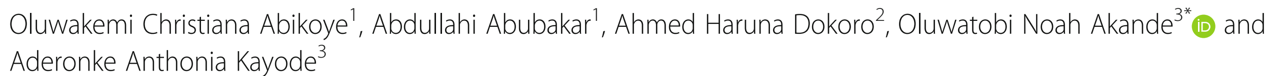
Abikoye et al. *EURASIP Journal on Information Security* (2020) 2020:14
https://doi.org/10.1186/s13635-020-00113-y

EURASIP Journal on
Information Security



RESEARCH Open Access

A novel technique to prevent SQL injection and cross-site scripting attacks using Knuth-Morris-Pratt string match algorithm

Oluwakemi Christiana Abikoye[1], Abdullahi Abubakar[1], Ahmed Haruna Dokoro[2], Oluwatobi Noah Akande[3*] and Aderonke Anthonia Kayode[3]

Abstract

Structured Query Language (SQL) injection and cross-site scripting remain a major threat to data-driven web applications. Instances where hackers obtain unrestricted access to back-end database of web applications so as to steal, edit, and destroy confidential data are increasing. 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 Knuth-Morris-Pratt (KMP) 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 using PHP scripting language and Apache XAMPP Server. The security level of the technique was measured using different test cases of SQL injection, cross-site scripting (XSS), and encoded injection attacks. Results obtained revealed that the proposed technique was able to successfully detect and prevent the attacks, log the attack entry in the database, block the system using its mac address, 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, Knuth-Morris-Pratt (KMP) string matching algorithm

1 Introduction

Internet is fast becoming a household technology with 4.39 billion users in January 2019 compared to 3.48 billion users in January 2018 [1]. This showed that more than one million new users got connected daily. This growth rate is being facilitated by data-driven web applications and services which enable users to transact their online activities with ease. Most modern organizations and individuals heavily rely on these web applications to reach out to their numerous customers. Users' inputs via web applications are used to query back end databases so as to provide the needed information. This trend has therefore opened up web applications and services to attacks by hackers. Moreover, the popularity of web application in social networking, financial transaction, and health problems are increasing very rapidly; as a result, software vulnerabilities are becoming very critical issues, and thus, web security has now become a major concern [2]. The vulnerabilities are mostly application layer vulnerabilities such as domain name server attacks, Inline Frame flaws, remote file inclusion, web authentication flaws, remote code execution, XSS, and SQL injection [3, 4]. A survey carried out by Open Web Application Security Project (OWASP) identified top 10 vulnerabilities as at June 2019 to be injection flaws, broken authentication and session management, sensitive data exposure, XML external entity, broken access control, security misconfiguration, XSS, insecure deserialization, using components with known

* Correspondence: akande.noah@lmu.edu.ng
[3]Computer Science Department, Landmark University, Kwara State, Omu-Aran, Nigeria
Full list of author information is available at the end of the article

© The Author(s). 2020 **Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

vulnerabilities, insufficient logging, and monitoring. However, among these forms of attacks, XSS and SQL injection have been identified as the most dangerous [5]. The WordPress Security Learning Center also submits that if SQL injection and XSS vulnerabilities could be handled in a code, then 65% vulnerabilities has been eliminated. Since web applications use data supplied by users in SQL queries, hackers 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 users WHERE password = 1234* could be manipulated when hackers supply inputs like 'anything*' OR* '1*'* = *'*1*'; #.* 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 hacker 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 Cross-site scripting (XSS) attacks

XSS is another similar attack where hackers prepare and execute a fragment of JavaScript in the security context of the targeted domain thereby incorporating malicious contents into web pages presented by a trusted web application. Most web applications that do not properly screen user input before loading web pages are susceptible to XSS attacks. Once a site has been affected, users could be redirected to automatically open malicious websites, the entire user session could be hijacked, and users' login details could also be stolen. Since the content is claimed to be from a trusted server, it is processed like normal contents. For example, the pseudo code below shows how latest comments are displayed on a website using a simple sever-site script:

$$echo\ "<html><body>";$$

$$echo\ "<h1>Most\ recent\ comments</h1>";$$

$$echo\ database.latestComment;$$

$$echo\ "</html></body>";$$

The scripts assume that the comments consist of only text. However, since the user's input is directly included, an attacker can submit his comment as "<script> doSomethingEvil();</script>". Therefore, users who visit the page will receive the following response:

$$echo\ "<html><body>";$$

$$echo\ "<h1>Most\ recent\ comments</h1>";$$

$$echo\ "<script>doSomethingEvil();</script>";$$

$$echo\ "</html></body>";$$

When the user's browser loads the page, it executes whatever JavaScript is contained inside the *<script>* tags. In this case, the attacker can write a JavaScript function that steals the victim's session cookie. This session cookie can be used to impersonate the victim subsequently.

XSS vulnerabilities have been categorized into three categories which are reflected, stored, and Document Object Model (DOM)-based [3]. 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 at will [1]. 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 [6–8], and attackers continually devise new security vulnerabilities that could be exploited. Therefore, new techniques aimed at detecting and preventing these attacks are essential.

1.2 SQL injection attacks

SQL-injection attacks could be in six categories:

a) Boolean-based SQL injection or tautology attack:

Boolean values (True or False) are used to carry out this type of SQL injection. The malicious SQL query forces the web application to return a different result depending on whether the query returns a TRUE or FALSE result. For instance, "$aaa\ OR\ 2=2$" has been inserted into SQL query "SELECT ∗ FROM users WHERE password = aaa OR 2 = 2" as the password so as to alter the structure of the *WHERE* clause of the

original query. This yields a SQL query with two different conditions separated with a logical operator *OR*. The first condition "password = aaa" might not be true, but the second condition "2 = 2" must be true. Therefore, the logical operator *OR* returns true if at least one of the operand is true thereby forcing the web application to return a different result.

b) Union-based SQL injection: this is the most popular of all the SQL injections. It uses the UNION statement to integrate two or more select statements in a SQL query thereby obtaining data illegally from the database. For instance, 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.
c) Error-based SQL injection: this is the simplest of all the SQL injection vulnerabilities; however, it only affects web applications that use MS-SQL Server. The most common form of this vulnerability requires an attacker to supply an SQL statement with improper input causing a syntax error such as providing a string when the SQL query is expecting an integer. For example, the SQL query: *SELECT * FROM customer WHERE pin = convert (int, (SELECT firstName FROM customer LIMIT 1))* tries to convert the first name of the first customer in the customer's table into integer type which is not possible. As a result, it causes the database server to throw an error containing the information about the structure of the table.
d) Batch query SQL injection/piggy backing attacks: this form of injection is dangerous as it attempts to take full control of the database. An attacker terminates the original query of the application and injects his own query into the database server. For instance, considering the SQL query: aaa; INSERT INTO users VALUES ('Abubakar', '1234');#, the first semicolon (;) terminates the original query, and query adds the username "Abubakar " and password "1234" to the users table ,and the hash (#) comments out the remaining query so that it will not be executed by the server. However, this form of attack works on only SQL-Server 2005, because it is the only server that accepts multiple queries at a time.
e) Like-based SQL injection. This injection type is used by hackers to impersonate a particular user using the SQL keyword *LIKE* with a wildcard operator (%). For instance, an attacker can inject input: "anything OR username LIKE 'S%' ;# instead of a username to have SQL query: SELECT * FROM users 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.
f) Hexadecimal/decimal/binary variation attack (encoded injection): in this type of injection, the hacker leverages on the diversity of the SQL language by using hexadecimal or decimal representations of the keywords instead of the regular strings and characters of the injection code. For instance, the traditional SQL injection code: *UNION SELECT * FROM users; #* could be replaced with:

'9…xsyx2ƒiv

ig„2B2p‚yw

2ėĕāĔĕY5'

Therefore, SQL injection vulnerability is a serious attack that must be prevented. Its different categories have further revealed that a prevention technique that works for a specific category may not perfectly work for another category. This has made the quest to eradicate SQL injection vulnerabilities an open field of research.

