



Recognition and Deterrence of SQL injection attacks in database using web service

Rahul Shrivastava¹, Joy Bhattacharyji², Roopali Soni³

Mtech. Student, CSE Department, Oriental college of Technology, Bhopal, India, rahul04.shri@gmail.com.¹

Assistant Professor, CSE Department, Oriental college of Technology, Bhopal, India joy1950joy@gmail.com ²

Assistant Professor, CSE Department, Oriental college of Technology, Bhopal, India rupal123_s@yahoo.com.³

Abstract: A large number of web applications, especially those deployed by companies for e-business operations involve high reliability, efficiency and confidentiality. Such applications are often written in script languages like PHP embedded in HTML, allowing establishing connection to databases, retrieving data, and putting them in the Web. One of the most common in web application attacks is SQL Injection. SQL injection is an attack in which malicious code is inserted into strings that are later passed to an instance of SQL Server for parsing and execution. Any procedure that constructs SQL statements should be reviewed for injection vulnerabilities because SQL Server will execute all syntactically valid queries that it receives. In this paper, an attempt has been made to classify the SQL Injection attacks based on the vulnerabilities in web applications. A brief review of the existing approaches for the detection of SQL injection attack also has been presented. This paper proposes a novel specification-based methodology for the prevention of SQL injection Attacks. The two most important advantages of the new approach against existing analogous mechanisms are that, first, it prevents all forms of SQL injection attacks; second, Current technique does not allow the user to access database directly in database server. The innovative technique “Web Service Oriented XPATH Authentication Technique” is to detect and prevent SQL- Injection Attacks in database the deployment of this technique is by generating functions of two filtration models that are Active Guard and Service Detector of application scripts additionally allowing seamless integration with currently-deployed systems. We have implemented our approach using .Net platform. Our result shows that we successfully detect the attacker and prevent them to breach the security.

Keywords: Database security, Runtime Monitoring, SQL injection attacks, web application security

I. INTRODUCTION

The World Wide Web has experienced remarkable growth in recent years. Businesses, individuals and governments have found that web applications can offer effective, efficient and reliable solutions to the challenges of communicating and conducting commerce in the 21st century. Various corporate bodies whose business model completely focuses on the Web like eBay, Google, Yahoo, Amazon etc. have taken Web interactions to newer heights. As more and more enterprise applications dealing with sensitive financial and medical data turn online, the security of such Web applications has come under close scrutiny. Compromise of these applications represents a serious threat to the organizations that have deployed these web applications as well as to the users that trust these systems to store confidential data. [1]

SQL injection is an attack in which malicious code is inserted into strings that are later passed to an instance of SQL Server for parsing and execution. Any procedure that constructs SQL statements should be reviewed for injection vulnerabilities because SQL Server will execute all syntactically valid queries that it receives. It has become a common issue with database-driven web sites. The flaw is easily detected, and easily exploited, and as such, any site or software package with even a minimal user base is likely to be subject to an attempted attack of this kind. Essentially, the attack is accomplished by placing a meta character into data input to then place SQL commands in the control plane, which did not exist there before. This flaw depends on the fact that SQL makes no real distinction between the control and data planes [2].

The fear of SQL injection attacks has become increasingly frequent and serious. SQL-Injection Attacks are a class of attacks that many of these systems are highly vulnerable to, and there is no known fool-proof defend against such attacks. Compromise of these web applications represents a serious threat to organizations that have deployed them, and also to users who trust these systems to store confidential data. The Web applications that are vulnerable to SQL-Injection attacks user inputs the attacker's embeds commands and gets executed [3]. The attackers directly access the database underlying an application and leak or alter confidential information and execute malicious code

[4][5]. In some cases, attackers even use an SQL Injection vulnerability to take control and corrupt the system that hosts the Web application. The increasing number of web applications falling prey to these attacks is alarmingly high [6] Prevention of SQLIA's is a major challenge.

