International Journal of Computing and Corporate Research
ISSN (Online) : 2249-054X
Volume 5 Issue 3 May 2015
International Manuscript ID : 2249054XV5I3052015-02

ANALYSIS AND IMPLEMENTATION OF MULTILAYERED SECURED ALGORITHM FOR ASSORTED APPLICATIONS

Ankur Tiwari
Research Scholar
Computer Science and Engineering
Chandigarh University, Gharuan
Punjab, India

Isha Sharma
Assistant Professor
Computer Science and Engineering
Chandigarh University, Gharuan
Punjab, India

ABSTRACT

A SQL injection attack consists of insertion or "injection" of a SQL query via the input data from the client to the application. A successful SQL injection exploit can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file present on the DBMS file system and in some cases issue commands to the operating system. SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input in order to effect the execution of predefined SQL commands. One traditional approach to preventing SQL injection attacks is to handle them as an input validation problem and either accept only characters from a whitelist of safe values or identify and escape a blacklist of potentially malicious values. Whitelisting can be a very effective means of enforcing strict input validation rules, but parameterized SQL statements require less maintenance and can offer more guarantees with respect to security. As is almost always the case, blacklisting is riddled with loopholes that make it ineffective at preventing SQL injection attacks. There are other lots of other methods to avoid and push back such attacks. In this research work, we have proposed and implemented the fuzzy random attempt and user behavior analysis based algorithm for avoidance of malicious access of the devices connected with Internet of Things (IoT).

Keywords - Software Testing, IoT Security, Security against SQL Injections

INTRODUCTION

SQL infusion is a code infusion system, used to strike information driven requisitions, in which noxious SQL explanations are embedded into a passage field for execution (e.g. to dump the database substance to the agressor). SQL infusion must abuse a security helplessness in a provision's product, for instance, when client data is either mistakenly sifted for string exacting break characters installed in SQL articulations or client information is not determinedly written and startlingly executed. SQL infusion is generally known as an ambush vector for sites however might be utilized to assault any kind of SQL database.

A standard SQL query is composed of one or more SQL commands, such as SELECT, UPDATE, or INSERT. SQL injection is one of the most common application layer attacks. According to NIST SQL injection amounted to 14% of the total web application vulnerabilities in 2006 . SQL injection is the act of passing a SQL query or command as input into a web application. It exploits web applications that use client-side data in a SQL query without proper input validation. SQL injection attacks usually target data residing in a database.

A SQL injection attack occurs on database-driven websites when unauthorized SQL queries are executed on vulnerable sites. This attack can bypass a firewall and can affect a fully patched system. For this to happen port 80, the default web port, is the only thing required. SQL injection attacks target a specific web application where the vulnerability of the relational database is either know or discovered by the attacker.

In this example students will be separated into individual groups and they will proceed to assigning roles. Some students will portray the role as the web application developer and others will portray the role as the attacker. The developers will create an application that includes a relational database. The attackers will try to hack the application. This case study uses the examples developed by Mitchell Horper to let students get hands-on experience .

In general Web applications use data read from a client to construct SQL queries. This can lead to vulnerability where an attacker can execute SQL queries to cause SQL injection attacks. Several SQL injection attacks such as manipulating the contents of a query command, forcing login and modify information in a database will be discussed in this section.

A hacker can manipulate the contents of a query to create a SQL injection attack. For example

Change the userPass into ' ' or 1=1 --' to create a select command like this:

select count(*) from users where userName='john' and userPass=' '
or 1=1 --'

Therefore the query only checks for the username of john. Instead of checking for a matching password, it checks for an empty password, or the conditional equation of 1=1. In this case if the password field is empty or 1 equals 1(which is always true), a valid row will be found in the users table with username john. The single line delimeter (--) that comments out the last quote stops ASP returning an error about any unclosed quotations. As the result one row will be returned and the message “Logged In” will be displayed.