2 Related works

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 [9] employed SHA-1 hashing algorithm to prevent batch query SQL injections. It works by extracting query attribute values from the stored inputs; these were hashed using SHA-1 hashing algorithm. After that, any other input will also be hashed and compared with the initially hashed stored input before further execution. If the compared hashed input is the same, then the SQL query will be executed otherwise rejected. With this technique, erroneous inputs cannot be processed directly by the SQL query, and an attempt to fetch stored inputs values will return hashed values that are already encrypted. Similarly, Boyer-Moore string matching algorithm for SQL injection attacks detection and prevention was introduced by authors in [10]. Input values are scanned

for possible attributes of SQL injection attacks. A hybrid approach which leverages on the strengths of static and dynamic approach to detecting and preventing SQL injection was proposed by Ghafarian. Static approach attempts to find fault in the written SQL query (database layer) while dynamic approach finds vulnerabilities that could be present at runtime (common gateway interface layer). Firstly, an algorithm was written to match strings from input SQL query to stored SQL query. The result obtained was matched with the expected valid query. Any observed vulnerability will cause the query to be discarded. For the dynamic approach, an algorithm was proposed to examine incoming queries dynamically. Once a vulnerability is detected, the query will be discarded.

Authors in [11] proposed a technique for preventing SQL injection and XSS attack. Web applications were categorized into two: those whose query does not change regardless of the time, and parameter were classified as static while those whose query change due to time or data passed into it were called dynamic web applications. Static and dynamic mapping model were used to detect and prevent vulnerabilities from both categories. Similar technique using Aho–Corasick pattern matching technique was proposed by Prabakar et al. Authors in [12] employed instruction set randomization to prevent second-order SQL injection. The technique dynamically builds SQL instruction sets from trusted SQL keywords. Input SQL query was saved into a proxy server and not directly into the database. Therefore, the contents of the proxy server will be examined for any malicious stings. However, the technique can only prevent Boolean-based SQL injection attack. Similar technique using instruction set randomization was also reported in [13–15]

Similarly, a technique to prevent SQL injection using ASCII codes was proposed by Srivastava. User's input was first converted to ASCII values before saving them in the database. Subsequent input will be converted to ASCII values and matched with stored ASCII values. Should there be any difference, the input SQL query will be discarded. Similarly, parse tree validation technique and code conversion were proposed for detecting SQL injection and XSS attacks in [16]. Parse tree was employed to check if user's input is vulnerable; if not vulnerable, the input will be converted to ASCII code before being stored in the database. However, ASCII code conversion consumes space, and the technique cannot handle encoded SQL injection. Ramesh [17] employed syntactic analysis for SQL injection detection and prevention. Every input SQL query will be parsed through a grammar specifically written to detect piggy backed SQL, tautology queries, and union queries. However, the proposed technique was not designed to handle other forms of SQL injections and XSS attacks. SHA-1 Hashing technique was also proposed in [18]. to prevent SQL injection and session hijacking. A unique hash value was calculated for user's input supplied during registration which is then compared with subsequent login details provided. Additionally, session hijacking was prevented by generating hash values for legitimate system and browser parameters such as browser name, host IP browser platform, and version. This hash value will be compared with subsequent values supplied. The technique was reported to be able to prevent UNION, error-based, piggy backing, and tautology attacks. Signature-based approach that conducts deep packet inspection of HTTP packet payloads was proposed in [19]. Packets sent between clients and server are screened for possible malicious keywords. Filtering technique that uses certain PHP functions was employed for SQL injection and XSS attack prevention by Voitovych et al [20]. All user's

Table 1 Special characters used to compose SQL-injection code

S/N	Character	Description
1	'	Character string indicator
2	-- or #	Single line comment
3	/*…*/	Multiple line comment
4	%	Wildcard attribute indicator
5	;	Query terminator
6	+ or ||	String concatenate
7	=	Assignment operator
8	>, <, <=, >=, ==, <> or ! =	Comparison operators

Table 2 Keywords used to compose SQL-injection code

S/N	Keyword	Description
1	*OR*	Used in Boolean-based injection attack
2	*UNION*	Used in union-based injection attack
3	*DROP*	Used to destroy the entire database table
4	*DELETE*	Used to delete rows in a database table
5	*TRUNCATE*	Used to empty a particular table in a database
6	*SELECT*	Used to retrieve record from a database table
7	*UPDATE*	Used to modify record in a database table
8	*INSERT*	Used to add record to a table in a database
9	*LIKE*	Used with the wildcard (%) to select a record that contains a particular string pattern.
10	*CONVERT( )*	Used in error-based SQL injection to causes the database server to displays some error messages.

Table 3 Different forms of injection code with their common patterns

<table>
<tr><th>S/N</th><th>Injection type</th><th>Common pattern</th><th>Example</th></tr>
<tr><td>1</td><td>Boolean-based</td><td>' OR '…' = | > | > =| < | < =|<>|! = '…';#</td><td>' OR '1' = '1';#
123' OR 'a' <> 'b' ;#
' OR '2 + 3' < = '10' ;#</td></tr>
<tr><td>2</td><td>Union-based</td><td>' union select … from …;#</td><td>' union select * from users; #
' union select name from a;#</td></tr>
<tr><td>3</td><td>Error-Based</td><td>'…convert (|avg(| round(...</td><td>111' convert(int, 'abcd')
A' avg('&%$#@*')</td></tr>
<tr><td>4</td><td>Batch query</td><td>'; drop | delete | insert | truncate | update | select…;#</td><td>aaa' ; delete * from users; #
' ; drop table users; #</td></tr>
<tr><td>5</td><td>Like-based</td><td>'OR … LIKE '…%';#</td><td>' OR username LIKE 'S%'#</td></tr>
<tr><td>6</td><td>XSS</td><td><script> …'…;</script></td><td><script>alert('Xss');</script></td></tr>
</table>

input will be filtered for any illegal characters before further processing.

