ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 4, Issue 9, September 2015

SQL Injection Attack Prevention for Web Applications

Prem Shanker Dwivedi[1], Atma Prakash Singh[2]

Student (M.Tech.), Dept of Computer Science & Engg, Azad Institute of Engineering & Technology, Lucknow, India[1]

Assist. Prof., Dept. of Computer Science & Engg, Azad Institute of Engineering & Technology, Lucknow, India[2]

Abstract: Presently web users heavily depend on database- driven web applications for an increasing amount of activities, such as banking, reservation and shopping. When performing such activities, we entrust our personal information to these web applications and their underlying databases. Web applications are often vulnerable to attacks, which can give an attacker complete access to the applications' underlying database. In an SQL Injection Attack, an attacker attempts to exploit vulnerabilities in custom web applications by entering SQL code in an entry field such as a login. If successful, such an attack can give the attacker access to the data on the database used by the application and the ability to run malicious code on the Web site. Attacks occur when developers combine hard-coded strings with user-provided input to create dynamic queries. Intuitively, if user input is not properly validated, attackers may be able to change the developer's intended SQL command by inserting new SQL keywords or operators through specially crafted input strings. As the SQL Injection Attack passes through all the stages as like a normal request from genuine user the core components of the server may not be able to detect the attack on the Database. Several methods have been proposed to detect and prevent SQL injection attacks. We devise a method that uses defensive coding and secure hash algorithm to prevent SQL injection attacks. This method is illustrated by overview, diagrams and step by step procedure for implementing the technique to protect web application against SQL Injection. We show that this technique can be used effectively to prevent SQL Injection Attacks through bypass authentication in web application without degrading the system's performance. Finally the method is implemented by using a web application and MySQL database.

Keywords: SQL, Hash, Web application, Database, vulnerable.

I. INTRODUCTION

In today's technology environment the Internet become an integral part of human life and many enterprises dependent on it in different ways like storing employee profiles, accessing the files on remote servers and maintaining user information, and so on. The Internet is also an inexpensive solution for the enterprises to maintain a wide area network, and individual use the Internet for many other uses like shopping, meeting friends, reading news, and so on. Due to the rapid developments in Internet transfer speeds and the flexibility depending on the web applications is improved a lot. Because of the extensive use of internet in day-to-day life it became easier for hackers to attack on personal computers and to theft identity information like credit cards and personnel files. The enterprises can protect their employees by taking high security measures like one-time passwords and unique identification numbers. However, for a normal consumer it is more likely to lose their personal information due to the attacks on internet. Due to the pressure on the employees who are developing the application, they try to deliver the application quickly more than considering about all the security measures that need to consider when developing the application. This leads the program to be vulnerable to internet attacks. One type of the attack need to be considered when developing is SQL injection by which the hacker can attack the background database application and get the credit card details of a customer to use it for unauthorized transactions.

The SQL injection attacks are done on the internet applications more than the intranet applications. Normally the administrator would not able to recognize that there is an attack happened on the database, because of the fact that the hacker can execute the SQL command as normal user. As the SQL injection, attack is executed as normal script that is executed by the application it is highly difficult for the administrator that there is an attack is running on the background. The web applications are being subjected to bombardment of attacks that can pose risk to an entire enterprise. The hackers deploy a wide variety of attack vectors against the web applications ranging from SQL Injection Attacks (SQLIA) to Cross-site scripting (XSS), Remote File Execution, etc. Most of these attacks result in severe information breach which can lead to exposing confidential information like credit card details of its customers, malicious content being posted on the organization's website, take control of the personal computers using botnets for sending to pide, stealing passwords etc. Hence there is a strong need to test the web applications for potential vulnerabilities and flaws before they are being deployed. This testing is currently performed using automated tools for identifying some of the vulnerabilities and the rest of them are being done manually by security experts which consumes considerable time and effort. This attracts more Research to develop efficient tools and techniques to identify the vulnerabilities effectively in an automated fashion.

