ISSN:0975-9646

Sampada Gadgil/ (IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 4 (2) , 2013, 345- 349

SQL Injection Prevention in Banking

Sampada Gadgil

Department Of IT,
SIES Graduate School Of Technology
Navi Mumbai, India

***Abstract*— SQL injection attack (SQLIA) is a type of attack on web applications that exploits the fact that input provided by web clients is directly included in the dynamically generated SQL statements. SQLIA is one of the foremost threats to web applications. In an SQL injection attack, an attacker might insert a malicious SQL query as input to perform an unauthorized database operation. The attack works when a program builds queries based on strings from the client, and passes them to a database server without handling characters that have special meaning to the server using SQL injection attacks, an attacker can retrieve or modify confidential and sensitive information from the database. The growing use of web-applications for business purposes has given motivation to attackers to explore the possibilities and exploit these types of attacks. In this paper all type of SQL injection attack are discussed. An application is developed for online banking application. This application prevents various types of SQL attacks.**

Keywords— **SQL injection, Web application, WASP, Detection, Prevention**

I. INTRODUCTION

In recent years, most of our daily tasks are depend on database driven web applications because of increasing activity, such as banking, booking and shopping [1]. For performing activities such as ordering the merchandize or paying the bills, information must be trustable to these web applications and their underlying databases but unfortunately there is no any guarantee for confidentiality and integrity of this information. Web applications are often vulnerable to attacks, which can give attackers easily access to the application's underlying database. SQL Injection is a security exploit method in which the attacker aims at penetrating a back-end database to manipulate, steal or modify information in the database. The SQL Injection attack method exploits the Web application by injecting malicious queries, causing the manipulation of data. Almost all SQL databases and programming languages are potentially vulnerable. SQL Injection is subset of an unverified user input vulnerability ("buffer overflows" are a different subset), and the idea is to convince the application to run SQL code that was not intended. Structured Query Language (SQL) is the nearly universal language of databases that allows the storage, manipulation, and retrieval of data. Databases that use SQL include MS SQL Server, MySQL, Oracle, Access and File maker Pro and these databases are equally subject to SQL injection attack.
Not preventing SQL Injection attacks leaves your business at great risk of :

a. Sensitive information can be altered or accessed.
b. It may lead to financial losses
c. In some cases like banking application, attacker can withdraw money.
d. Steal customer information such as credit card numbers

To get a better understanding of SQL injection, we need to have a good understanding of the kinds of communications that take place during a typical session between a user and a web application. The following figure shows [2] the typical communication exchange between all the components in a typical web application system.

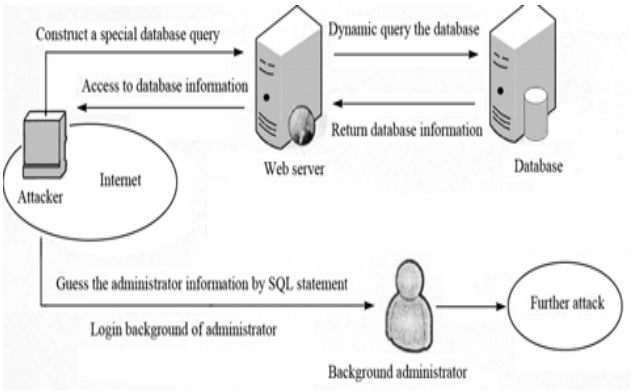

Fig 1: SQL Attack Process

SQL vulnerabilities exist where there are applications need to dynamically constructed SQL statements according to Web client environment. Because the server-side applications use SQL statements in the form of patchwork operation of the database, which allows an attacker to submit the data they want included in SQL statements. For all platforms, based SQL standard database software, SQL language is valid. As long as the client data replaces the data in SQL statements, may be attacked. The result of SQLIA can be disastrous because a successful SQL injection can read sensitive data from database, modify database data and execute operations on the database. The main consequences of these vulnerabilities are attacks on [3]

1) Authorization: Critical data may be altered by a successful SQLIA.
2) Authentication: If there is no any proper control on username or password, it may be possible to login to a system as a normal user without knowing a right username or password.
3) Confidentiality: Usually database contains sensitive data. So attacker can access that data.
4) Integrity: By a successful SQLIA, not only attacker reads sensitive information but also it is possible to change or delete this information.