3 The proposed detection and prevention technique

With a view to come up with a technique that could detect and prevent the various forms of SQL injection 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 five phases: formation of SQL injection string pattern, designing parse tree for the various forms of attacks, detecting SQL-injection and XSS attacks, preventing SQL-injection and XSS attacks using KMP algorithm, and formulating the filter functions.

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 are retrieved and documented as made available in Tables 1 and 2.

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 Designing parse tree for the various forms of attacks

Parse tree was used to represent the syntactic pattern of the various forms of SQL-Injection and Cross Site Scripting attacks. The parse trees are as follows:

(i). Boolean-based SQL injection attacks

```
start:
    input: array of characters S (the text to be searched) and array of characters W (the word sought)
    output: array of integers P (positions in S at which W is found); an integer nP (number of positions)
    define variables:
    an integer j ← 0 (the position of the current character in S)
    an integer k ← 0 (the position of the current character in W)
    an array of integers T (the table computed elsewhere)
    let nP ← 0
    while j < length(s) do
        if W[k]= S[j] then let j ← j + 1, k ← k + 1
    if k = length(W) then let P[nP] ← j −k, nP ←Np+1
    (occurrence found if only first occurrence is needed, may be returned here)
    Let K ←T[k] if k < 0 then let J ←j+1, let k ←k+1
    Let K ←T[k] (T[length (W)] can't be -1)
    Else
    Let K ←T[k] if K < 0 let j ← j + 1, k ← k + 1
Stop
```

3.3 Detecting SQL injection and XSS attacks

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

(i). Boolean-based SQL-injection attacks: As presented in Table 3, 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' ;# (Fig. 1).

(ii). Union-based SQL injection attacks: 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; # (Fig. 2).

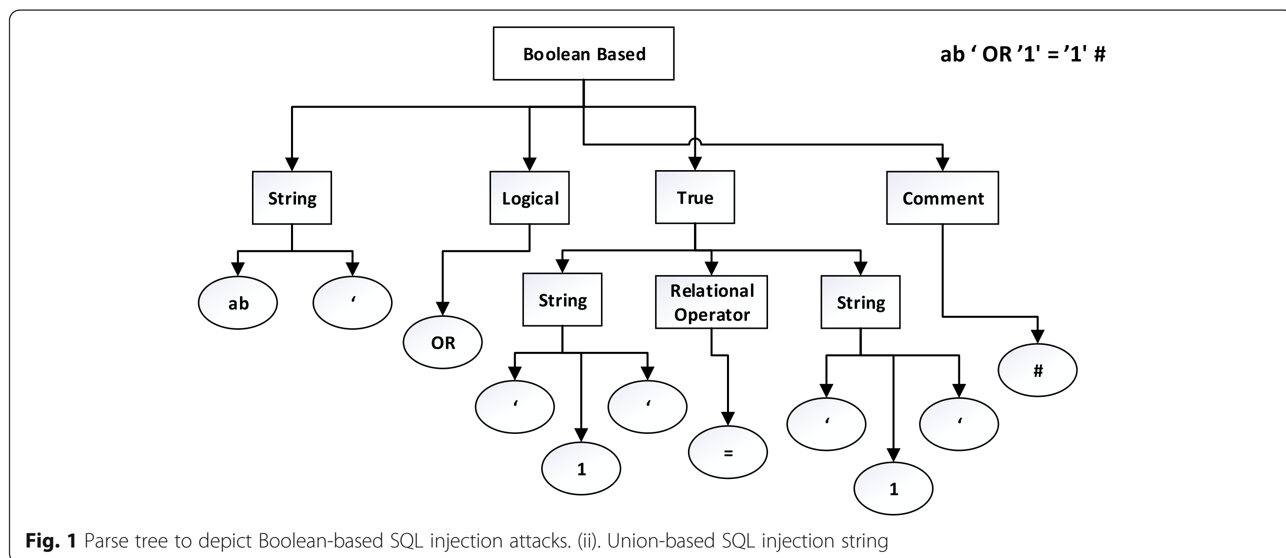


Fig. 1 Parse tree to depict Boolean-based SQL injection attacks. (ii). Union-based SQL injection string

(iii).Error-based SQL injection attacks: The presence of a single quote (‘) from the user's input, followed by zero or more SQL functions, indicates the presence of error-based SQL injection attacks. Example includes 111’ convert (int, ‘abcd’); A’ avg(‘&%$#@*’), and ‘ round (‘abc’, 2) (Fig. 3).

(iv).Batch query SQL injection attacks: Input strings with a single quote (‘) followed by a SQL keyword “DROP”, “DELETE”, “INSERT” etc. then one or more identifiers, followed by semicolon (;) with a hash (#). Examples include aaa’; delete * from users; # or ‘; drop table users; # (Fig. 4).

(v). Like-based SQL injection attack: from Table 3, category (e) 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%’;# (Fig. 5).

(vi).XSS attack: this can be detected when a JavaScript open tag “<script>” is encountered from the input

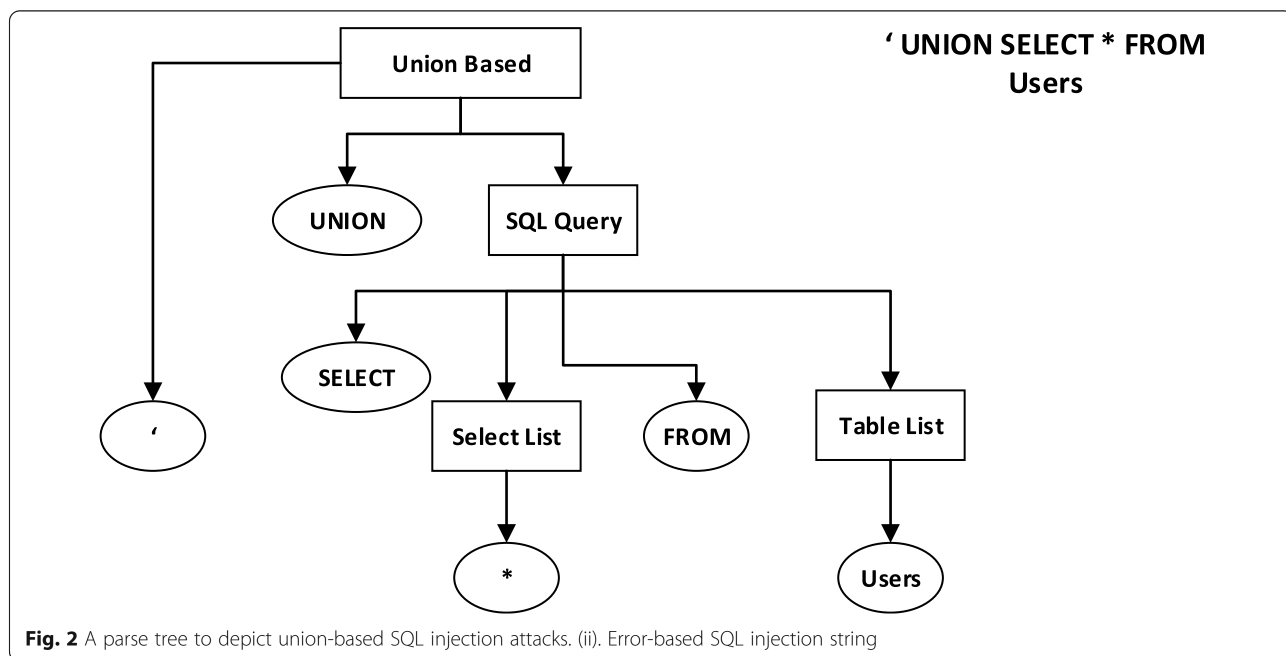


Fig. 2 A parse tree to depict union-based SQL injection attacks. (ii). Error-based SQL injection string