In this paper, we present a new technique which consists of two filtration models, Active Guard filtration model and Service Detector filtration model. Active Guard Filtration Model in application layer build a Susceptibility detector to detect and prevent the susceptibility characters or meta characters to prevent the malicious attacks from accessing the data's from database. Service Detector Filtration Model in application layer validates user input from XPATH_Validator where the Sensitive data's are stored from the Database at second level filtration model. The user input fields compare with the data existed in XPATH_Validator if it is identical then the Authenticated /legitimate user is allowed to proceed.

The rest of the paper is organized as follows: in section 2 we present SQL-INJECTION ATTACKS and then discuss the RELATED WORK in Section 3. In Section 4 and 5, proposed architecture and implementation result are discussed .Finally; we conclude the paper in Section 6.

II. SQL-INJECTION ATTACKS

In this section, we present different techniques that are used to perform SQL Injection attacks.

a. Tautologies

Tautology-based attack is to inject code in one or more conditional statements so that they always evaluate to true. The most common usages of this technique are to bypass authentication pages and extract data. If the attack is successful when the code either displays all of the returned records or performs some action if at least one record is returned. Example:

Consider a typical web application in which an user on a client machine can access services provided by a web server, having a database backend, like an online email account. When the user enters a login and a password in the web form and presses the Submit button, an URL is generated(<http://mywebpage.com/home.aspx?login=guest&pass=test>) and sent to the web server. The user input is interpreted by the home.aspx, which then in turn builds a dynamic SQL query, submits the query to the database and uses the response from the database to generate HTML-pages that are sent back to the user. Suppose the query in the asp page is of the form;

```
SELECT PROFILE FROM EMPLOYEE WHERE NAME='$login' AND PASS='$pass'
```

If the login and password as provided by the user are used, the query to be submitted to the database takes the form;

```
SELECT PROFILE FROM EMPLOYEE WHERE NAME='guest' AND PASS='test'
```

A web site that uses this asp would be vulnerable to SQLIAs. If the user were to enter [' OR 1=1 --] and [] instead of [guest] and [test], the query would take the form;

```
SELECT PROFILE FROM EMPLOYEE WHERE NAME=' ' OR 1=1 --' AND PASS=' '.
```

The characters "--" mark the beginning of a comment in SQL, and everything after that is ignored The code injected in the conditional (OR 1=1) transforms the entire WHERE clause into a tautology the query evaluates to true for each row in the table and returns all of them. Thus an attacker can bypass all authentication modules in place and gain unrestricted access to critical information on the web server. In our example, the returned set evaluates to a not null value, which causes the application to conclude that the user authentication was successful. Therefore, the application would invoke method AccountDetails.aspx and to access the application.

b. Union Query

In union-query attacks, Attackers do this by injecting a statement of the form: UNION SELECT <rest of injected query> because the attackers completely control the second/injected query they can use that query to retrieve information from a specified table. The result of this attack is that the database returns a dataset that is the union of the results of the original first query and the results of the injected second query.

Example: An attacker could inject the text "" UNION SELECT pass1 from user_info where LoginID='secret - -" into the login field, which produces the following query:

```
SELECT pass1 FROM user_info WHERE loginID="" UNION SELECT pass1 from user_info where LoginID='secret' -- AND pass1=""
```

Assuming that there is no login equal to "", the original first query returns the null set, whereas the second query returns data from the "user_info" table. In this case, the database would return column "pass1" for account "secret". The database takes the results of these two queries, unions them, and returns

them to the application. In many applications, the effect of this operation is that the value for “pass1” is displayed along with the account information.