SQL injection attacks are extremely dangerous types of attacks. These attacks are very much harmful for the web based application. This paper focus on various types of SQL attacks. In this paper, attack types are explained with an example. An application is developed for preventing these attacks. This application is developed for banking system.

II. SQL ATTACK TYPES

There are different methods of attacks that depending on the goal of attacker are performed together or sequentially. For a successful SQLIA the attacker should append a syntactically correct command to the original SQL query.

1. Tautologies: This type of attack injects SQL tokens to the conditional query statement to be evaluated always true. Consider this example [1]:

"SELECT * FROM employee WHERE userid = '112' and password ='aaa' OR '1 '='1

As the tautology statement (1=1) has been added to the query statement so it is always true.

2. Union Queries: These attacks are related to the Tautology SQLIAs. The attacker exploits vulnerable parameters in order to change the result set of a given query. The trick here is that the attacker injects the SQL code in such a way that it returns data from a table different from the one intended by the programmer. The result of Union Query injection attacks will be a new dataset returned by the database, containing the union of the first (developer-intended) and the second (attacker-intended) query.

Suppose for our examples [1], that the query executed from the server is the following:

SELECT Name, Phone FROM Users WHERE Id=$id By injecting the following Id value:

$id= I UNION ALL SELECT credit Card Number, 1 FROM Credit CarT able

We will have the following query:

SELECT Name, Phone FROM Users WHERE Id= 1 UNION ALL SELECT creditCardNumber, 1 FROM Credit CarTable, which will join the result of the original query with all the credit card users.

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 [4]. 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.

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.

4. *Inference*: By this type of attack, intruders change the behaviour of a database or application [1]. There are two well-known attack techniques that are based on inference: blind injection and timing attacks.

Blind injection - is a well-known type of SQLIA where the attacker simply injects an SQL statement and observes the application's behaviour: if the page continues to function normally, then this would imply that the injected statement resulted to true, otherwise the attacker will see a non descriptive error and a page that might differ from the normal one.

Timing attacks - are similar to blind injection attacks, but here the attacker formulates the SQL statements in the form of if-then command and uses SQL commands like WAITFOR to cause delays in the database statements execution along one of the branches of the if-then conditional. [5]

SELECT account FROM users WHERE login='realUser' and ASCII(SUBSTRING((SELECT TOP 1 name from sysobjects), 1, 1))

< X WAITFOR 5 -- 'AND pass=' ' AND pin=0, where the attacker asks if the ASCII value of the first character in the first name in the table is less than a number (X in our example) and he will get 5 seconds delay in database execution of the statement in case that condition evaluates to true.

5. *Illegal/Logically Incorrect Queries*: This type of SQLIAs is used to trigger syntax errors (which would be used to identify injectable parameters), type conversion errors (to deduce the data types of certain columns or extract data from them) or logical errors (which often reveal names of the tables and columns that caused the error), in order for the attacker to gather information about the type and structure of the back-end database of a given Web-application.

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

In the example above [6], we have assumed that the application is using Microsoft SQL Server and the metadata table is called sysobjects. The above-query will try extract the first user table 'xtype=u' and to convert the table name into an integer.

This will result in a type conversion error and MS SQL Server will say: \Microsoft OLE DB Provider for SQL Server (0x80040E07) Error converting varchar value CreditCards to a column of data type int.".This message will be valuable to the attacker, since now he will know the type of the database is Microsoft SQL Server,and secondly, he will now know the name of the first user defined table in the database (CreditCards).

6. *Stored Procedure:* SQLIAs 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 [6]. Therefore, once an attacker determines which backend database is in use, SQLIAs can be crafted to execute stored procedures provided by that specific database, including procedures that interact with the operating system.