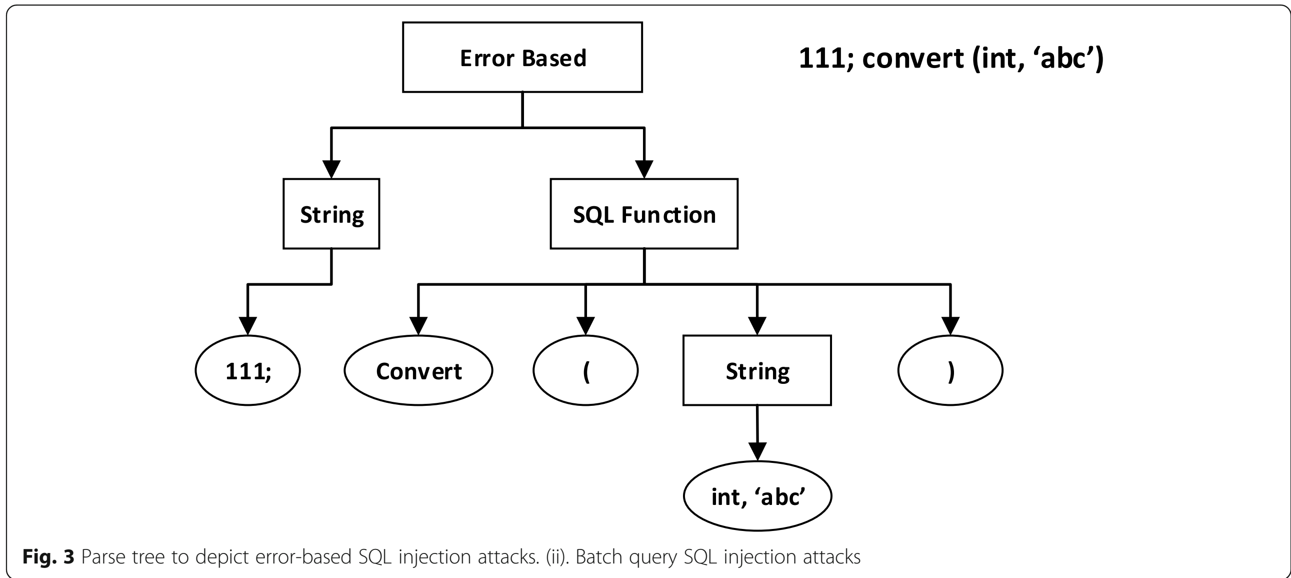


Fig. 3 Parse tree to depict error-based SQL injection attacks. (ii). Batch query SQL injection attacks

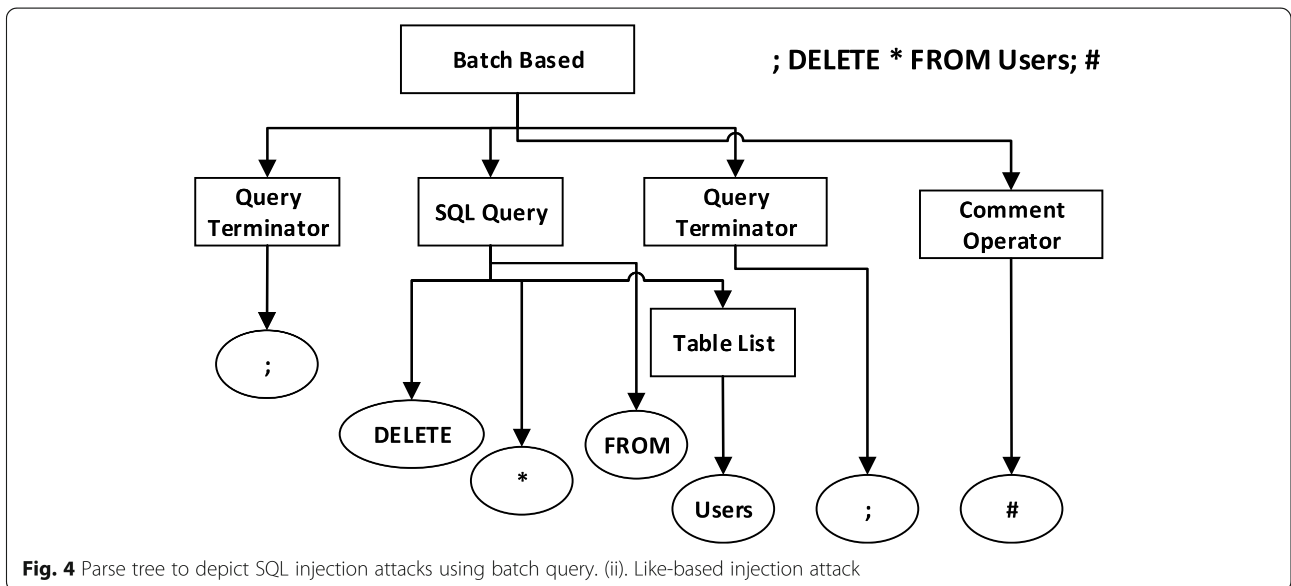


Fig. 4 Parse tree to depict SQL injection attacks using batch query. (ii). Like-based injection attack