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 4, Issue 9, September 2015

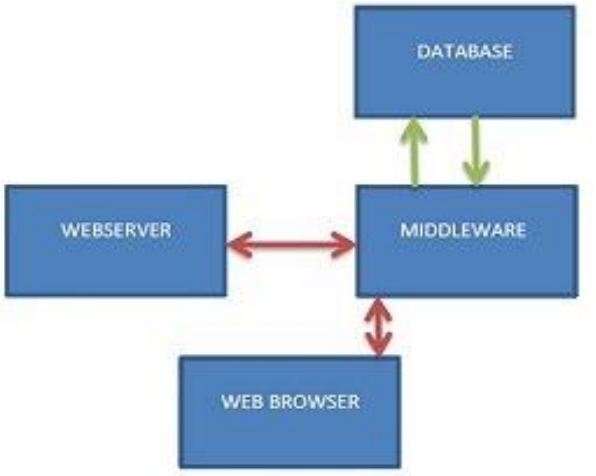

Fig.1. Architecture of web application

SQL Injection attack is one of the most important attacks in the Open Web Application Security Project (OWASP) top 10 vulnerability list and it has resulted in massive attacks on a number of websites in the past few years. SQL Injection vulnerabilities are easy to detect and exploit , that is why SQL Injection Attacks are frequently employed by malicious users. Furthermore, SQL Injection Attack techniques have become more common, more ambitious, and increasingly sophisticated, so there is a deep need to find and effective and feasible solution for this problem in the computer security community. Detection and prevention against SQL Injection Attacks is a topic of active research in the industry and academia. To achieve those purposes, automatic tools and security systems have been implemented, but none of them are complete or accurate enough to guarantee an absolute level of security on web applications. One of the important reasons of this shortcoming is that there is a lack of common and complete methodology for the evaluation of performance. So we feel that there should be such type of mechanism which will be easily deployable and provide a good performance. To achieve this, our research work is driven to the way of developing a new modified SQL Injection Prevention Technique.

II. OVERVIEW OF SQL INJECTION

SQL Injection is an attack in which malicious code is inserted into strings that are later passed to an instance of a database server for parsing and execution.

The basic principle of SQL injection is to take advantage of insecure code on a system connected to the Internet in order to pass commands directly to a database and then to take advantage of a poorly secured system to leverage an attacker's access. A single suspect responsible for the majority of SQL Injection problems: the single quote (‘), also known as tick. The SQL Injection process uses an iterative methodology. You first try a single invalid character and examine the effect. Then you try a simple SQL command and examine the effect. Eventually, you will reach the point where you have the correct number of ticks, parenthesis or other formatting characters. An SQL injection attack has a set of properties, such as assets under threat, vulnerabilities being exploited and attack techniques utilized by threat agents.

A. SOURCES OF SQL INJECTION INPUT

There are several sources for SQL Injection inputs given as follows:

1) **Injection through user input:** In this case, attackers inject SQL commands by providing suitably crafted user input. A Web application can read user input in several ways based on the environment in which the application is deployed. In most SQL Injection Attacks that target Web applications, user input typically comes from ‘form submissions’ that are sent to the Web application via HTTP GET or POST requests.
2) **Injection through cookies:** Cookies are files that contain state information generated by Web applications and stored on the client machine. When a client returns to a Web application, cookies can be used to restore the client's state information. Since the client has control over the storage of the cookie, a malicious client could tamper with the cookie's contents. If a Web application uses the cookie's contents to build SQL queries, an attacker could easily submit an attack by embedding it in the cookie.
3) **Injection through server variables:** Server variables are a collection of variables that contain HTTP, network headers, and environmental variables. Web applications use these server variables in a variety of ways, such as logging usage statistics and identifying browsing trends. If these variables are logged to a database without sanitization, this could create SQL injection vulnerability. Because attackers can forge the values that are placed in HTTP and network headers, they can exploit this vulnerability by placing an SQLIA directly into the headers.
4) **Second-order injection:** In second-order injections, attackers add malicious inputs into a system or database to indirectly trigger an SQLIA when that input is used at a later time. Second-order injections are not trying to cause the attack to occur when the malicious input initially reaches the database. Instead, attackers rely on knowledge of where the input will be subsequently used and craft their attack so that it occurs during that usage. An example of a second order injection attack is the following:

   A user registers on a website using a seeded user name, such as “admin’ –”. The application properly escapes the single quote in the input before storing it in the database, preventing its potentially malicious effect. At this point, the user modifies his/her password, an operation that typically involves the following: (1) checking that the user knows the current password and (2) changing the password if the check is successful. To do this, the Web application might construct an SQL command as follows:

SQL String = “UPDATE users SET password=’” + newPassword +”’ WHERE 403piderin=’” + 403piderin + “’ AND password=’” + oldPassword + “’”

In the query, newPassword and oldPassword are the new

DOI 10.17148/IJARCCE.2015.4987

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 4, Issue 9, September 2015

and old passwords, respectively, and 404piderin is the name of the user currently logged-in (i.e., ‘‘admin’--’’). Therefore, the query string that is sent to the database is (assuming that newPassword and oldPassword are “newpwd” and “oldpwd”):

UPDATE users SET password=’newpwd’ WHERE 404piderin=’admin’—‘AND password=’oldpwd’

Because “--” is the SQL comment operator, everything after it is ignored by the database. Therefore, the result of this query is that the database changes the password of the administrator (“admin”) to an attacker-specified value. Second-order injections can be especially difficult to detect and prevent because the point of injection is different from the point where the attack actually manifests itself.