c. Stored Procedures

A stored procedure is an operation set that is stored. Typically, stored procedures are written in SQL. Since stored procedures are stored on the server side, they are available to all clients. Once the stored procedure is modified, all clients automatically get the new version.

```
1. CREATE PROCEDURE [EMP].[RetrieveProfile] @Name varchar(50), @Passwd varchar(50)
2. WITH EXECUTE AS CALLER
3. AS
4. BEGIN
5. DECLARE @SQL varchar(200);
6. ...
7. SET @SQL='select PROFILE from EMPLOYEE where '
8. ...
9. IF LEN(@Name) > 0 AND LEN(@Passwd) > 0
10. BEGIN
11. ...
12. SELECT @SQL=@SQL+'NAME="'+@Name+'" and '
13. SELECT @SQL=@SQL+'PASS="'+@Passwd+'";
14. ...
15. END
16. ELSE
17. BEGIN
18. ...
19. SELECT @SQL=@SQL+'LOGIN="Guest"';
20. ...
21. END
22. ...
23. EXEC(@SQL)
24. ...
25. END
```

Code 1. Stored Procedure Vulnerable To SQL-Injection

A sample stored procedure called with the username and password as user inputs in a variable length string format is shown in Code 1. Notice that, there is an EXEC system function which allow the user to dynamically build a SQL statement in string format and later execute it. This feature is supported in most other business database products also. The only difference here is that the SQL statement is dynamically built in the database side. The techniques leveraged by the attacker to exploit this vulnerability is almost the same.

To launch an SQLIA, the attacker simply injects “ ’ ; SHUTDOWN; --” into either the LoginID or pass1 fields. This injection causes the following query: SELECT * FROM user_info WHERE loginID='secret' AND pass1='; SHUTDOWN; --'. The first query is executed normally, and then the second, malicious query is executed which results database shutdown. this is a piggybacked query attack.

d. Extended stored procedures

IIS(Internet Information Services) Reset There are several extended stored procedures that can cause permanent damage to a system[19]. Extended stored procedure can be executed by using login form with an injected command as the LoginId LoginId:'execmaster..xp_XXX;-- Password:[Anything] LoginId:'execmaster..xp_cmdshell'iisreset';-- Password:[Anything]. select password from user_info where LoginId="; exec master..xp_cmdshell 'iisreset'; --' and Password=" This Attack is used to stop the service of the web server of particular Web application. Stored procedures primarily consist of SQL commands, while XPs can provide entirely new functions via their code. An attacker can take advantage of extended stored procedure by entering a suitable command. This is possible if there is no proper input validation. xp_cmdshell is a built-in extended stored procedure that allows the execution of arbitrary command lines. For example: exec master..xp_cmdshell 'dir' will obtain a directory listing of the current working directory of the SQL Server process. In this example, the attacker may try entering the following input into a search

form can be used for the attack. When the query string is parsed and sent to SQL Server, the server will process the following code: `SELECT * FROM user_info WHERE input text =' exec master.. xp_cmdshell LoginId /DELETE'--`. Here, the first single quote entered by the user closes the string and SQL Server executes the next SQL statements in the batch including a command to delete a LoginId to the user_info table in the database.

e. Alternate Encodings

Alternate encodings do not provide any unique way to attack an application they are simply an enabling technique that allows attackers to evade detection and prevention techniques and exploit vulnerabilities that might not otherwise be exploitable. These evasion techniques are often necessary because a common defensive coding practice is to scan for certain known “bad characters,” such as single quotes and comment operators. To evade this defence, attackers have employed alternate methods of encoding their attack strings (e.g., using hexadecimal, ASCII, and Unicode character encoding). Therefore, attackers have been very successful in using alternate encodings to conceal their attack strings. Example: Because every type of attack could be represented using an alternate encoding, here we simply provide an example of how esoteric an alternatively encoded attack could appear. In this attack, the following text is injected into the login field: `“secret’; exec(0x73687574646f776e) - - ”`. The resulting query generated by the application is: `SELECT * FROM user_info WHERE loginID=’secret’; exec(char(0x73687574646f776e)) -- AND pass1=’`. This example makes use of the `char()` function and of ASCII hexadecimal encoding. The `char()` function takes as a parameter an integer or hexadecimal encoding of a character and returns an instance of that character. The stream of numbers in the second part of the injection is the ASCII hexadecimal encoding of the string “SHUTDOWN.” Therefore, when the query is interpreted by the database, the following query: `SELECT * FROM user_info WHERE loginID=’secret’; SHUTDOWN; -- AND pass1=’`. It would result in the execution, by the database, of the SHUTDOWN command.