This method can be used for the username field. If changing the username is ' or 1=1 --- and password is empty such as:

Username: ' or 1=1 ---
Password: [Empty]

And execute a select query:
select count(*) from users where userName=' ' or 1=1 --' and userPass=' '

A count of all rows in the users table will be return. This is an example of SQL injection attack that is implemented by adding code that manipulates the contents of a query to get an undesired result.
The following example demonstrates how force login SQL injection works. Consider the following query that is based on the users table.
select userName from users where userName=' ' having 1=1

A page call login.asp can easily be developed to query the database by using these login credentials:
Username: ' having 1=1 ---
Password: [Anything]

When a user clicks on the submit button to start the login process, the SQL query causes ASP to send the following error message to the browser:

Microsoft OLE DB Provider for SQL Server (0x80040E14)
Column 'users.userName' is invalid in the select list because it is not contained in an aggregate function and there is no GROUP BY clause.
/login.asp, line 16

This error message tells the unauthorized user the name of one field from the database: users.userName. Using the name of this field, a user can use SQL Server's LIKE keyword to login with the following credentials:

Username: ' or users.userName like 'a%' ---
Password: [Anything]

Once again, this performs an injected SQL query against the users table:
select userName from users where userName=' ' or
users.userName like 'a%' --' and userPass=' '

When the users table was created, a user whose userName field was admin and userPass field was wwz04ff was also created. Logging in with the username and password shown above uses SQL's like keyword to get the username. The query grabs the userName field of the first row whose userName field starts with a, which in this case is admin:

SQL Injection: Modify the Content of a Database

Let us create a products table and rows on the SQL server as following:

```
create table products
(
id int identity(1,1) not null,
prodName varchar(50) not null,
)

insert into products(prodName) values('Pink Hoola Hoop')
insert into products(prodName) values('Green Soccer Ball')
insert into products(prodName) values('Orange Rocking Chair')
```

Let us create a products.asp ASP script as follows:

```
<%
dim prodId
prodId = Request.QueryString("productId")

set conn = server.createObject("ADODB.Connection")  /* connect to database
set rs = server.createObject("ADODB.Recordset")

query = "select prodName from products where id = " & prodId  /* select a product

conn.Open "Provider=SQLOLEDB; Data Source=(local);
Initial Catalog=myDB; User Id=sa; Password="
rs.activeConnection = conn
rs.open query

if not rs.eof then
response.write "Got product " & rs.fields("prodName").value
else response.write "No product found"
```

```
end if
%>
```

Visit products.asp in the browser with the following URL:
http://localhost/products.asp?productId=1

The following line of text in the browser is displayed:
Got product Pink Hoola Hoop

Notice product.asp returns a field from the recordset based on the field's name:
response.write "Got product" & rs.fields("prodName").value

Although this may seem more secure it is not. By manipulating the database a SQL injection can occur because the WHERE clause of the query is based on a numerical value:
query = "select prodName from products where id = " & prodId

The products.asp page requires a numerical product Id passed as the productId querystring variable.

Consider the following URL to products.asp:
http://localhost/products.asp?productId=0%20or%201=1

Each %20 in the URL represents a URL-encoded space character, so the URL looks like:
http://localhost/products.asp?productId=0 or 1=1

When used in conjunction with products.asp, the query looks like:
select prodName from products where id = 0 or 1=1

International Journal of Computing and Corporate Research
ISSN (Online) : 2249-054X
Volume 5 Issue 3 May 2015
International Manuscript ID : 2249054XV5I3052015-02

From the above select command we know how to use some URL-encoding, the names of the products can be pulled from the product table with the following url:
http://localhost/products.asp?productId=0%20having%201=1

This would generate the following error in the browser:
Microsoft OLE DB Provider for SQL Server (0x80040E14)
Column 'products.prodName' is invalid in the select list because it is not contained in an aggregate function and there is no GROUP BY clause.
/products.asp, line 13

Take the name of the products field (products.prodName) and call up the following URL in the browser:
http://localhost/products.asp?productId=0;insert%20into%20products
(prodName)%20values(left(@@version,50))

Here is the query without the URL-encoded spaces:
http://localhost/products.asp?productId=0;insert into
products(prodName) values(left(@@version,50))

It returns "No product found". However it also runs an INSERT query on the products table, adding the first 50 characters of SQL server's @@version variable (which contains the details of SQL Server's version, build, etc.) as a new record in the products table.

To get to the SQL server's version, a user must call up the products.asp page with the value of the latest entry in the products table such as:
http://localhost/products.asp?productId=(select%20max(id)
%20from%20products)

This query takes the ID of the latest row added to the products table using SQL server's MAX function. The output is the new row that contains the SQL server version details:
Got product Microsoft SQL Server 2000 - 8.00.534 (Intel X86)

This method of injection can be used to perform numerous tasks.