B. TYPES OF SQL INJECTION ATTACK

SQL Injection Attacks are discriminated on the basis of the query injected. The different types of attacks are generally not performed in isolation; many of them are used together or sequentially, depending on the specific goals of the attacker.

1) **Tautology based Attack:** The general goal of a tautology-based attack is to inject code in one or more conditional statements so that they always evaluate to true. The consequences of this attack depend on how the results of the query are used within the application. The most common usages are to bypass authentication pages and extract data. In this type of injection, an attacker exploits an inject-able field that is used in a query’s WHERE conditional. Transforming the conditional into a tautology causes all of the rows in the database table targeted by the query to be returned. In general, for a tautology-based attack to work, an attacker must consider not only the inject-able/vulnerable parameters, but also the coding constructs that evaluate the query results. Typically, 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.

   For example, an attacker submits “ ’ or 1=1 - - ” for the login input field (the input submitted for the other fields is irrelevant). The resulting query is:

SELECT accounts FROM users WHERE login=’’ or 1=1–AND pass=’’ AND pin=12

   The code injected in the conditional (OR 1=1) transforms the entire WHERE clause into a tautology. The database uses the conditional as the basis for evaluating each row and deciding which ones to return to the application. Because the conditional is a tautology, the query evaluates to true for each row in the table and returns all of them.

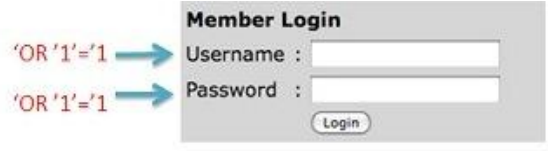

Fig.2. Tautology based SQL injection.

2) **Illegal/ Logically Incorrect Queries:** This technique is usually used during the information gathering stage of the attack. Through injecting illegal/logically incorrect requests, an attacker may gain knowledge that aids the attack, such as finding out the inject-able parameters, data types of columns within the tables, names of tables, etc. This is usually done using the HAVING and GROUP BY clause. This attack lets an attacker gather important information about the type and structure of the back-end database of a Web application. The attack is considered a preliminary, information gathering step for other attacks. The vulnerability leveraged by this attack is that the default error page returned by application servers is often overly descriptive. In fact, the simple fact that an error messages is generated can often reveal vulnerable/inject-able parameters to an attacker. Additional error information, originally intended to help programmers debug their applications, further helps attackers gain information about the schema of the back-end database. When performing this attack, an attacker tries to inject statements that cause a syntax, type conversion, or logical error into the database. Syntax errors can be used to identify inject-able parameters. Type errors can be used to deduce the data types of certain columns or to extract data. Logical errors often reveal the names of the tables and columns that caused the error. Consider the following example:

In this example, an attacker’s goal is to cause a type conversion error that can reveal relevant data. To do this, the attacker injects the following text into input field pin: “convert (int, (select top 1 name from sysobjects where xtype=’u’))”. The resulting query is:

SELECT accounts FROM users WHERE login=’’
AND pass=’’ AND pin= convert (int, (select top 1 name from sysobjects where xtype=’u’))

In the attack string, the injected select query attempts to extract the first user table (xtype=’u’) from the database’s metadata table (assume the application is using Microsoft SQL Server, for which the metadata table is called sysobjects). The query then tries to convert this table name into an integer. Because this is not a legal type conversion, the database throws an error. For Microsoft SQL Server, the error would be: ”Microsoft OLE DB Provider for SQL Server (0x80040E07) Error converting nvarchar value ’CreditCards’ to a column of data type int.”

There are two useful pieces of information in this message that aid an attacker. First, the attacker can see that the database is an SQL Server database, as the error message explicitly states this fact. Second, the error message reveals the value of the string that caused the type conversion to occur. In this case, this value is also the name of the first user-defined table in the database: “CreditCards.” A similar strategy can be used to systematically extract the name and type of each column in the database. Using this information about the schema of the database, an attacker can then create further attacks that target specific pieces of information.

DOI 10.17148/IJARCCE.2015.4987

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 4, Issue 9, September 2015

3) Piggy Backed Queries: In this attack type, an attacker tries to inject additional queries into the original query. We distinguish this type from others because, in this case, attackers are not trying to modify the original intended query; instead, they are trying to include new and distinct queries that "piggy-back" on the original query. As a result, the database receives multiple SQL queries. The first is the intended query which is executed as normal; the subsequent ones are the injected queries, which are executed in addition to the first. This type of attack can be extremely harmful. If successful, attackers can insert virtually any type of SQL command, including stored procedures, into the additional queries and have them executed along with the original query. Vulnerability to this type of attack is often dependent on having a database configuration that allows multiple statements to be contained in a single string. Consider the following Example:

If the attacker inputs "'; drop table users - -" into the pass field, the application generates the query:

SELECT accounts FROM users WHERE login='doe' AND pass=''; drop table users –' AND pin=123

After completing the first query, the database would recognize the query delimiter (";") and execute the injected second query. The result of executing the second query would be to drop table users, which would likely destroy valuable information. Other types of queries could insert new users into the database or execute stored procedures. Note that many databases do not require a special character to separate distinct queries, so simply scanning for a query separator is not an effective way to prevent this type of attack.