f. Deny Database service

This attack used in the websites to issue a denial of service by shutting down the SQL Server. A powerful command recognized by SQL Server is SHUTDOWN WITH NOWAIT [19]. This causes the server to shutdown, immediately stopping the Windows service. After this command has been issued, the service must be manually restarted by the administrator. select password from user_info where LoginId=’shutdown with nowait; --’ and Password=’0’. The ‘--’ character sequence is the ‘single line comment’ sequence in Transact - SQL, and the ‘;’ character denotes the end of one query and the beginning of another. If he has used the default sa account, or has acquired the required privileges, SQL server will shut down, and will require a restart in order to function again. This attack is used to stop the database service of a particular web application.

III. RELATED WORK

There are existing techniques that can be used to detect and prevent input manipulation vulnerabilities.

A. Web Vulnerability Scanning

Web vulnerability scanners crawl and scan for web vulnerabilities by using software agents. These tools perform attacks against web applications, usually in a black-box fashion, and detect vulnerabilities by observing the applications’ response to the attacks [7]. However, without exact knowledge about the internal structure of applications, a black-box approach might not have enough test cases to reveal existing vulnerabilities and also have false positives.

B. Intrusion Detection System (IDS)

Valeur and colleagues [8] propose the use of an Intrusion Detection System (IDS) to detect SQLIA. Their IDS system is based on a machine learning technique that is trained using a set of typical application queries. The technique builds models of the typical queries and then monitors the application at runtime to identify queries that do not match the model in that it builds expected query models and then checks dynamically-generated queries for compliance with the model. Their technique, however, like most techniques based on learning, can generate large number of false positive in the absence of an optimal training set.

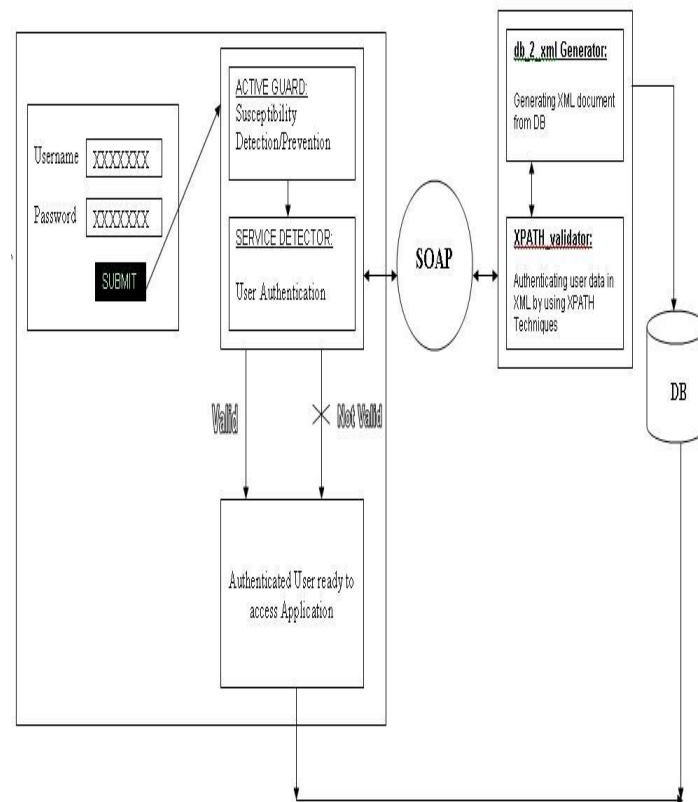
Su and Wassermann [8] propose a solution to prevent SQLIAs by analyzing the parse tree of the statement, generating custom validation code, and wrapping the vulnerable statement in the validation code. They conducted a study using five real world web applications and applied their SQLCHECK wrapper to each application. They found that their wrapper stopped all of the SQLIAs in their attack set without generating any false positives. While their wrapper was effective in preventing SQLIAs with modern attack structures, we hope to shift the focus from the structure of the attacks and onto removing the SQLIVs.

