IJARCCE

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

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 4, Issue 11, November 2015

Types of SQL Injection attacks

Vineet Nayak[1], Nupur Kalra[2], Ankit Gera[3]

Nitte Mahalinga Adhyantaya Memorial Institute of Technology, Nitte, India [1, 2, 3]

Abstract: This SQL injection is a software vulnerability that occurs when data entered by users is sent to the sql interpreter as a part of SQL query. Attackers provide specially crafted input data to the SQL interpreter and trick the interpreter to execute unintended commands. Attackers utilize this vulnerability by providing specially crafted input data to the SQL interpreter in such a manner that the interpreter is not able to distinguish between the intended commands and the attacker's specially crafted data. The interpreter is tricked into executing unintended commands. A SQL injection attack exploits security vulnerabilities at the database layer. By exploiting the SQL injection flaw, attackers can read, modify or delete sensitive data.

Keywords: Attacker, database, query, injection.

I. INTRODUCTION

SQL Injection is one of the web attack methods used by intruders to steal data from different organizations. Hence, it is the most common application layer attack method used in today's world. It is the type of attack that takes advantage of frivolous coding of your web applications that allows intruder to inject SQL commands into say a login form that allows them to gain access to the data which is present within the database.

SQL Injection is the hacking technique. An attempt is made to pass sql commands through a web application for execution by the backend database. If not checked properly, a sql injection attacks can come into existence by web applications that allow intruders to view data from the database and/or even remove it.

In essence, SQL Injection comes into picture because the areas available for user input allow SQL statements to pass through the database and query it directly.

SQL injection attacks are successful based on two important factors: the nature and size of your business and the age, modification on your applications, efficiency and count of your technical staff

II. TYPES OF SQL ATTACKS

In this section, we present and discuss the different kinds of SQL Injection Attacks. 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.

Tautologies

Attack Intent: Bypassing authentication; identifying injectable parameters; extracting data.

Description: 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 most common usages are to bypass authentication pages and extract data. In this type of injection, an attacker exploits an injectable 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 injectable/ vulnerable parameters, but also the coding constructs that evaluate the query results. (Halfond, Viegas, & Alessandro, 2006)

Example 1: Bypassing login script.

Query: SELECT name from authors where username = '$_POST[username]'ANDpassword='$_POST[password]' ;

This query take input from the system user; suppose the user enters:

Username: a' OR '1=1'

Password: a' OR '1=1'

Constructed query: SELECT name from authors where username = 'a' OR '1=1' AND password='a' OR '1=1'

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 to return. Because the condition, the query evaluates to true for each row and returns all of them. This would cause this user to be authenticated as the user whose data is in the first row in the returned result set.

Solution:

```
$username = $_POST[username];
$username = mysqli_real_escape_string ($username);
 mysql_query (SELECT first_name, last_name from authors where username = '$username');
```

Illegal/Logically Incorrect Queries

Attack Intent: Identifying injectable parameters; performing database finger printing; Extracting data.

Description: This attack lets the attacker gather important information about the type and structure of the back-end database of an 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; 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 injectable 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.

Example 2: Cause a type conversion error that can reveal relevant data.
Password: AND 'pin: "convert (int, (select top 1 name from sysobjects where xtype='u'))
Query: SELECT name from authors where username = '' AND password='' AND 'pin = convert (int,(select top 1 name from sysobjects where xtype='u'))
The 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 default error would be "Microsoft OLE DB Provider for SQL Server (0x80040E07) Error converting nvarchar value 'CreditCards' to a column of data type int."
Two useful pieces of information in this message aids an attacker. First, the attacker can see that the database is an SQL Server database. 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.

Union Query
Attack Intent: Bypassing Authentication; extracting data.
Description: 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 than 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 database returns a dataset that is the union of the results of the original first query and the results of the injected second query. One example usage of this multiple attacks is where the attacker uses the logically incorrect query attack to data about a table's structure then use the union query to get data from this table.

Example 3: Referring to example 2, an attacker could inject the text
Username: ' UNION SELECT cardNo from CreditCards where acctNo=10032 - -"
Query: SELECT name from authors where username = '' UNION SELECT cardNo from CreditCards where acctNo=10032 -- AND password=''
Note: It is common technique to force the SQL parser to ignore the rest of the query written by the developer with -- which is the comment sign in SQL.

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. The database takes the results of these two queries, unions them, and returns them to the application.

Piggy Backed Queries
Attack Intent: Extracting data; Adding or modifying data; Performing DOS; executing remote commands.
Description: In this attack, 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 which are all executed. 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.

Example 4: The attacker inputs:
Password: "'; drop table users - -"
Query: SELECT name from authors where username = '' AND password='' drop table users -- AND pin=123
After completing the first query, the database would recognize the query delimiter (";") and execute the injected second query. Dropping the users table 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.
Solution: Configure the database to block executing multiple statements within a single string.

Stored Procedures
Attack Intent: Performing privilege escalation; performing DOS; Executing remote commands.
Description: SQL Injection Attacks of this type try to execute stored procedures present in the database. Most 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. Additionally, because stored procedures are often written in special scripting languages, they can contain other types of vulnerabilities, such as buffer overflows; these vulnerabilities allow attackers to run arbitrary code on the server or escalate their privileges. Here is a stored procedure that checks credentials:

```
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
```

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

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 4, Issue 11, November 2015

Example 5: 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, 6 and 7 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 SQL Injection Attack, the attacker simply enters:
Password: ' ; SHUTDOWN; --
Query: SELECT name from authors where username = 'Jay' AND password=' '; SHUTDOWN; --
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.