4) Union Query: In union-query attacks, an attacker exploits a vulnerable parameter to change the data set returned for a given query. With this technique, an attacker can trick the application into returning data from a table different from the one that was intended by the developer. 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. Referring to the previous example, an attacker could inject the text "' UNION SELECT cardNo from CreditCards where acctNo=10032 - -" into the login field, which produces the following query:

SELECT accounts FROM users WHERE login='' UNION SELECT card No from CreditCards where acct No=10032 – AND pass='' AND pin=

Assuming that there is no login equal to "", the original first query returns the null set, whereas the second query returns data from the "CreditCards" table. In this case, the database would return column "cardNo" for account "10032." 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 "cardNo" is displayed along with the account information.

5) Stored Procedures: SQL Injection Attacks of this type try to execute stored procedures present in the database. Today, most database vendors ship databases with a standard set of stored procedures that extend the functionality of the database and allow for interaction with the operating system. Therefore, once an attacker determines which backend database is in use, SQL Injection Attacks can be crafted to execute stored procedures provided by that specific database, including procedures that interact with the operating system. It is a common misconception that using stored procedures to write Web applications renders them invulnerable to SQL Injection Attacks. Developers are often surprised to find that their stored procedures can be just as vulnerable to attacks as their normal applications. Additionally, because stored procedures are often written in special scripting languages, they can contain other types of vulnerabilities, such as buffer overflows, that allow attackers to run arbitrary code on the server or escalate their privileges.

```
CREATEPROCEDURE DBO.isAuthenticated
@userName varchar2, @pass varchar2, @pin
int AS EXEC("SELECTaccountsFROM
usersWHERElogin='"+@userName+"'and
pass='" +@password+ "' and pin="
+@pin");GO
```

Fig.3. Stored procedure for checking credentials.

Consider the Example: This example demonstrates how a parameterized stored procedure can be exploited via an SQL Injection Attack. In the example, we assume that the query string constructed at lines 5 and 6 of our example has been replaced by a call to the stored procedure defined in Figure 2. The stored procedure returns a true/false value to indicate whether the user's credentials authenticated correctly. To launch an SQLIA, the attacker simply injects " ' ; SHUTDOWN; - -" into either the 405piderin or password fields. This injection causes the stored procedure to generate the following query:

SELECT accounts FROM users WHERE login='doe' AND pass=' '; SHUTDOWN; -- AND pin=23

At this point, this attack works like a piggy-back attack. The first query is executed normally, and then the second, malicious query is executed, which results in a database shut down. This example shows that stored procedures can be vulnerable to the same range of attacks as traditional application code.

6) Inference: In this attack, the query is modified to recast it in the form of an action that is executed based on the answer to a true/ false question about data values in the

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 4, Issue 9, September 2015

database. In this type of injection, attackers are generally trying to attack a site that has been secured enough so that, when an injection has succeeded, there is no usable feedback via database error messages. Since database error messages are unavailable to provide the attacker with feedback, attackers must use a different method of obtaining a response from the database. In this situation, the attacker injects commands into the site and then observes how the function/response of the website changes. By carefully noting when the site behaves the same and when its behavior changes, the attacker can deduce not only whether certain parameters are vulnerable, but also additional information about the values in the database. There are two well-known attack techniques that are based on inference. They allow an attacker to extract data from a database and detect vulnerable parameters.

Blind SQL Injection: In this technique, the information must be inferred from the behavior of the page by asking the server true/ false questions. If the injected statement evaluates to true, the site continues to function normally. If the statement evaluates to false, although there is no descriptive error message, the page differs significantly from the normally-functioning page.

Timing Attacks: A timing attack allows an attacker to gain information from a database by observing timing delays in the response of the database. This attack is very similar to blind sql injection, but uses a different method of inference. To perform a timing attack, attackers structure their injected query in the form of an if/then statement, whose branch predicate corresponds to an unknown about the contents of the database. Along one of the branches, the attacker uses a SQL construct that takes a known amount of time to execute, (e.g. the WAITFOR keyword, which causes the database to delay its response by a specified time). By measuring the increase or decrease in response time of the database, the attacker can infer which branch was taken in his injection and therefore the answer to the injected question.

Referring to the running example, we illustrate two ways in which Inference based attacks can be used. The first of these is identifying inject-able parameters using blind injection.