A Real World SQL injection attack

In May 2008 China and Taiwan were hit by a large SQL injection attack that inserted malware in thousands of websites . On May 13, the attack was detected as originating from a server from in China. The attackers made no effort to hide the source IP address. Many victim websites were ruined because they sustained lots of permanent changes from the SQL injection attacks. Thousands of websites were hit and most of them were in China.

The hackers used automated queries through Google's search engine to identify vulnerable websites. The attackers used automated queries to Google Inc's search engine to identify Web sites vulnerable to the attack. The attack uses SQL injection to infect websites with malware, which exploits vulnerabilities in the browsers of those who visit the sites. The malware came from 1,000 different servers and targeted 10 vulnerabilities in Internet Explorer and related plug-ins that are popular in Asia . The Mackay Memorial Hospital had a screenshot that shows that the rendering of the site had been affected and displayed the SQL sting injected by the attack. The large companies Web sites such as SouFun.com, Mycar168.com in China have been affected.

The impact was on a large-scale. There were thousands of victim websites that had no service. Many individuals had to find other ways to do their business.

Preventing SQL Injection Attacks

If a software developer designs scripts and applications with security consideration most of SQL injection attacks can be avoided. In the following section several methods that software developers can use to reduce web applications vulnerability for SQL injection attacks will be discussed .

Method 1: Limit User Access

The default system account for SQL server 2000 should never be used because of its unrestricted nature. Setting up accounts for specific purposes is always a good idea. For example, if a database lets users view and order products, then the administrator must set up an account called **webUser_public** that has **SELECT** rights on the products table, and **INSERT** rights only on the orders table.

If a user does not make use of extended stored procedures, or has unused triggers, stored procedures, user-defined functions, etc, then remove them, or move them to an isolated server. SQL injection attacks make use of extended stored procedures such as **xp_cmdshell** and **xp_grantlogin**. Removing them can block the attack before it occurs.

Method 2: Escape Quotes

SQL injection attacks require the user of single quotes to terminate an expression. The chance of an SQL injection attack can be reduced by using a simple replace function and converting all single quotes to two single quotes. Using ASP to create a generic replace function will handle the single quotes automatically. See the following example:

```
<%
function stripQuotes(strWords)
stripQuotes = replace(strWords, " ' ", " ' ' ")
```

```
end function
%>
```

If the stripQuotes function is used in conjunction with the first query then it will change from the following select query:

```
select count(*) from users where userName='john' and
userPass=' ' or 1=1 --'
```

into the following select query:

```
select count(*) from users where userName='john'' and
userPass=' ' ' or 1=1 --'
```

This can stop the SQL injection attack because the clause for the **WHERE** query now requires both the userName and userPass fields to be valid.

Method 3: Remove Culprit Characters/Character Sequences

Certain characters and character sequences such as **; , --, select, insert** and **xp_** can be used to perform an SQL injection attack. Removing these characters and character sequences from user input can reduce the chance of an injection attack occurring.

The following code demonstrates a basic function can handle all of this:

```
<%
function killChars(strWords)

dim badChars
dim newChars
```

```
badChars = array ("select", "drop", ";", "--", "insert",
"delete", "xp_")
newChars = strWords

for i = 0 to uBound(badChars)
newChars = replace(newChars, badChars(i), "")
next

killChars = newChars
end function
%>
```

Using stripQuotes in combination with killChars greatly removes the chance of any SQL injection attack from succeeding. Look at the select query in the following:

```
select prodName from products where id=1; xp_cmdshell 'format
c: /q /yes '; drop database myDB; --
```

Ran through stripQuotes and then killChars, would end up looking like this:

```
prodName from products where id=1 cmdshell ' 'format c:
/q /yes '' database myDB
```

As the result it will return no records from the query.

SQL injection is one of common application layer threats. It is the act of passing a SQL query or command as input into a web application and exploits the web application that uses client-side data in a SQL query without proper input validation.

SQL injection is a topic taught in the computer science curriculum. The SQL injection attacks case study material has been developed to help instructors teach SQL injection attacks and help students learn the various SQL injection attacks as well as the ways to prevent these attacks. By using exercises students will get hands-on experience of how SQL injection works and also ways to combat them.