CREATE PROCEDURE DBO.isAuthenticated @userName varchar2, @pass varchar2, @pin int AS EXEC("SELECT accounts FROM users WHERE login= ' " +@userName+ " 'and pass=' "+@password+" 'and pin="+@pin);

GO

For authorized/unauthorized user the stored procedure returns true/false. As an SQLIA, intruder input " , ;
SHUTDOWN; - -" for usemame or password. Then the stored procedure generates the following query:
SELECT accounts FROM users WHERE login='doe' AND pass = ' '; SHUTDOWN; -- AND pin=
The first original query is executed and consequently the second query which is illegitimate is executed and causes database shut down.
7.*Alternate Encodings*: This attack type is used in conjunction with other attacks. In other words, 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.
In this attack, the following text is injected into the *login* field:
"legalUser'; exec(0x73687574646f776e) - - ". The resulting query generated by the application is:
SELECT accounts FROM users WHERE login='legalUser'; exec(char(0x73687574646f776e)) -- AND pass='' AND pin=
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, it would result in the execution, by the database, of the SHUTDOWN command.

III APPLICATION Development

Here online banking application is developed to prevent various types of SQL attacks. Banking application contains sensitive data like account information, user id or password etc.
This application consists of two modules:
1. *Normal Process*: In this module, attacker can make SQL attack and he can access sensitive information. He can enter invalid password using SQL keyword or operator. He can modify the account or withdraw the money.
2. *Prevent Process*: In this module WASP prevention technique is used to avoid SQL attacks. So attacker cannot access the sensitive information. Even if the unauthorized user tries to access account data is not available to him. Both the modules contain sub module to perform different types of banking operations.

A Deployment Requirement

Operating System used is Windows XP Professional. Front End is Microsoft Visual Studio .Net 2005. Language to develop this application used is Visual C # .Net. Back End is SQL Server 2000. C# is the programming language that most directly reflects the underlying Common Language Infrastructure (CLI). Most of its intrinsic types correspond to value-types implemented by the CLI framework. However, the language specification does not state the code generation requirements of the compiler: that is, it does not state that a C# compiler must target a Common Language Runtime, or generate Common Intermediate Language (CIL), or generate any other specific format. Theoretically, a C# compiler could generate machine code like traditional compilers of C++ or FORTRAN. However, in practice, all existing compiler implementations target CIL.
SQL Server 2000 is much more integrated with Windows NT Server than any of its predecessors. Databases are now stored directly in Windows NT Server files .SQL Server is being stretched towards both the high and low end. SQL Server 2000 creates a database using a set of operating system files, with a separate file used for each database. Multiple databases can no longer share the same file. There are several important benefits to this simplification. Files can now grow and shrink, and space management is greatly simplified. All data and objects in the database, such as tables, stored procedures, triggers, and views, are stored only within the operating system files. When a database is created, all the files that comprise the database are zeroed out (filled with zeros) to overwrite any existing data left on the disk by previously deleted files. This improves the performance of day-to-day operations.

IV IMPLEMENTATION TECHNIQUE

The implementation makes use of WASP [7] tool for online banking application. WASP stands for Web Application SQL Injection Preventer tool. WASP tool consists of two approaches. These are positive tainting and syntax aware evaluation.
1) Positive tainting [7]: The effectiveness of this approach, in that it helps address problems caused by incompleteness in the identification of relevant data to be marked. In the case of negative tainting, incompleteness leads to trusting data that should not be trusted and, ultimately, to false negatives. Our basic approach, as explained in the following sections, automatically marks as trusted all hard-coded strings in the code and then ensures that all SQL keywords and operators are built using trusted data. In some cases, this basic approach is not enough because developers can also use external query fragments—partial SQL commands that come from external input sources—to build queries. Because these string fragments are not hard coded in the application, they would not be part of the initial set of trusted data identified by our approach and the approach would generate false positives when the string fragments are used in a query. To account for these cases, our technique provides developers with a mechanism for specifying sources of external data that should be trusted. The data sources can be of various types such as files, network connections, and server variables. Our approach uses this information to mark data that comes from these additional sources as trusted. In a typical scenario, we expect developers to specify most of the trusted sources before testing and deployment. However, some of these sources might be overlooked until after a false positive is reported, in which case, developers would add the omitted items to the list of trusted sources. In this process, the set of trusted data sources monotonically grows and eventually converges to a complete set that produces no false positives. It is important to note that false positives that occur after deployment would be due to the use of external data sources that have never been used during in-house testing.