Consider two possible injections into the login field. The first being "legalUser' and 1=0 - -" and the second, "legalUser' and 1=1 - -". These injections result in the following two queries:

SELECT accounts FROM users WHERE login='legalUser' and 1=0 – ' AND pass='' AND pin=0

SELECT accounts FROM users WHERE login='legalUser' and 1=1 – ' AND pass='' AND pin=0

Now, let us consider two scenarios. In the first scenario, we have a secure application, and the input for login is validated correctly. In this case, both injections would return login error messages, and the attacker would know that the login parameter is not vulnerable. In the second scenario, we have an insecure application and the login parameter is vulnerable to injection. The attacker submits the first injection and, because it always evaluates to false, the application returns a login error message. At this point however, the attacker does not know if this is because the application validated the input correctly and blocked the attack attempt or because the attack itself caused the login error. The attacker then submits the second query, which always evaluates to true. If in this case there is no login error message, then the attacker knows that the attack went through and that the login parameter is vulnerable to injection. The second way inference based attacks can be used is to perform data extraction. Here we illustrate how to use timing based inference attack to extract a table name from the database.

In this attack, the following is injected into the login parameter:

''legalUser' and ASCII (SUBSTRING ((select top 1 name from sysobjects), 1, 1)) > X WAITFOR 5 –''.

This produces the following query:

SELECT accounts FROM users WHERE login='legalUser' and ASCII (SUBSTRING((select top 1 name from sysobjects),1,1)) > X WAITFOR 5 – ' AND pass='' AND pin=0

In this attack the SUBSTRING function is used to extract the first character of the first table's name. Using a binary search strategy, the attacker can then ask a series of questions about this character. In this case, the attacker is asking if the ASCII value of the character is greater-than or less-than-or-equal-to the value of X. If the value is greater, the attacker knows this by observing an additional 5 second delay in the response of the database. The attacker can then use a binary search by varying the value of X to identify the value of the first character.

7) **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. 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.

Select * from user_info where LoginId='1; xp_cmdshell 'format c:/q /yes '; drop database mydb; --AND pass1 = 0

III. RELATED WORK

Several techniques have been proposed by researches to detect and prevent SQL injection attacks. These techniques can be broadly classified into two categories, static approach and dynamic approach.

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 4, Issue 9, September 2015

A. Static Code Checkers

JDBC-Checker is a technique for statically checking the type correctness of dynamically-generated SQL queries [12, 13]. This technique was not developed with the intent of detecting and preventing general SQLIAs, but can nevertheless be used to prevent attacks that take advantage of type mismatches in a dynamically-generated query string. JDBC-Checker is able to detect one of the root causes of SQLIA vulnerabilities in code improper type checking of input. However, this technique would not catch more general forms of SQLIAs because most of these attacks consist of syntactically and type correct queries.

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 [26]. The primary drawback of this technique is that its scope is limited to detecting and preventing tautologies and cannot detect other types of attacks.

B. Combined Static and Dynamic Analysis

AMNESIA is a model-based technique that combines static analysis and runtime monitoring [17, 16]. 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 SQLIAs and prevented from executing on the database. In their evaluation, the authors have shown that this technique performs well against SQLIAs. 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.

Similarly, two related approaches, SQLGuard [6] and SQLCheck [7] also check queries at runtime to see if they conform to a model of expected queries. In these approaches, the model is expressed as a grammar that only accepts legal queries. In SQLGuard, the model is deduced at runtime by examining the structure of the query before and after the addition of user-input. In SQLCheck, the model is specified independently by the developer. Both approaches use a secret key to delimit user input during parsing by the runtime checker, so security of the approach is dependent on attackers not being able to discover the key. Additionally, the use of these two approaches requires the developer to either rewrite code to use a special intermediate library or manually insert special markers into the code where user input is added to a dynamically generated query.

C. Taint Based Approaches

WebsSARI detects input-validation related errors using information flow analysis [18]. In this approach, static analysis is used to check taint flows against preconditions for sensitive functions. The analysis detects the points in which preconditions have not been met and can suggest filters and sanitization functions that can be automatically added to the application to satisfy these preconditions. The WebSSARI system works by considering as sanitized input that has passed through a predefined set of filters. In their evaluation, the authors were able to detect security vulnerabilities in a range of existing applications. The primary drawbacks of this technique are that it assumes that adequate preconditions for sensitive functions can be accurately expressed using their typing system and that having input passing through certain types of filters is sufficient to consider it not tainted. For many types of functions and applications, this assumption is too strong.