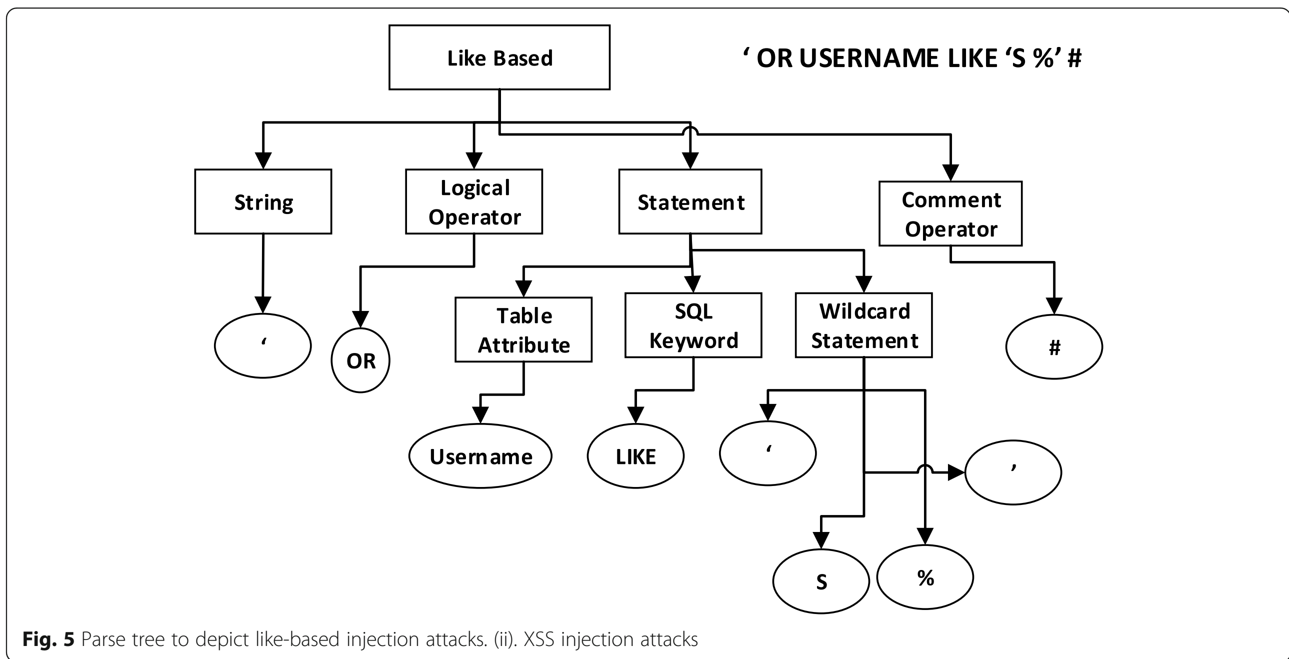


Fig. 5 Parse tree to depict like-based injection attacks. (ii). XSS injection attacks

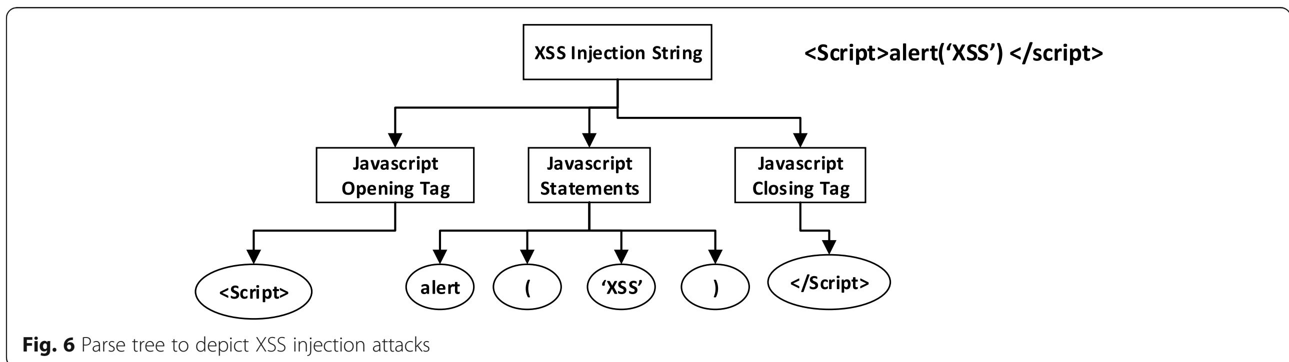


Fig. 6 Parse tree to depict XSS injection attacks

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(" XSS "); </script> (Fig. 6).

3.4 Preventing SQL-injection and XSS attacks using KMP algorithm

KMP 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:

$$I = \sum_{i=0}^{n} f_i \quad Where\ f\ is\ the\ user's\ input\ from\ each\ form\ text\ field$$

```
filter(I) { data = convertASCIItoString(I);
    if(data <> ""){
        a = checkBooleanBasedSqli(data)
        b = checkUnionBasedSqli(data);
        c = checkErrorBasedSqli(data);
        d = checkBatchQuerySqli(data);
        e = checkLikeBasedSqli(data);
        f = checkXss(data);
            if(true (a||b||c||d||e||f)){
                blockUser();
                resetHTTP();
                warningMessage();
            }
                else {grantAccess(); }
    }
```

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, reset the HTTP request, and display a corresponding warning message. The first statement in the algorithm below represents user's input which is collected from the web form using POST Method, and it is donated by *I*. The filter() then collects the user's input and firstly converts any ASCII String found in order to prevent encoded injection attack. If there is no any ASCII String and it is not empty, then, the user's input will be parsed to other functions in order to check whether it contains some injection code of Boolean-based SQLI, Union-based SQLI, Error-based SQLI, Batch query SQLI, Like-based SQLI, and XSS, and the outcomes of the functions are represented as a, b, c, d, e, and f respectively. If one of the result returns true, then, an injection string is found in the user's input, and it then triggers some functions: *blockUser()*, *resetHTTP()*, and *warningMessage()* so that to block the user, reset the HTTP request and issue a warning message. Otherwise, access is granted. The pseudo code illustrating this process goes thus:

```
checkBooleanBasedSqli(input){
injPattern[] = {"'","'","'","=""'","'","#"};
lOprt[] = {"or","||"};
rOprt[] = {'=','>','>=','<','<=','<>','!='};
for(i = 0; i < injPattern.length; i++)
{
        if(KMP_Search(input,injPattern[i]) > 0)
                {
if(i == 0){counter = 0;
        for(j = 0; j < lOprt.length; j++){
if(KMP_Search(input, lOprt[j]) > 0){counter++;}}
    if(counter == 0){result = false; break;}
            if(i == 2)
    {counter = 0;
        for(k = 0; k < rOprt.length; k++){
        if(KMP_Search(input,rOprt[k]) > 0){counter++;}
    }
    if(counter == 0){result = false; break;}
        if((i + 1) == injPattern.length){result = true;}
        input = end(slice(injPattern[i], input, 2));
  }else{ result = false; break;}
            return result;
}
```

(i). Formulating the *checkBooleanBasedSqli*( ) function: this was used to prevent Boolean-based SQL injection attack:

```
CheckUnionBasedSqli(input){
injPattern[] = {"'","union "," select ","from" ,"#"};
for(i = 0; i < injPattern.length; i++)
{
    if(KMP_Search(input,injPattern[i]) > 0)
    {
        if((i + 1) == injPattern.length){result = true;}
        input = end(slice(injPattern[i], input, 2));
    }
    else{ result = false; break;
}return result;
}
```

(ii). Formulating the *checkUnionBasedSqli*() *f*unction: this was used to prevent union-based SQL injection attack:

```
checkErrorBasedSqli(input){
injPattern[] = {"'",")"};
sqlFn[] = {"convert(","avg(","round(","sum(","max(","min("};
for(i = 0; i < injPattern.length; i++){
    if(KMP_Search(input,injPattern[i]) > 0)
    {
if(i == 0){counter = 0;
    for(j = 0; j < sqlFn.length; j++){
if(KMP_Search(input, sqlFn[j]) > 0){counter++;}}
    if(counter == 0){result = false; break;}
    if((i + 1) == injPattern.length){result = true;}
    input = end(slice(injPattern[i], input, 2));
} else{ result = false; break;}
    return result;
}
```