C. Combined Static and Dynamic Analysis.

AMNESIA is a model-based technique that combines static analysis and runtime monitoring [4][9]. In its static phase, AMNESIA uses static analysis to build models of the different types of queries an application can legally generate at each point of access to the database. In its dynamic phase, AMNESIA intercepts all queries before they are sent to the database and checks each query against the statically built models. Queries that violate the model are identified as SQLIA's and prevented from executing on the database. In their evaluation, the authors have shown that this technique performs well against SQLIA's. The primary limitation of this technique is that its success is dependent on the accuracy of its static analysis for building query models. Certain types of code obfuscation or query development techniques could make this step less precise and result in both false positives and false negatives. Livshits and Lam [10] use static analysis techniques to detect vulnerabilities in software. The basic approach is to use information flow techniques to detect when tainted input has been used to construct an SQL query. These queries are then flagged as SQLIA vulnerabilities. The authors demonstrate the viability of their technique by using this approach to find security vulnerabilities in a benchmark suite. The primary limitation of this approach is that it can detect only known patterns of SQLIA's and, because it uses a conservative analysis and has limited support for untainting operations, can generate a relatively high amount of false positives. Wassermann and Su propose an approach that uses static analysis combined with automated reasoning to verify that the SQL queries generated in the application layer cannot contain a tautology [11][12]. The primary drawback of this technique is that its scope is limited to detecting and preventing tautologies and cannot detect other types of attacks.

IV. PROPOSED ARCHITECTURE

This proposed technique consists of two filtration models to prevent SQLIA'S. 1) Active Guard filtration model 2) Service Detector filtration model. The steps are summarized and then describe them in more detail in following sections.



a. Active Guard Filtration Model

Active Guard Filtration Model in application layer build a Susceptibility detector to detect and prevent the Susceptibility characters or Meta characters to prevent the malicious attacks from accessing the data's from database.

b. Service Detector Filtration Model

Service Detector Filtration Model in application layer validates user input from XPATH_Validator where the Sensitive data's are stored from the Database at second level filtration model. The user input fields compare with the data existed in XPATH_Validator if it is identical then the Authenticated /legitimate user is allowed to proceed.

c. Web Service Layer

Web service builds two types of execution process that are DB_2_Xml generator and XPATH_ Validator. DB_2_Xml generator is used to create a separate temporary storage of Xml document from database where the Sensitive data's are stored in XPATH_ Validator, The user input field from the Service Detector compare with the data existed in XPATH_ Validator, if the data's are similar XPATH_ Validator send a flag with the count iterator value = 1 to the Service Detector by signifying the user data is valid.

Procedures Executed in Active Guard

```
Function stripQuotes(ByVal strWords)
    stripQuotes = Replace(strWords, "'", "''")
    Return stripQuotes
End Function

Function killChars(ByVal strWords)
    Dim arr1 As New ArrayList
    arr1.Add("select")
    arr1.Add("--")
    arr1.Add("drop")
    arr1.Add(";")
    arr1.Add("insert")
    arr1.Add("delete")
    arr1.Add("xp_")
    arr1.Add("")
    Dim i As Integer
    For i = 0 To arr1.Count - 1
        strWords = Replace(strWords, arr1.Item(i), " ", , ,
            CompareMethod.Text)
    Next
    Return strWords
End Function
```