Livshits and Lam [19] 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 SQLIAs and, because it uses a conservative analysis and has limited support for untainting operations, can generate a relatively high amount of false positives. Several dynamic taint analysis approaches have been proposed. Two similar approaches by Nguyen-Tuong and colleagues [22] and Pietraszek and Berghe [23] modify a PHP interpreter to track precise per-character taint information. The techniques use a context sensitive analysis to detect and reject queries if untrusted input has been used to create certain types of SQL tokens. A common drawback of these two approaches is that they require modifications to the runtime environment, which affects portability. A technique by Haldar and colleagues [15] and SecuriFly [20] implement a similar approach for Java. However, these techniques do not use the context sensitive analysis employed by the other two approaches and track taint information on a per-string basis (as opposed to per character). SecuriFly also attempts to sanitize query strings that have been generated using tainted input. However, this sanitization approach does not help if injection is performed into numeric fields. In general, dynamic taint-based techniques have shown a lot of promise in their ability to detect and prevent SQLIAs. The primary drawback of these approaches is that identifying all sources of tainted user input in highly-modular Web applications and accurately propagating taint information is often a difficult task.

D. New Query Development Paradigms

Two recent approaches, SQL DOM [21] and Safe Query Objects [7], use encapsulation of database queries to provide a safe and reliable way to access databases. These techniques offer an effective way to avoid the SQLIA problem by changing the query-building process from an unregulated one that uses string concatenation to a systematic one that uses a type-checked API. Within their API, they are able to systematically apply coding best practices such as input filtering and rigorous type checking of user input. By changing the development paradigm in

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 4, Issue 9, September 2015

which SQL queries are created, these techniques eliminate the coding practices that make most SQLIAs possible. Although effective, these techniques have the drawback that they require developers to learn and use a new programming paradigm or query-development process. Furthermore, because they focus on using a new development process, they do not provide any type of protection or improved security for existing legacy systems.

E. Intrusion Detection Systems

Valeur and colleagues [25] propose the use of an Intrusion Detection System(IDS) to detect SQLIAs. 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 their evaluation, Valeur and colleagues have shown that their system is able to detect attacks with a high rate of success. However, the fundamental limitation of learning based techniques is that they can provide no guarantees about their detection abilities because their success is dependent on the quality of the training set used. A poor training set would cause the learning technique to generate a large number of false positives and negatives.

F. Proxy Filters

Security Gateway [24] is a proxy filtering system that enforces input validation rules on the data flowing to a Web application. Using their Security Policy Descriptor Language (SPDL), developers provide constraints and specify transformations to be applied to application parameters as they flow from the Web page to the application server. Because SPDL is highly expressive, it allows developers considerable freedom in expressing their policies. However, this approach is human-based and, like defensive programming, requires developers to know not only which data needs to be filtered, but also what patterns and filters to apply to the data.

G. Instruction Set Randomization.

SQLrand [5] is an approach based on instruction-set randomization. SQLrand provides a framework that allows developers to create queries using randomized instructions instead of normal SQL keywords. A proxy filter intercepts queries to the database and de-randomizes the keywords. SQL code injected by an attacker would not have been constructed using the randomized instruction set. Therefore, injected commands would result in a syntactically incorrect query. While this technique can be very effective, it has several practical drawbacks. First, since it uses a secret key to modify instructions, security of the approach is dependent on attackers not being able to discover the key. Second, the approach imposes a significant infrastructure overhead because it require the integration of a proxy for the database in the system.

H. CANDID

This paper [2, 4] modifies web applications written in Java through a program transformation. This tool dynamically mines the programmer-intended query structure on any input and detects attacks by comparing it against the structure of the actual query issued. CANDID's natural and simple approach turns out to be very powerful for detection of SQL injection attacks.

IV. PROPOSED TECHNIQUE

Our research work proposes the technique, SOL Injection Attack Prevention for Web Application (SIAPWA). In the papers [1][3] encryption methods have been used in conjunction with adding some extra columns in a Login table to avoid SQL injection attacks. These techniques requires much more extra space, when we have a large number of users in a Login table and also introduces a significant amount of delay during comparison of credentials. However our proposed technique requires no extra columns in a login table, due to which no extra space will be required and comparison delay is also negligible. In this technique the credentials provided by the user are directly stored into a Login table in encrypted form. For the encryption of credentials (username, password) we use SHA2(secure hash algorithm 2). SHA2 is more stronger than SHA1,the latter has been already broken by hackers. The hash values of username and password are calculated and stored in Login Table when the user's account is first time created with the web application. Whenever user wants to login to database his/her identity is checked by comparing username and password entered by the user with the already stored values in the Login table . Whenever comparison returns to be true, the user able to access the database otherwise database access denied. These hash values are calculated at runtime using stored procedure when user wants to login into the database.