Inference
Attack Intent: Identifying injectable parameters; Extracting data; Determining database schema.
Description: 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 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. In this situation, the attacker injects commands into the application and then observes how the application responds. From careful observation, 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:
Blind Injection: Information is 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. 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 pause the execution for a known amount of time (e.g. the WAITFOR keyword). By measuring the response time of the database, the attacker can infer which branch was taken in his injection and therefore the answer to the injected question.

Example 6: Identifying injectable parameters using blind injection. Consider two possible injections into the login field.

- "legalUser' and 1=0 - -"
- "legalUser' and 1=1 - -"

Query 1: SELECT name from authors where username = 'legalUser' and 1=0 -- ' AND password=' ' AND pin=0;
Query 2: SELECT name from authors where username = 'legalUser' and 1=1 -- ' AND password=' ' AND pin=0;
Scenario 1: 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.
Scenario 2: 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. 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.

Example 7: Using Timing based inference attack to extract a table name from the database.
Username: ''legalusr' and ASCII(SUBSTRING((select top 1 name from sysobjects),1,1)) > X WAITFOR 5 --''.
Query:
SELECT name from authors where username = 'legalUser' ASCII(SUBSTRING((select top 1 name from sysobjects),1,1)) > X WAITFOR 5 -- 'AND password=' ' AND pin=0;
Here, the SUBSTRING function extracts the first character of the first table's name. Using a binary search strategy, the attacker can 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.

Alternate Encodings
Attack Intent: Evading detection.
Description: In this attack, the injected text is modified so as to avoid detection by defensive coding practices and also many automated prevention techniques. 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. 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 defense, attackers have employed alternate methods of encoding their attack strings (e.g., using hexadecimal, ASCII, and Unicode character encoding). Common scanning and detection techniques do not try to evaluate all specially encoded strings, thus allowing these attacks to go undetected. An effective code-based defense against alternate encodings is difficult to implement in practice because it requires developers to consider of all of the possible encodings that could affect a given query string as it passes through the different application layers.

DOI 10.17148/IJARCCE.2015.41126

Therefore, attackers have been very successful in using alternate encodings to conceal their attack strings.

Example 8: Every type of attack could be represented using an alternate encoding; here we simply provide an example of how mystic an alternatively-encoded attack could appear.
Username: "legalUser'; exec(0x73687574646f776e) - - "
Query:
SELECT name from authors where username ='legalUser'; exec(0x73687574646f776e) - - AND password=' ';
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.
Query: SELECT name from authors where username = 'legalUser'; exec(SHUTDOWN) - - AND password=' ';

III. CONCLUSION

In this paper, we have presented a survey of current techniques of SQL injection as well as a solution methodology for preventing the attacks. To perform this evaluation, we first identified the various types of SQL Injection attacks .We also studied the different mechanisms through which SQL Injection Attacks can be introduced into an application and identified the techniques that are able to handle the mechanisms. Many of the techniques have problems handling attacks that take advantage of poorly coded stored procedures and SQL queries cannot handle attacks. This difference could be explained by the fact that focused techniques try to incorporate defensive best practices into their attack prevention mechanisms. Version of this template is V2. Most of the formatting instructions in this document have been compiled by Causal Productions from the IEEE LaTeX style files. Causal Productions offers both A4 templates and US Letter templates for LaTeX and Microsoft Word. The LaTeX templates depend on the officialIEEEtran.cls and IEEEtran.bst files, whereas the Microsoft Word templates are self-contained.

ACKNOWLEDGMENT

We would like to thank our teachers and friends for continuous support and encouragement. The library staff for providing us with the required material for the topic.

REFERENCES

[1] Wei, K., Muthuprasanna, M., & Suraj Kothari. (2006, April 18). Preventing SQL injection attacks in stored procedures. Software Engineering IEEE Conference. Retrieved November 2, 2007, from http://ieeexplore.ieee.org
[2] Thomas, Stephen, Williams, & Laurie. (2007, May 20). Using Automated Fix Generation to Secure SQL Statements. Software Engineering for Secure Systems IEEE CNF. Retrieved November 6, 2007, from http://ieeexplore.ieee.org
[3] Merlo, Ettore, Letarte, Dominic, Antoniol & Giuliano. (2007 March 21). Automated Protection of PHP Applications Against SQL-injection Attacks. Software Maintenance and Reengineering, 11th European Conference IEEE CNF. Retrieved November 9, 2007, from http://ieeexplore.ieee.org.
[4] Wassermann Gary, Zhendong Su. (2007, June). Sound and precise analysis of web applications for injection vulnerabilities. ACM SIGPLAN conference on Programming language design and implementation PLDI, 42 (6). Retrieved November 7, 2007, from http://portal.acm.org
[5] Friedl's Steve Unixwiz.net Tech Tips. (2007). SQL Injection Attacks by Example. Retrieved November 1, 2007, from http://www.unixwiz.net/techtips/sql-injection.html
[6] Massachusetts Institute of Technology. Web Application S ecurity MIT Security Camp. Retrieved November 1, 2007, from http://web.mit.edu/netsecurity/Camp/2003/clambert-slides.pdf

DOI 10.17148/IJARCCE.2015.41126