In other words, false positives are likely to occur only for totally untested parts of applications.
Therefore, even when developers fail to completely identify additional sources of trusted data beforehand, we expect these sources to be identified during normal testing and the set of trusted data to quickly converge to the complete set. It is also worth noting that none of the subjects that we collected and examined so far required us to specify additional trusted data sources. All of these subjects used only hard-coded strings to build query. Strings Incompleteness may thus leave the application vulnerable to attacks and can be very difficult to detect, even after attacks actually occur, because they may go completely unnoticed. With positive tainting, incompleteness may lead to false positives, but it would never result in an SQLIA escaping detection. We track taint information at the character level rather than at the string level. We do this because, for building SQL queries, strings are constantly broken into substrings, manipulated, and combined.
2)Syntax aware evaluation[7]: The key feature of syntax aware evaluation is that it considers the context in which trusted and untrusted data is used to make sure that all parts of a query other than string or numeric literals (for example, SQL keywords and operators) consist only of trusted characters. As long as untrusted data is confined to literals, we are guaranteed that no SQLIA can be performed. The WASP technique performs syntax-aware evaluation of a query string immediately before the string is sent to the database to be executed. To evaluate the query string, the technique first uses a SQL parser to break the string into a sequence of tokens that correspond to SQL keywords, operators, and literals. The technique then iterates through the tokens and checks whether tokens (that is, substrings) other than literals contain only trusted data. If all such tokens pass this check, the query is considered safe and is allowed to execute.

V EXPERIMENTAL RESULTS

In this application, SQL injection affects the data in the normal process module. It is shown in the figure 2. Here SQL operator is used for attack purpose. Attacker can get sensitive data by applying SQL keywords or operators which is shown in figure 3.

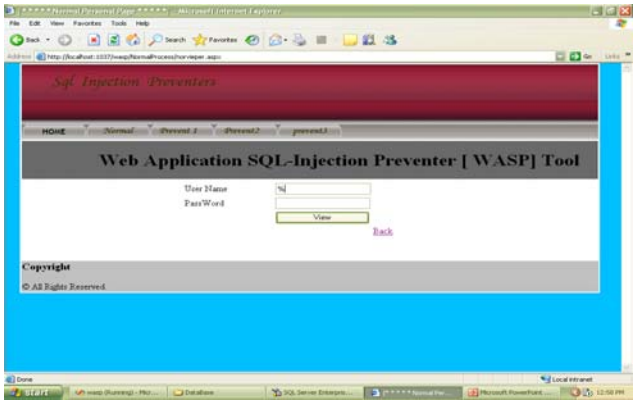

Fig 2: An example of SQL attack

Figure 2 shows example of SQL attack. In this attack, an unauthorized user is accessing the information. He is using SQL operator for this purpose and he is getting the details.

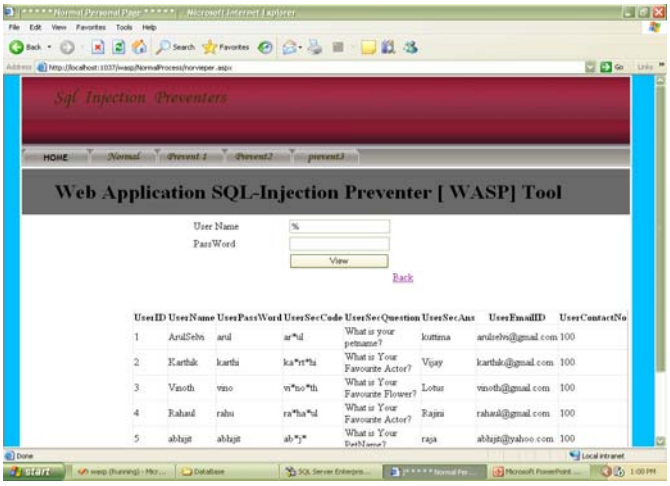

Fig 3: Result of SQL attack

Figure 4 shows one more attack on web application. This is an example of tautological attack. In the user name field, attacker will insert the name and SQL symbols. But when the attacker clicks the submit button, access to the data is not given to the attacker.