If only username and password are used for authentication, and the attacker enters Username = ' OR 1=1 – – and Password = pwd;

The query becomes like this:

```
Select * from Login where Username = ''
OR 1=1 – –' and Password = 'pwd';
```

Fig.4. Query without using hash values

There the user will be to bypass authentication. Whereas using PSIAW approach, the query for authentication will become like this:

```
Select* from Login where
Hashusername='hash_valueof(OR1=1--)'and
Hashpassworddd= 'hash_valueof('pwd');
```

Fig.5. Query using encrypted values

Thus using encrypted values for password and username, the hacker cannot bypass authentication as attacker does not know the hash values of username and password and hence can't access the database of the web application. Thus, web application is secured. The error messages generated by application should not show that any hash values are calculated at the back end and it's getting matched with the entered one. This prevents the attacker from accessing database as he is not aware of any hash

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 4, Issue 9, September 2015

values used and does not know the hash values of username and password as hash values are calculated at runtime. Only two text boxes are provided at the interface for entering username and password, he will not be able to enter hash values from anywhere. Hence, the attacker will not be able to attack database and web application is secured. When user changes password, encrypted value of old password supplied as well as new password is calculated. Encrypted value of old password must matched with the stored encrypted value and new value is stored with the new password in the Login Table.

Every time database is accessed, encrypted values of supplied parameters are calculated and matched with the stored one. Whenever it does not match it simple generates the message, username and password do not match. So the attacker does not get to know about the encrypted values concept.

A. ARCHITECTURE

Architecture for SQL Injection Attack prevention in Web Application (SIAPWA) technique consists of four components: User Login Interface, Encryption algorithm (SHA2), SQL Query Component and Database.

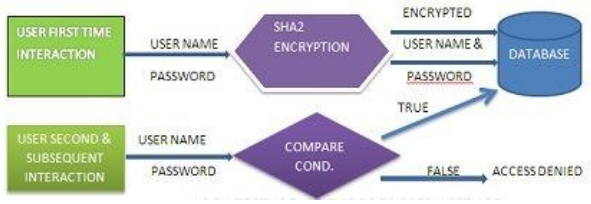

Fig.6. Architecture of the proposed technique

Here, user login interface is just the user entry form containing two columns for username and password. Main component of SIAPW is SQL Query Component. SQL Query component is the component where hash value of username and password is calculated. These values are then compared with username and password, whenever a user try to login. Every time the user enters username and password, their hash values are calculated. The query formed is then sent to database. Subcomponents of SQL Query component are username hash value of username and hash value of password. User Login table is the component where hash value of username and hash value of password are stored.

TABLE I User Login table.

Username	password
Wertr08992nnqsyuyttgghjj 678799	Ttgghjikaannnm556 67787ygtfd
Ddscvgghhcdnloewupkdpj pjbwe	Kihdswdfhahwgtd8 55458ujhdm

V. CONCLUSION AND FUTURE WORK

Many web applications employ a middleware technology designed to request from a relational database in SQL parlance. SQL Injection is a common technique that hackers employ to attack these webs based applications. These attacks reshape the SQL queries, thus altering the behavior of the program for the benefit of the hacker. In our research work, we have presented a technique for protecting authentication against SQL Injection. This technique requires no additional column in a login table. Username and Password are directly stored into the Login table in encrypted form. When the user gets itself registered with a web application, it selects its username and password.

At the same time, hash value of username and password is computed at the coding side and stored in the Login table with Username and Password. When user logs in to the web application, hash value of username and password are matched at the backend and user is allowed to access the data. If SQL Injection attack string is entered for logging into the database, its hash value does not match with the hash values stored in the table and hence attacker cannot access the database.

This technique is tested with different SQL Injection Attack strings. This technique was successful in preventing the login with these strings. Hence, this technique is quite useful in protecting authentication against SQL Injection Attack.

This technique introduce delays log in time by approximately 1ms which is negligible in comparison to the security of database obtained due to this technique.

This technique has a limitation also. This technique can be implemented in the beginning of website development. This technique can't be implemented with the websites already developed as Database has to be changed. Reengineering of website will have to be done to implement this technique on the existing website.

This technique is able to protect only authentication mechanism. Rest of the SQL Injection techniques can't be prevented using this technique. So, there is a need to protect SQL injection attack at other places in web application. Then, this technique will be able to prevent SQL Injection Attack completely.

REFERENCES

[1] Ravindra Kumar, Neha singh, "SQL INJECTIONS – A HAZARD TO WEB APPLICATIONS" International Journal of Advanced Research in Computer Science and Software Engineering, Volume 2, Issue 6, June 2012.

[2] Sruthi Bandhakavi, Prithvi Bisht, P. Madhusudan, CANDID: Preventing SQL Injection Attacks using Dynamic Candidate Evaluation. Proceedings of the 14th ACM conference on Computer and communications security. ACM, Alexandria,Virginia, USA.page:12-24.

