained from the various attack attempts, the proposed technique was able to successfully detect and prevent all the attacks.

5. CONCLUSION

A novel approach to detect and prevent SQL injection and XSS attacks is presented in this paper. The various types and patterns of the attacks were first studied. Based on the identified patterns, a filter() function was formulated using the Bitap string matching algorithm. The formulated filter() function detects and prevents any form of SQL injection and XSS attacks. Every

input string is expected to pass through this filter () function. If at least one function returns True, then, the filter() function will block that user, and display a corresponding warning message and send mail to the tenants. The technique was tested using a test plan that consist of different forms of Boolean-based, union-based, comment-based, like-based and cross-site scripting attacks. The test results show that the technique can successfully detect and prevent the attacks, log the attack entry in the database, block the system using its Mac Address to prevent further attack, and issue a blocked message and send mail to the tenants. A comparison of the proposed technique with existing techniques revealed that the proposed technique is more efficient because it is not limited to a particular form of attack, and it can handle different forms of SQL injection and XSS attacks along with send notification to the tenants.

6. REFERENCES

[1] Oluwakemi Christiana Abikoye, AbdullahiAbubakar, Ahmed HarunaDokoro, Oluwatobi Noah Akande and AderonkeAnthoniaKayode, "A novel technique to prevent SQL injection and cross-site scripting attacks using Knuth-Morris-Pratt string match algorithm", Abikoye et al. EURASIP Journal on Information Security (2020) 2020:14.

[2] Ashish John, Ajay Agarwal, Manish Bhardwaj, "An adaptive algorithm to prevent SQL injection", American Journal of Networks and Communications 2015; 4(3-1): 12-15.

[3] Jalal Omer Atoum and AmerJibrilQaralleh, "A Hybrid Technique for SQL Injection Attacks Detection and Prevention", International Journal of Database Management Systems (IJDMS) Vol.6, No.1, February 2014.

[4] Inyong Lee, SoonkiJeong, Sangsoo Yeo, JongsubMoond, "A novel method for SQL injection attack detection based on removing SQL query attribute values", I. Lee et al. / Mathematical and Computer Modelling 55 (2012) 58–68

[5] AtefehTajpour, Suhaimi Ibrahim, Mohammad Sharifi, "Web Application Security by SQL Injection Detection Tools", International Journal of Computer Science Issues · March 2012