Procedures Executed in Service Detector

```
<WebMethod(> _
Public Sub Db_2_XML()
    adapt=New SqlDataAdapter("select LoginId,Password from user_info", cn)
    dst = New DataSet("Main_Tag")
    adapt.Fill(dst, "Details")
    dst.WriteXml(Server.MapPath("XML_DATA\XML_DATA.xml"))
End Sub
```

Procedures Executed in Web Service

```
<WebMethod(EnableSession:=True)> _
Public Function XPath_XML_Validation(ByVal userName As String, ByVal Password As Integer)
    As Integer
    Dim xpathdoc As New XPathDocument(Server.MapPath("XML_DATA\XML_DATA.xml"))
    Dim navi As XPathNavigator = xpathdoc.CreateNavigator()
    Dim expr As XPathExpression = navi.Compile("/Main_Tag/Details[LoginId='" & userName & "' and Password='"
    & Password & "']")
    Dim nodes As XPathNodeIterator = navi.Select(expr)
    Dim count2 As Integer = nodes.Count.ToString()
    Return count2
```

End Function

V. IMPLEMENTATION RESULT

In this section we present the results of the experiment we conduct. We conduct the experiment on a PC with i7-2.8GHz processor, 4GB RAM. Visual studio 2008 and SQL 2008 are used for the experiment. We create two scenarios- scenario-1: without SQL detection/prevention scheme (allow option) and scenario-2: with SQL detection/prevention scheme (prevent option). The attacker can attack when without prevention scheme (allow option) is used. We show attacker fails to attack when scenario with our proposed SQL detection/prevention scheme (prevent option) is used, that shows our proposed techniques perform better.

A. Simulation Result

We have simulated a banking application as SQL injection makes a great impact on that application. The following results describe the our flow of operation while performing experiment.

On start of our application, the login page appears. Admin and User use this login page to use the application. Figure1 is the login page we get on starting application. Then admin logs into the system(Figure2).

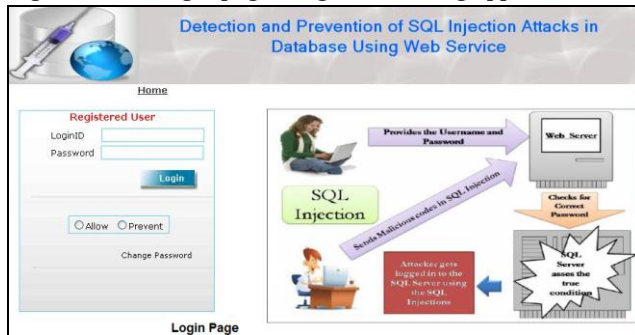


Figure1: Login Page

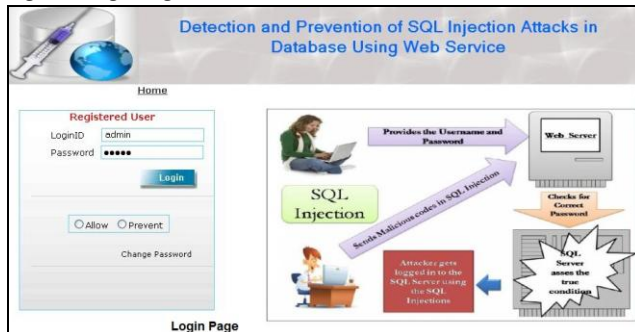


Figure2: Admin login activity

Figure3 shows different operation admin can perform like new user creation(Figure4). Once a user is created and verified by admin, new user can log in to use the application.