[3] Ms. Zeinab Raveshi, Mrs. Sonali R. Idate, Efficient Method to Secure Web applications and Databases against SQL Injection Attacks, International Journal of Advanced Research in Computer Science and Software Engineering, Volume 3, Issue 5, May 2013.

[4] P. Bisht, P. Madhusudan. CANDID: Dynamic Candidate Evaluations for Automatic Prevention of SQL Injection Attacks. ACM Transactions on Information and System Security Volume: 13, Issue: 2, 2010.

[5] S. W. Boyd and A. D. Keromytis. SQLrand: Preventing SQL Injection Attacks. In Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference, pages 292–302, June 2004.

[6] G. T. Buehrer, B. W. Weide, and P. A. G. Sivilotti. Using Parse Tree Validation to Prevent SQL Injection Attacks. In International Workshop on Software Engineering and Middleware (SEM), 2005.

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 4, Issue 9, September 2015

[7] W. R. Cook and S. Rai. Safe Query Objects: Statically Typed Objects as Remotely Executable Queries. In Proceedings of the 27th International Conference on Software Engineering (ICSE 2005), 2005.

[8] Mayank Namdev , Fehreen Hasan, Gaurav Shrivastav, Review of SQL Injection Attack and Proposed Method for Detection and Prevention of SQLIA, International Journal of Advanced Research in Computer Science and Software Engineering, Volume 2, Issue 7, July 2012.

[9] Sonam Panda, Ramani , "Protection of Web Application against Sql Injection Attacks", International Journal of Modern Engineering Research (IJMER) Vol.3, Issue.1, Jan-Feb. 2013 pp-166-168 ISSN: 2249-6645.

[10] Shubham Srivastava, Rajeev Ranjan Kumar Tripathi, "Attacks Due to SQL Injection and Their Prevention Method for Web-Application",(IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 3 (2) , 2012,3615-3618.

[11] Shaukat Ali, Azhar Rauf, Huma Javed," SQLIPA: An Authentication Mechanism Against SQL Injection", European Journal of Scientific Research ISSN 1450-216X Vol.38 No.4 (2009), pp 604-611.

[12] C. Gould, Z. Su, and P. Devanbu. JDBC Checker: A Static Analysis Tool for SQL/JDBC Applications. In Proceedings of the 26th International Conference on Software Engineering (ICSE 04) – Formal Demos, pages 697–698, 2004.

[13] C. Gould, Z. Su, and P. Devanbu. Static Checking of Dynamically Generated Queries in Database Applications. In Proceedings of the 26th International Conference on Software Engineering (ICSE 04), pages 645–654, 2004.

[14] Sunita Gond, Neha Mishra, Defenses To Protect Against SQL Injection Attacks, International Journal of Advanced Research in Computer and Communication Engineering,Vol. 2, Issue 10, October 2013

[15] V. Haldar, D. Chandra, and M. Franz. Dynamic Taint Propagation for Java. In Proceedings 21st Annual Computer Security Applications Conference, Dec. 2005.

[16] W. G. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In Proceedings of the IEEE and ACM International Conference on Automated Software Engineering (ASE 2005), Long Beach, CA, USA, Nov 2005.

[17] W. G. Halfond and A. Orso. Combining Static Analysis and Runtime Monitoring to Counter SQL-Injection Attacks. In Proceedings of the Third International ICSE Workshop on Dynamic Analysis (WODA 2005), pages 22–28, St. Louis, MO, USA, May 2005.

[18] Y. Huang, F. Yu, C. Hang, C. H. Tsai, D. T. Lee, and S. Y. Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In Proceedings of the 12th International World Wide Web Conference (WWW 04), May 2004.

[19] 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.

[20] M. Martin, B. Livshits, and M. S. Lam. Finding Application Errors and Security Flaws Using PQL: A Program Query Language. In Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications(OOPSLA 2005), pages 365–383, 2005.

[21] R. McClure and I. Kr¨uger. SQL DOM: Compile Time Checking of Dynamic SQL Statements. In Proceedings of the 27th International Conference on Software Engineering (ICSE 05), pages 88–96, 2005.

[22] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically Hardening Web Applications Using Precise Tainting Information. In Twentieth IFIP International Information Security Conference (SEC 2005), May 2005.

[23] T. Pietraszek and C. V. Berghe. Defending Against Injection Attacks through Context-Sensitive String Evaluation. In Proceedings of Recent Advances in Intrusion Detection (RAID2005), 2005.

[24] D. Scott and R. Sharp. Abstracting Application-level Web Security. In Proceedings of the 11th International Conference on the World Wide Web (WWW 2002), pages 396–407, 2002.

[25] F. Valeur, 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 and Vulnerability Assessment (DIMVA), Vienna, Austria, July 2005.

[26] 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.