**BUSINESS IMPACT ANALYSIS**

The SQL injection attack consists of the insertion or “injection” of a SQL query via the input data from the client to the application. A successful SQL injection exploit can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file present on the DBMS file system and in some cases issue commands to the operating system. SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input in order to affect the execution of predefined SQL commands.

**Threat Modelling**

* SQL injection attacks allow attackers to spoof identity, tamper with existing data, cause repudiation issues such as voiding transactions or changing balances, allow the complete disclosure of all data on the system, destroy the data or make it otherwise unavailable, and become administrators of the database server.
* SQL Injection is very common with PHP and ASP applications due to the prevalence of older functional interfaces. Due to the nature of programmatic interfaces available, J2EE and ASP.NET applications are less likely to have easily exploited SQL injections.
* The severity of SQL Injection attacks is limited by the attacker’s skill and imagination, and to a lesser extent, defense in depth countermeasures, such as low privilege connections to the database server and so on. In general, consider SQL Injection a high impact severity.

**Conduct a thorough analysis of the potential business impact of SQL injection vulnerability**

SQL injection attack occurs when untrusted data enters a program and is used to dynamically construct a SQL query. The consequences of such attacks can affect confidentiality, authentication, authorization, and integrity.

* Confidentiality: SQL injection vulnerabilities can lead to the loss of confidentiality as attackers can access sensitive data stored in SQL databases. This can result in the exposure of valuable or confidential information, leading to various risks such as fraud or identity theft.
* Authentication: Poorly constructed SQL commands used for checking user credentials can be exploited to gain unauthorized access to a system. Attackers can potentially connect to a system as another user without knowing their password, compromising the authentication process.
* Authorization: SQL injection vulnerabilities can allow attackers to manipulate authorization information stored in a SQL database. By exploiting these vulnerabilities, attackers can modify or change authorization settings, potentially granting themselves unauthorized privileges.
* Integrity: In addition to accessing sensitive information, SQL injection attacks can also modify or delete data in databases. Attackers can manipulate SQL queries to make unauthorized changes to the data, compromising the integrity of the system.

**Risk Factors:**

The platform affected can be:

Language: SQL

Platform: Any (requires interaction with a SQL database)

SQL injection attacks commonly affect database-driven websites and software packages. These vulnerabilities are easily detectable and exploitable, making any site or software with even a minimal user base susceptible to such attacks.

**Examples**:

**Example 1** demonstrates a SQL injection attack where an attacker provides malicious input, resulting in an SQL query with incorrect syntax. This can lead to unintended consequences and potential security breaches.

In SQL: select id, firstname, lastname from authors

If one provided: Firstname: evil'ex and Lastname: Newman

the query string becomes:

select id, firstname, lastname from authors where firstname = 'evil'ex' and lastname ='newman'

which the database attempts to run as:

Incorrect syntax near il' as the database tried to execute evil.

A safe version of the above SQL statement could be coded in Java as:

String firstname = req.getParameter("firstname");

String lastname = req.getParameter("lastname");

// FIXME: do your own validation to detect attacks

String query = "SELECT id, firstname, lastname FROM authors WHERE firstname = ? and lastname = ?";

PreparedStatement pstmt = connection.prepareStatement( query );

pstmt.setString( 1, firstname );

pstmt.setString( 2, lastname );

try

{

ResultSet results = pstmt.execute( );

}

**Example 2** showcases a dynamic SQL query constructed in C# code. Due to the concatenation of user input, the query becomes vulnerable to SQL injection. An attacker can manipulate the query to bypass security checks and access unauthorized data.

...

string userName = ctx.getAuthenticatedUserName();

string query = "SELECT \* FROM items WHERE owner = '"

+ userName + "' AND itemname = '"

+ ItemName.Text + "'";

sda = new SqlDataAdapter(query, conn);

DataTable dt = new DataTable();

sda.Fill(dt);

...

The query that this code intends to execute follows:

SELECT \* FROM items

WHERE owner =

AND itemname = ;

However, because the query is constructed dynamically by concatenating a constant base query string and a user input string, the query only behaves correctly if itemName does not contain a single-quote character. If an attacker with the user name wiley enters the string "name' OR 'a'='a" for itemName, then the query becomes the following:

SELECT \* FROM items

WHERE owner = 'wiley'

AND itemname = 'name' OR 'a'='a';

The addition of the OR 'a'='a' condition causes the where clause to always evaluate to true, so the query becomes logically equivalent to the much simpler query:

SELECT \* FROM items;

This simplification of the query allows the attacker to bypass the requirement that the query only return items owned by the authenticated user; the query now returns all entries stored in the items table, regardless of their specified owner.

**Example 3** highlights the effects of a malicious value passed as input. The attack string can execute multiple SQL statements, allowing the attacker to delete data from the database or execute arbitrary commands. The use of comment characters can further manipulate the query to achieve the desired outcome.

If an attacker with the user name hacker enters the string "name'); DELETE FROM items; --" for itemName, then the query becomes the following two queries:

SELECT \* FROM items

WHERE owner = 'hacker'

AND itemname = 'name';

DELETE FROM items;

--'

Many database servers, including Microsoft® SQL Server 2000, allow multiple SQL statements separated by semicolons to be executed at once. While this attack string results in an error in Oracle and other database servers that do not allow the batch-execution of statements separated by semicolons, in databases that do allow batch execution, this type of attack allows the attacker to execute arbitrary commands against the database.

Notice the trailing pair of hyphens (--), which specifies to most database servers that the remainder of the statement is to be treated as a comment and not executed. In this case the comment character serves to remove the trailing single-quote left over from the modified query. In a database where comments are not allowed to be used in this way, the general attack could still be made effective using a trick similar to the one shown in Example 1. If an attacker enters the string "name'); DELETE FROM items; SELECT \\* FROM items WHERE 'a'='a", the following three valid statements will be created:

SELECT \* FROM items

WHERE owner = 'hacker'

AND itemname = 'name';

DELETE FROM items;

SELECT \* FROM items WHERE 'a'='a';