METHODOLOGY USED

- Collection of the Training Data Set of Users and Cheatsheet for Analysis
- The Training Data Set Consists of the URLs for investigation
- Generation of the patterns and keys
- Deep Analysis on each parameter
- Applying the proposed model on the Training data set
- Fetch Results
- Data Interpretation

ALGORITHMIC APPROACH

1. Initialize the URL (Ui)

2. Generate the Form Hierarchy FHj -> Ui

3. Create and Maintain an Injection Vulnerability Relation IVRt

4. Put the Sequence of Vulnerability Values with Fuzzy Parameter in FPj -> IVRt

5. Insert the SI (SQL Injection) in FHj

6. Measure Risk / Vulnerability Aspect with each SIS (SQL Injection String)

7. Create and Maintain the separate and arbitrary Relation of Vulnerability Investigation

8. Detailed Analysis of the Attacks

a. Classification of Attacks

b. Analysis of Authenticated and Legitimate Traffic

9. Detailed Report and Plots Generation

PROPOSED SYSTEM

1. Read/ scan Web Form Values
2. Associate mandatory parameters : username, password
3. Generate and investigate fetched parameters
4. Perform deep association and vulnerability exploitation injection avoidance technique on the training data set
5. If final results obtained

   Then Stop and terminate with success
6. Go to step 3
7. Plot charts and interpretation

FUZZY SET OF RULES

RULE - 1

if

PositiveAttempt => (0, 0, 1)

=> (FALSE, FALSE, TRUE)

NegativeAttempts => (111)*

=> (FALSE, FALSE, FALSE)*

PositiveAttempt => (0, 0, 1)

=> (FALSE, FALSE, TRUE)

OR

NegativeAttempt => (1, 1, 0)

=> (TRUE, TRUE, FALSE)

Exit with Message ("Not Allowed")

RULE - 2

```
(IdentifyUser) =>
Navigate and Generate Previous User Attempts
x =>
(Fetch (AllUserAttempts))
if
(ANY(x) = "SQL Injection")
Exit with Message "BlackList User Profile"
```