(iii)Formulating the *checkBatchQuerySqli*() *f*unction: this was used to prevent batch query SQL injection attack:

```
checkBatchQuerySqli(input){
injPattern[] = {"'",";",";","#"};
sqlk[] = {"delete","drop","insert","truncate","update","select","alter"};
for(i = 0; i < injPattern.length; i++)
{
    if(KMP_Search(input,injPattern[i]) > 0)
    {
if(i == 0){counter = 0;
    for(j = 0; j < sqlk.length; j++){
if(KMP_Search(input, sqlk[j]) > 0){counter++;}}
    if(counter == 0){result = false; break;}
    if((i + 1) == injPattern.length){result = true;}
    input = end(slice(injPattern[i], input, 2));
}
    else { result = false; break;}
    return result;
}
```

(iv).Formulating the *checkLikeBasedSqlis*() *f*unction: this was used to prevent like-based SQL injection attack:

```
checkLikeBasedSqli(input){
injPattern[] = {"'","like ","'","%" "'","#"};
lOprt[] = {"or","||"};
for(i = 0; i < injPattern.length; i++)
{
    if(KMP_Search(input,injPattern[i]) > 0)
    {
if(i == 0){counter = 0;
    for(j = 0; j < lOprt.length; j++)
    {
        if(KMP_Search(input,lOprt[j]) > 0){counter++;}
    }
    if(counter == 0) {result = false; break;}
    if((i + 1) == injPattern.length){result = true;}
    input = end(slice(injPattern[i], input, 2));
} else { result = false; break;}
    return result;
}
```

Fig. 7 Developed ABC journal of education

(v). Formulating the *checkXss*( ) function: this was used to prevent XSS attacks.

```
checkXss(input){
injPattern[] = {" </script > ","' "," </script > "};
for(i = 0; i < injPattern.length; i + +)
{
        if(KMP_Search(input, injPattern[i]) > 0){
        if((i + 1) == injPattern.length){result = true; }
        input = end(slice(injPattern[i], input, 2));
}
        else { result = false; break; }
        return result;
}
```

Therefore, to detect and prevent any of the attacks, every input strings will be passed through all the functions formulated. If at least one function return True, then, the following functions will be triggered: blockUser(), resetHTTP(), and warningMessage(). These functions are used to interact with the prospective hackers.

4 Results and discussion

The proposed technique was implemented using PHP Scripting Language and Apache XAMPP Server. The PHP was selected as a scripting Language, because it is the most widely used server-side scripting language in building database-driven web-based application while the Apache XAMPP Server was chosen due to its cross-platform compatibility, it supports any operating system, and it also supports both PHP scripting language and SQL.

4.1 Test environment

Apache Web Server and Internet Information Server (ISS) were 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 66.0.3515.44), Google Chrome (version 79.0.3945.130), Mozilla Fire fox (Version 67.0.4), and Internet Explorer 11 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 both remote server and local WAMP/XAMPP server. To test the proposed technique, a vulnerable web application shown in Fig. 7 was purposely developed.

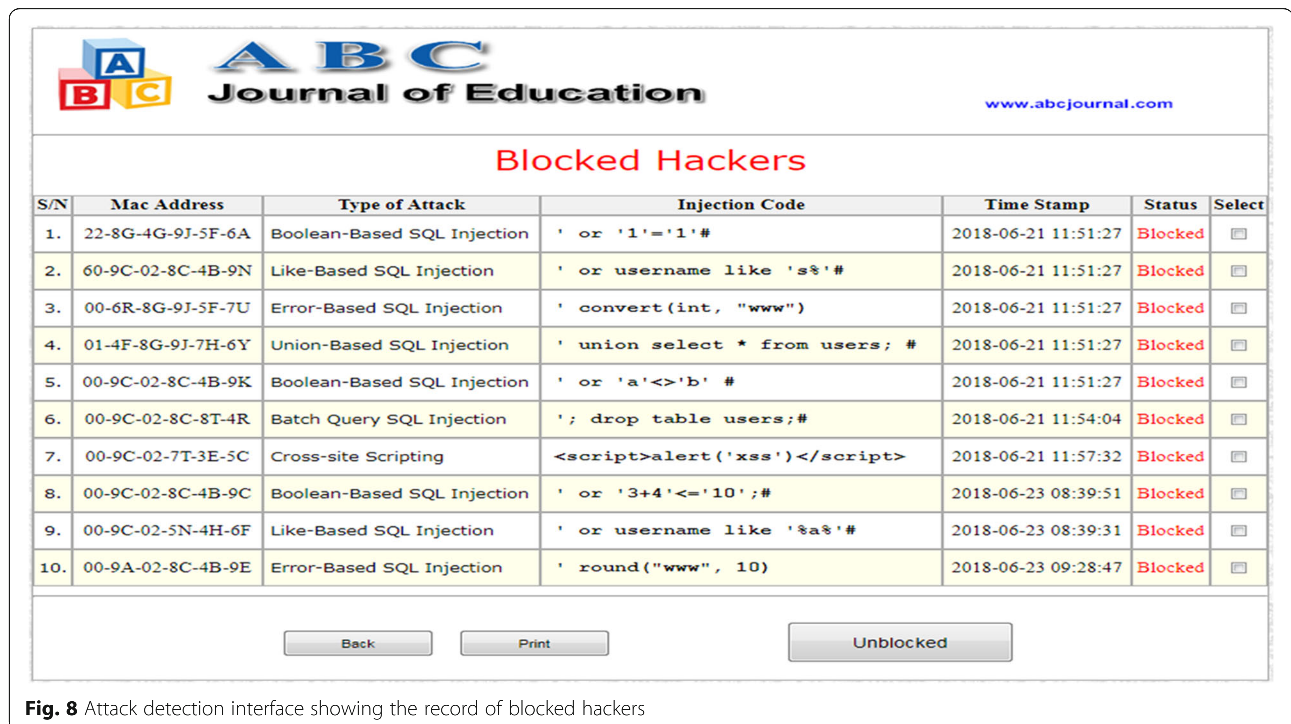
An attempt was made to submit various known SQL injection and XSS attack patterns using a "test plan" shown in Table 4. The test plan consists of test cases which contains input string for various attacks.

The input strings were supplied via the input fields of the web application. When an attack was detected, the uses 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 table shown in Fig. 8.

Based on the results obtained from the various attack attempts, the proposed technique was able to successfully detect and prevent all the attacks.