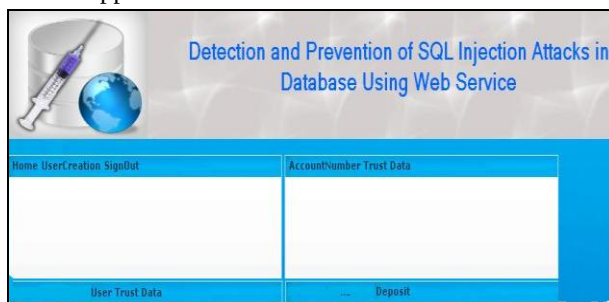


Figure 3: Admin Home page after successful login



Figure 4: New User Created by Admin

Figure5 shows the attacker wants to enter the system by forging (by using tautology) and fails as our proposed system protects it from intrusion by detecting malicious string present in login field. Figure 6 shows when our system is in non-prevention mode, attacker bypass security and access into user account.



Figure 5: Tautology Attack Detected and Prevented by Active guard

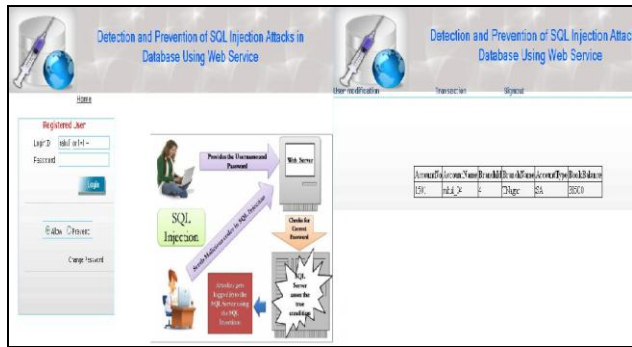


Figure 6: Tautology Attack Bypassed Successfully

Figure7 shows although an attacker bypass the first level security of our proposed system(Active guard filter), while it tries to create a new user with malicious string and wants to store malicious string into database to further exploit the database by using that malicious user fails to do so, as our second level security(service detector filter) successfully detect and protect it to create malicious user and prevent it to update into database with malicious data. Figure 8 malicious string detected while trying to access to log into our application and prevents it disclosing account details of a particular account no.



Figure 7: Service Detector prevents to insert malicious data inside database

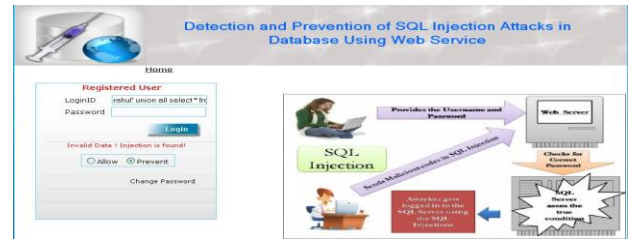


Figure 8: Union Attack Detected and Prevented Successfully

Figure9 shows detection of malicious string that indirectly save the database from truncate, prevent from inconsistent database. Similarly Figure10 shows inference is detected and prevented.



Figure 9: Piggybacked Attack Detected and Prevented Successfully

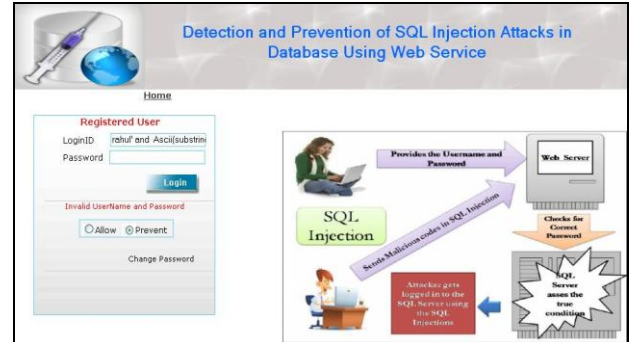


Figure 10: Inference Detected and Prevented Successfully

5.2 SQLIA Prevention Accuracy

Both the protected and unprotected web Applications are tested using different types of SQLIA's; namely use of Tautologies, Union, Piggy-Backed Queries, Inserting additional SQL statements, Second-order SQL injection and various other SQLIA s. Table 1 shows that the proposed technique prevented all types of SQLIA s in all cases. The proposed technique is thus a secure and robust solution to defend against SQLIA's.