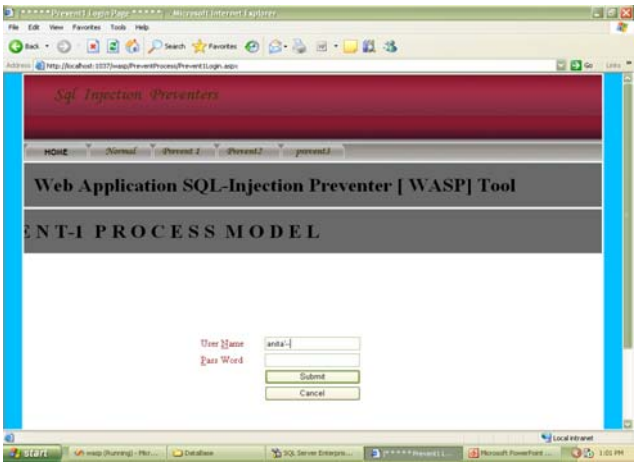

Fig 4: An example of SQL attack

This attack is prevented, which is shown in the following figure.

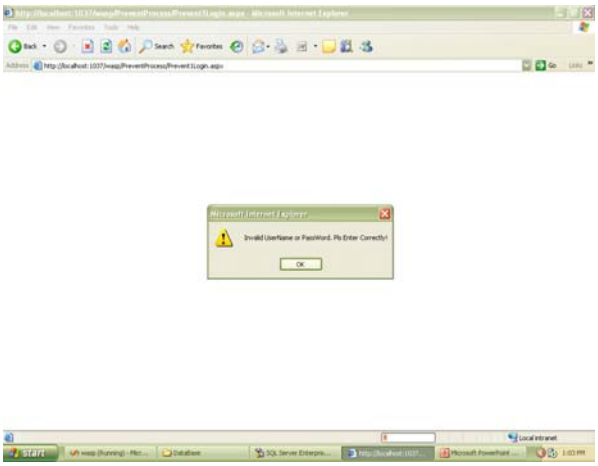

Fig 5: Prevention of SQL attack

VI CONCLUSION AND FUTURE WORK

SQL injection attacks are serious for web application. These attacks can access, alter or destroy the sensitive data. In this paper, concept of SQL attack is explained. Different types of attacks are explained with example.

The application is developed for online banking. For this WASP tools is used. Two modules are developed. One of the modules shows how attacker can access the data. The other one shows prevention of these attacks. It is found that the attacks can be prevented. Future scope is to actually deploy this application on web.

REFERENCES

[1] Atefeh Tajpour, Maslin Masrom, Mohammad Zaman Heydari, Suhaimi Ibrahim," *SQL Injection Detection and Prevention Tools Assessment*",2010 IEEE

[2] Yi Yan, Su Zhengyuan, Dai Zucheng,"*The Database Protection System Against SQL Attacks*", 2011 IEEE

[3] Atefeh Tajpour, Suhaimi Ibrahim, Maslin Masrom," *SQL Injection Detection and Prevention Techniques*", International Journal of Advancements in Computing Technology Volume 3, Number 7, August 2011

[4] William G.J. Halfond, Jeremy Viegas, and Alessandro Orso, "*A Classification of SQL Injection Attacks and Countermeasures*",2006 IEEE

[5] Fatbardh Veseli, "SQL Injection Attacks - Detection and Prevention Techniques"

[6] William G.J. Halfond, Jeremy Viegas, and Alessandro Orso, 'A Classification of SQL Injection Attacks and Countermeasures,2006 IEEE

[7] William G.J. Halfond, Alessandro Orso, "*WASP: Protecting Web Applications Using Positive Tainting and Syntax –Aware Evaluation*", IEEE Transaction on Software Engineering, VOL 34, NO 1, January/February 2008