Table 4 The test plan

S/N	Attack type	Sample injection code
1	Boolean-based SQLi	' OR " = "; #
2	Boolean-based SQLi	' OR '1'='1'; #
3	Boolean-based SQLi	' OR '3'! ='8' ;#
4	Boolean-based SQLi	' OR 'a'<>'b' ;#
5	Boolean-based SQLi	aa' OR '2 + 3' < = '7' ;#
6	Like-based SQLi	a' OR username LIKE 'S%';#
7	Like-based SQLi	' OR password LIKE '%2%';#
8	Like-based SQLi	' OR username LIKE '%e';#
9	Union-based SQLi	'UNION select * from users; #
10	Union-based SQLi	'UNION select cardNo, pin from customer; #
11	Error-based SQLi	' convert(int, (select * from users LIMIT 1))
12	Error-based SQLi	' convert(int, "aaaa")
13	Error-based SQLi	' round((select username from users), 3)
14	Batch query SQLi	' ; drop table users ; #
15	Batch query SQLi	' ; delete * from customer ; #
16	Batch query SQLi	' ; insert into users values ('Bala', '1234') ; #
17	Batch query SQL injection	' ; update table users set username = 'Bala', password ='123' ; #
18	Encoded cross-site scripting	<script> alert(" XSS ") </script>
19	Encoded SQL injection	& # x39 & # x85 & # x78 & # x73 & # x79 & # x78 & # x32 & # x83 & # x69 & # x76 & # x69 & # x67 & # x84 & # x32 & # x 42 & # x32 & # x70 & # x82 & # x79 & # x77 & # x32 & # x117 & # x115 & # x101 & # x114 & # x115 & # x45 & # x45
20	Cross-site scripting	<script> alert('XSS') </script>
21	Cross-site scripting	<script>myFunction( );</script>



S/N	Mac Address	Type of Attack	Injection Code	Time Stamp	Status	Select
1.	22-8G-4G-9J-5F-6A	Boolean-Based SQL Injection	' or '1'='1'#	2018-06-21 11:51:27	Blocked	
2.	60-9C-02-8C-4B-9N	Like-Based SQL Injection	' or username like 's%'#	2018-06-21 11:51:27	Blocked	
3.	00-6R-8G-9J-5F-7U	Error-Based SQL Injection	' convert(int, "www")	2018-06-21 11:51:27	Blocked	
4.	01-4F-8G-9J-7H-6Y	Union-Based SQL Injection	' union select * from users; #	2018-06-21 11:51:27	Blocked	
5.	00-9C-02-8C-4B-9K	Boolean-Based SQL Injection	' or 'a'<>'b' #	2018-06-21 11:51:27	Blocked	
6.	00-9C-02-8C-8T-4R	Batch Query SQL Injection	'; drop table users;#	2018-06-21 11:54:04	Blocked	
7.	00-9C-02-7T-3E-5C	Cross-site Scripting	<script>alert('xss')</script>	2018-06-21 11:57:32	Blocked	
8.	00-9C-02-8C-4B-9C	Boolean-Based SQL Injection	' or '3+4'<='10';#	2018-06-23 08:39:51	Blocked	
9.	00-9C-02-5N-4H-6F	Like-Based SQL Injection	' or username like '%a%'#	2018-06-23 08:39:31	Blocked	
10.	00-9A-02-8C-4B-9E	Error-Based SQL Injection	' round("www", 10)	2018-06-23 09:28:47	Blocked	

Fig. 8 Attack detection interface showing the record of blocked hackers

Table 5 Results of existing works vs proposed technique

			Attack type						
		Ref.	Boolean-based SQLI	Union-based SQLI	Error-based SQLI	Batch query SQLI	Like-based SQLI	XSS	Encoded injection
Methodology	Using pattern matching algorithm	[21]	✓	✓	✓	✓	✓	✓	X
		[22]	✓	✓	✓	✓	✓	✓	X
		[23]	✓	✓	✓	✓	✓	✓	X
		[24]	✓	✓	✓	✓	✓	✓	X
		[25]	✓	✓	✓	✓	✓	✓	X
		[26]	✓	✓	✓	✓	✓	✓	X
		[27]	✓	✓	✓	✓	✓	✓	X
	Using data encryption algorithm	[28]	✓	✓	✓	✓	✓	X	✓
		[18]	✓	✓	✓	✓	✓	X	✓
		[29]	✓	✓	✓	✓	✓	X	✓
		[30]	✓	✓	✓	✓	✓	X	✓
		[31]	✓	✓	✓	✓	✓	X	✓
		[32]	✓	✓	✓	✓	✓	X	✓
	ISR	[12]	✓	✓	✓	✓	✓	X	X
		[33]	✓	✓	✓	✓	✓	X	X
	Proposed algorithm		✓	✓	✓	✓	✓	✓	✓

4.2 Results of existing works and the proposed technique

Results obtained from the proposed technique were compared with those available in existing literature. Seven (7) existing literature which used pattern matching algorithms, six (6) literature which used data encryption algorithms, two (2) literature which used instruction set randomization, and one (1) literature which used PHP escaping functions were used for the comparison. As documented in Table 5, existing techniques which used data encryption algorithms were able to prevent all the five forms of SQL injection attacks but failed to prevent XSS attacks. Existing techniques which used PHP escaping functions and pattern matching algorithms were able to prevent all the five forms of SQL Injection attacks including the XSS attacks but failed to prevent encoded injection attacks. Existing techniques that used instruction set randomization were able to prevent all the five forms of SQL injection but failed to prevent XSS and encoded injection attacks. In a nut shell, any of the existing methods has its own drawback while the proposed algorithm prevents all the five forms of SQL injection attacks including XSS and encoded injection 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, 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 technique was tested using a test plan that consist of different forms of Boolean-based, union-based, error-based, batch query, like-based, encoded SQL injections 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. 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.

Acknowledgements

The authors appreciate Landmark University Centre for Research and Development, Landmark University, Omu-Aran, Nigeria for fully sponsoring the publication of this article.

Authors' contributions

Authors AHD, AA, and AOC formulated and implemented the methodology and drafted the manuscript. AOC adjusted the initial methodology and

supervised the work. ANO and KAA carried out the survey of related work and reviewed the manuscript. All authors read and approved the final manuscript.

Funding
Publication of this research article was funded by Landmark University Centre for Research and Development, Landmark University, Omu-Aran, Nigeria.

Availability of data and materials
Not applicable

Competing interests
The authors declare that there are no competing interests.

Author details
[1]Department of Computer Science, University of Ilorin, Ilorin, Nigeria. [2]Computer Science Department, Gombe State Polytechnic, Gombe, Nigeria. [3]Computer Science Department, Landmark University, Kwara State, Omu-Aran, Nigeria.