SQL Injection Types	Unprotected (allow option selected)	Protected (prevent option selected)
1. TAUTOLOGIES	Not Prevented	Prevented
2. UNION	Not Prevented	Prevented
3. STORED PROCEDURE	Not Prevented	Prevented
4.PIGGY BACKED QUERIES	Not Prevented	Prevented
5. ALTERNATIVE ENCODING	Not Prevented	Prevented
6. Inference	Not Prevented	Prevented

Table 1: SQLIA'S Prevention Accuracy

VI. CONCLUSION

We have presented a technique for detecting and protecting SQL injection attack in web service. The proposed technique is used to detect and prevent the SQLI flaw (Susceptibility characters & exploiting SQL commands) in Susceptibility Detector and prevent the Susceptibility attacker Web Service Oriented XPATH Authentication Technique checks the user input with valid database which is stored separately in XPATH and do not affect database directly then the validated user input field is allowed to access the web application as well as used to improve the performance of the server side validation. The proposed technique that contains two filter models- Active Guard filtration model and Service Detector filtration model. Active guard filtration model works in front end to check any abnormal entry into the website. If attacker successfully bypass the security layer provided by active filtration model, then it encounter second level of security model-Service Detector filtration model. It checks for any abnormal/malicious strings present in the string that makes entry in the database. If it find any abnormal/malicious strings present in the string then it protect the system by not permitting/executing the string and no malicious entry not made in database. The experimental result validate the efficiency of proposed method.

VII. REFERENCES:

- [1]Structured query language (SQL) related tools-Ke Wei, 2006
- [2] Li Sha Dong Xiaori RaoHong , An Adaptive Method Preventing Database from SQL Injection Attacks-,2010 IEEE
- [3] William G.J. Halfond, Alessandro Orso, “WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation IEEE Software Engineering ,VOL. 34, NO.1January/February 2008
- [4] William G.J. Halfond and Alessandro Orso , “AMNESIA: Analysis and Monitoring for Neutralizing SQLInjection Attacks”, ASE’05, November 7–11, 2005
- [5]William G.J. Halfond and Alessandro Orso, “A Classification of SQL injection attacks and countermeasures”,proc IEEE int’l Symp. Secure Software Engg., Mar. 2006.
- [6]Muthuprasanna, Ke Wei, Suraj Kothari, “Eliminating SQL Injection Attacks - A TransparentDefenceMechanism”, SQL Injection Attacks Prof. Jim Whitehead CMPS 183. Spring 2006, May 17, 2006
- [7] Kals, S., Kirda, E., Kruegel, C., and Jovanovic, N. 2006. SecuBat: a web vulnerability scanner. In Proceedings of the 15th International Conference on World Wide Web. WWW '06. ACM Press, pp. 247-256.
- [8] F. Valeur and D. Mutz and G. Vigna “A Learning-Based Approach to the Detection of SQL Attacks,” In Proceedings of the Conference on Detection of Intrusions and Malware Vulnerability Assessment (DIMVA), July 2005.
- [9] W. G. J. Halfond and A. Orso, “Combining Static Analysis and Runtime Monitoring to Counter SQL Injection Attacks,” 3rd International Workshop on Dynamic Analysis, 2005, pp. 1- 7
- [10] V. B. Livshits and M. S. Lam. Finding Security Errors in Java Programs with Static Analysis. In Proceedings of the 14th Usenix Security Symposium, pages 271–286, Aug. 2005.
- [11] G. Wassermann and Z. Su. An Analysis Framework for Security in Web Applications. In Proceedings of the FSE Workshop on Specification and Verification of component-Based Systems (SAVCBS 2004), pages 70–78, 2004.
- [12] IndraniBalasundaram, Dr. E. Ramaraj- An Approach to Detect and Prevent SQL Injection Attacks in Database Using Web Service, IJCSNS International Journal of Computer Science and Network Security, VOL.11 No.1, January 2011