RESULTS AND DISCUSSION

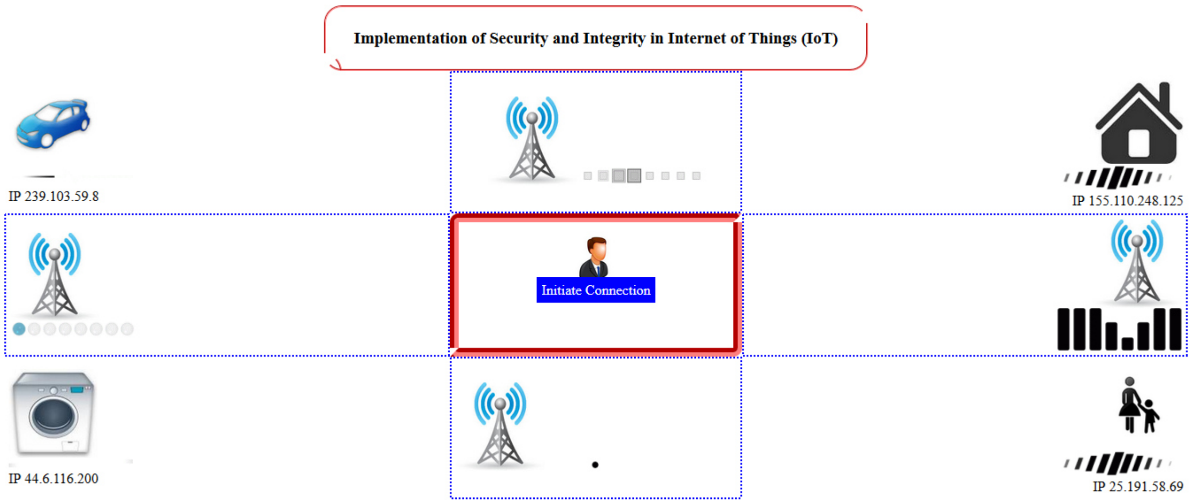
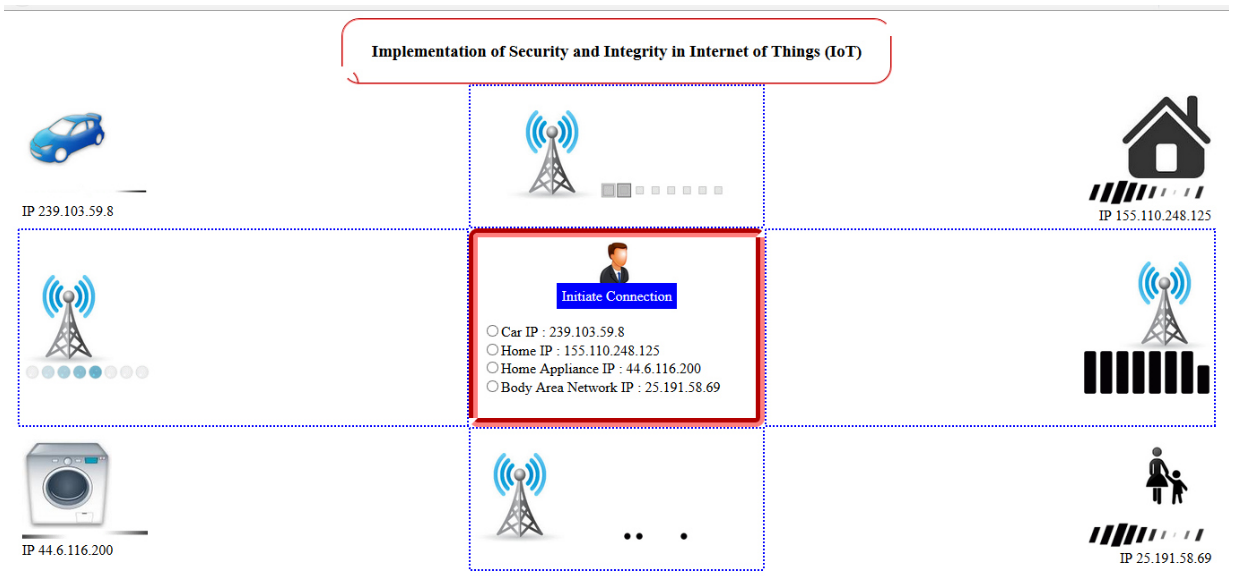
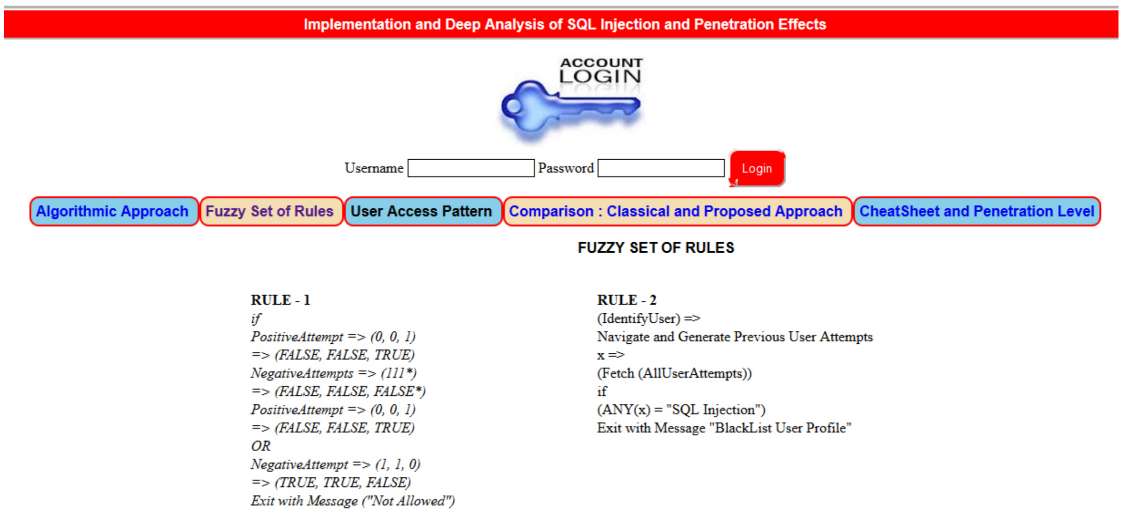


Figure 1 – Implementation Scenario of different devices in Communication



○ Car IP : 239.103.59.8
○ Home IP : 155.110.248.125
○ Home Appliance IP : 44.6.116.200
○ Body Area Network IP : 25.191.58.69

Figure 2 – User initiates the connection in IoT


Implementation and Deep Analysis of SQL Injection and Penetration Effects
ACCOUNT LOGIN
Username
Password
Login
Algorithmic Approach
Fuzzy Set of Rules
User Access Pattern
Comparison : Classical and Proposed Approach
CheatSheet and Penetration Level