Received: 7 August 2019 Accepted: 24 June 2020
Published online: 18 August 2020

References

1. Acunetix_web_application_vulnerability_report_2019
2. B. Soewito, F.E. Gunawan, Prevention structured query language injection using regular regular expression and escape string. Procedia Comput. Sci. **135**, 678–687 (2018) https://doi.org/10.1016/j.procs.2018.08.218
3. M.A. Ahmed, F. Ali, Multiple-path testing for cross site scripting using genetic algorithms. J. Syst. Archit. **000**, 1–13 (2015) https://doi.org/10.1016/j.sysarc.2015.11.001
4. Y. Jang, J. Choi, Detecting SQL injection attacks using query result size. Comput Security, 1–15 (2014) https://doi.org/10.1016/j.cose.2014.04.007
5. P.R. Mcwhirter, K. Kifayat, Q. Shi, B. Askwith, SQL injection attack classification through the feature extraction of SQL query strings using a gap-weighted string subsequence kernel. J. Inform. Sec. Appl. **40**, 199–216 (2018) https://doi.org/10.1016/j.jisa.2018.04.001
6. O.C. Abikoye, A.D. Haruna, A. Abubakar, N.O. Akande, E.O. Asani, Modified advanced encryption standard algorithm for information security. Symmetry **11**, 1–17 (2019) https://doi.org/10.3390/sym11121484
7. N.O. Akande, C.O. Abikoye, M.O. Adebiyi, A.A. Kayode, A.A. Adegun, R.O. Ogundokun, in *International Conference on Computational Science and Its Applications*. Electronic medical information encryption using modified blowfish algorithm (Springer, Cham, 2019), pp. 166–179 https://doi.org/10.1007/978-3-030-24308-1_14
8. A.O. Christiana, A.N. Oluwatobi, G.A. Victory, O.R. Oluwaseun, A Secured One Time Password Authentication Technique using (3, 3) Visual Cryptography Scheme. IOP Conf. Series: Journal of Physics: Conf. Series **1299**, 1–10 (2019 https://doi.org/10.1088/1742-6596/1299/1/012059)
9. Q. Temeiza, M. Temeiza, J. Itmazi, A novel method for preventing SQL injection using SHA-1 algorithm and syntax-awareness. Sudanese J. Comput. Geoinform. **1**(1), 16–26 (2017)
10. G. Buja, T.F. Abdul, B.A.J. Kamarularifin, M.A. Fakariah, T.F. Abdul-Rahman, *Detection model for SQL injection attack : an approach for preventing a web application from the SQL injection attack*, Symposium on Computer Applications and Industrial Electronics (2014), pp. 60–64
11. A.S. Piyush, A.N. Mhetre, *International Conference on Pervasive Computing (ICPC). A novel approach for detection of SQL injection and cross site scripting attacks* (2015), pp. 1–4
12. C. Ping, W. Jinshuang, P. Lin, Y. Han, *Research and implementation of SQL injection prevention method based on ISR*, IEEE International Conference on Computer and Communications (2016), pp. 1153–1156
13. U. Upadhyay, K. Girish, *SQL injection avoidance for protected database with ASCII using SNORT and honeypot*, International Conference on Advanced Communication Control and Computing Technologies (ICACCCT), (978) (2016), pp. 596–599
14. B. Appiah, E. Opoku-mensah, *SQL injection attack detection using fingerprints and pattern matching technique*, IEEE International Conference on Software Engineering and Service Science (ICSESS) (2017), pp. 583–587
15. C. Ping, *A second-order SQL injection detection method*, 2017 IEEE 2nd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC) (2017), pp. 1792–1796
16. A. John, A. Agarwal, M. Bhardwaj, *An adaptive algorithm to prevent SQL injection. 4* (2015), pp. 12–15 https://doi.org/10.11648/j.ajnc.s.2015040301.13
17. A. Ramesh, *An Authentication Mechanism to Prevent SQL Injection by Syntactic Analysis* (2015)
18. D. Karis, J. Vanajakshi, K.N. Manjunath, P. Srikanth, *An effective method for preventing SQL injection attack and session hijacking*, IEEE International Conference on Recent Trends in Electronics Information & Communication Technology (RTEICT) (2017), pp. 697–701
19. A. Pramod, A. Ghosh, A. Mohan, M. Shrivastava, R. Shettar, *SQLI detection system for a safer web application*, International Advance Computing Conference (IACC) (2015), pp. 237–240
20. O.P. Voitovych, O.S. Yuvkovetskyi, L.M. Kupershtein, *SQL injection prevention system*, International Conference "Radio Electronics & InfoCommunications" (UkrMiCo) (2016), pp. 2–5
21. P. Chen, J. Wang, L. Pan, H. Yu, *Research and implementation of SQL injection prevention method based on ISR*, IEEE International Conference on Computer and Communications (IEEE, Chengdu, 2016), pp. 1153–1156
22. G. Ahmad, *A hybrid method for detection and prevention of SQL injection attacks*, Computing Conference (London, 2017), pp. 833–838
23. P. Amith, G. Agneev, M. Amal, S. Mohit, S. Rajashree, *SQLI detection system for a safer web application*, IEEE International Advance Computing Conference (IACC) (IEEE, Banglore, 2015), pp. 237–240
24. R. Ashwin, B. Anirban, V.L. Anand, *An authentication mechanism to prevent SQL injection by syntactic analysis*, International conference on trends in automation, communications and Computing technology (I-TACT-15) (IEEE, Bangalore, 2015), pp. 1–6
25. A. Prabakar, M. KarthiKeyan, K. Marimuthu, *An efficient technique for preventing SQL injection attack using pattern*, International Conference on Emerging Trends in Computing, Communication and Nanotechnology (ICECCN) (2013), pp. 503–506
26. A. Ghafarian, *A hybrid method for detection and prevention of SQL injection attacks*, IEEE Comput Conference (2017), pp. 833–838
27. P. Amutha, M. KarthiKeyan, K. Marimuthu, *An efficient technique for preventing SQL injection attack using pattern matching algorithm*, IEEE international conference on emerging trends in Computing, communication and nanotechnology (ICECCN) (2013), pp. 503–506
28. T. Qais, T. Mohammad, I. Jamil, *A novel method for preventing SQL injection using SHA-1 algorithm and syntax-awareness*, International conference on information and communication Technologies for Education and Training and international conference on Computing in Arabic (ICCA-TICET) (IEEE, Khartoum, 2017), pp. 1–4
29. U. Utpal, K. Girish, *SQL injection avoidance for protected database with ASCII using SNORT and honeypot. International conference on advanced communication control and Computing technologies (ICACCCT)* (IEEE, Ramanathapuram, 2016), pp. 596–599
30. J. Ashish, A. Ajay, B. Manish, An adaptive algorithm to prevent SQL injection. Am. J. Networks Commun., 12–15 (2015)
31. M. Srivastava, *Algorithm to Prevent Back End Database against SQL Injection Attacks International Conference on Computing for Sustainable Global Development (INDIACom)* (2014), pp. 755–757
32. T. Pravallica, S. Betam, *An application to prevent SQL injection attacks using randomized encription algorithm. International journal of computer trends and technology (IJCTT)* (2013), pp. 2782–2786
33. B. Geogiana, B.A. Kamarularifin, B.H. Fakariah, F.A. Teh, *Detection model for SQL injection attack: an approach for preventing a web application from the SQL injection attack*, Symposium on Computer Applications and Industrial Electronics (IEEE, Penang, 2014), pp. 60–64

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.