FUZZY SET OF RULES

```
RULE - 1
if
PositiveAttempt => (0, 0, 1)
=> (FALSE, FALSE, TRUE)
NegativeAttempts => (111*)
=> (FALSE, FALSE, FALSE*)
PositiveAttempt => (0, 0, 1)
=> (FALSE, FALSE, TRUE)
OR
NegativeAttempt => (1, 1, 0)
=> (TRUE, TRUE, FALSE)
Exit with Message ("Not Allowed")
```

```
RULE - 2
(IdentifyUser) =>
Navigate and Generate Previous User Attempts
x =>
(Fetch (AllUserAttempts))
if
(ANY(x) = "SQL Injection")
Exit with Message "BlackList User Profile"
```

Figure 3 – Fuzzy Rules for User Behavior Analysis

USER ACCESS PATTERN – SAMPLE DATA SET

Username	Positive Attempts	Negative Attempts	String1	String2	ExecutionTime	StringType
user1	0	1	user1	1' or '1' = '1	0.00087785720825195	SQL Injection
user1	0	1	user1	1' or '1' = '1	0.00089120864868164	SQL Injection
user1	1	0	user1	user1	0.0012869834899902	Successful Login
user1	0	1	user1	assw	0.0040550231933594	Login Failed

SQL INJECTIONS CHEATSHEET – SAMPLE DATA SET

SQL Injection String	Penetration Level	Classification
admin' --	100	Attempt Admin Access
admin' #	90	Attempt Admin Access
admin'/*	80	Attempt Database Exploitation
' or 1=1--	90	Database Bypass
' or 1=1#	100	Access Database Tables
' or 1=1/*	70	Database Access
') or '1'='1--	80	Database Exploitation
') or ('1'='1--	50	Database Exploitation
1' ORDER BY 1--+	60	Access Database and Tables
1' ORDER BY 1--+	90	DB Exploitation

1' ORDER BY 2--+	60	DB Exploitation
1' ORDER BY 3--+	40	DB Exploitation
-1' UNION SELECT 1,2,3--+	80	DB Exploitation
1' ORDER BY 4--+	90	DB Exploitation
1' GROUP BY username HAVING 1=1--	80	Authentication Byepass
1' or '1' = '1	50	Authentication Byepass
1' OR '1' = '1	60	Access Username without Authentication

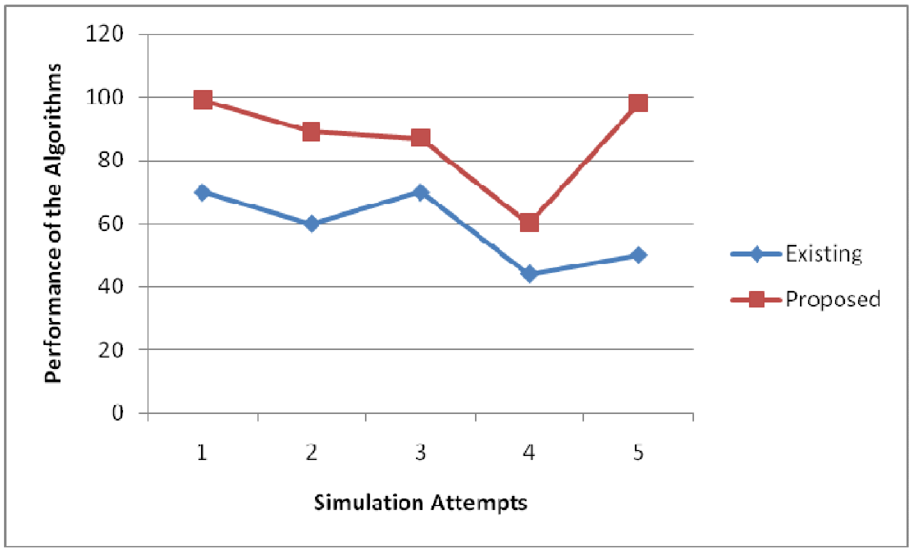


Figure 4 - Comparison between classical and proposed approach in terms of performance and overall security

CONCLUSION AND FUTURE WORK

SQL injection is a type of web application security vulnerability in which an attacker is able to submit a database SQL command that is executed by a web application, exposing the back-end database. A SQL injection attack can occur when a web application utilizes user-supplied data without proper validation or encoding as part of a command or query. The specially crafted user data tricks the application into executing unintended commands or changing data. SQL injection allows an attacker to create, read, update, alter or delete data stored in the back-end database. In its most common form, a SQL injection attack gives access to sensitive information such as social security numbers, credit card numbers or other financial data. According to Veracode's State of Software Security Report, SQL injection is one of the most prevalent types of web application security vulnerability. In this research work, the unique algorithm for the avoidance of SQL injections is underlined and implemented. The future work of the proposed technique can be extended towards making use of the swarm intelligence techniques. The swarm intelligence techniques makes use of number of iterations for the investigation and analysis of the given pseudocode or algorithmic flow. However, there are number of algorithms for the avoidance of web based attacks, there is the scope of improvements and further work. Additionally, there are hybrid approaches to improve the current research work in which two or more algorithms can be joined together and then results can be fetched using parallel techniques.

REFERENCES

[1] Bhavani M. Thuraisingham, Chris Clifton, Amar Gupta, Elisa Bertino, Elena Ferrari. "Directions for Web and E-Commerce Applications Security." *Proceedings of the 10th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (pp.200-204)*. 2001.

[2] Bojan Jovicic, Dejan Simic. "Common Web Application Attack Types and Security Using ASP.NET. " *ComSIS Consortium*. 2006.

[3] Cosmin Striletchi, Mircea-Florin Vaida. "Enhancing the Security of Web Applications." *Proceedings of the 25th International Conference on Information Technology Interfaces (pp. 463-468)* . 2003.

[4] Gary McGraw. "Software Security." *IEEE Security & Privacy (vol. 2, pp. 80-83).* 2004.

[5] J.D. Meier. "Web Application Security Engineering." IEEE Security &Privacy (vol. 4, no. 4, pp. 16–24). 2006.

[6] J.D. Meier, Alex Mackman, Michael Dunner, Srinath Vasireddy, Ray Escamilla and Anandha Murukan. “Improving Web Application Security: Threats and Countermeasures.” Microsoft Corporation. 2003.

[7] Jeff Zadeh, Dennis DeVolder. “Software Development and Related Security Issues.” *Proceedings of IEEE Southeastcon*. 2007.

[8] John R. Maguire, Gilbert Miller. “Web-Application Security: From reactive to proactive.” *IT Professional (vol. 12, no. 4, pp. 7-9)*. 2010.

[9] Mark Curphey, Rudolph Araujo. “Web application security assessment tools.” *IEEE Security & Privacy* (vol. 4, no. 4, pp. 32-41). 2006.

[10] Myat Myat Min, Khin Haymar Saw Hla. “Security on Software Life Cycle using Intrusion Detection System.” *6th AsiaPacific Symposium on Information and Telecommunication Technologies APSITT 2005 Proceedings.* 2005.

[11] Shin-Jer Yang, Jia-Shin Chen. “A study of security and performance issues in designing web-based applications.” *IEEE International Conference on e-Business Engineering.* 2007.

[12] Fu, X., Lu, X., Peltsverger, B., Chen, S., Qian, K., and Tao, L., A Static Analysis Framework for Detecting SQL Injection Vulnerabilities. *Proc. 31st Annual International Computer Software and Applications Conference 2007 (COMPSAC 2007)*, 24-27 July (2007), pp. 87-96.

[13] Thomas, S., Williams, L., and Xie, T., On automated prepared statement generation to remove SQL injection vulnerabilities. *Information and Software Technology*, Volume 51 Issue 3, March 2009, pp. 589–598.

[14] Haixia, Y. and Zhihong, N., A database security testing scheme of web application. *Proc. of 4th International Conference on Computer Science & Education 2009 (ICCSE '09)*, 25-28 July 2009, pp. 953-955.

[15] Ruse, M., Sarkar, T., and Basu. S., Analysis & Detection of SQL Injection Vulnerabilities via Automatic Test Case Generation of Programs. *Proc. 10th Annual International Symposium on Applications and the Internet*, 2010, pp. 31-37.

[16] Roichman, A., Gudes, E., Fine-grained Access Control to Web Databases. *Proceedings of 12th SACMAT Symposium*, France 2007.

[17] Shin, Y., Williams, L., and Xie, T., SQLUnitGen: Test Case Generation for SQL Injection Detection. North Carolina State University, Raleigh Technical report, NCSU CSC TR 2006-21