SQL Injection

Source:

https://owasp.org/www-community/attacks/SQL_Injection

Overview

A SQL injection attack consists of 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 Modeling

- 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, or 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.

Related Security Activities

How to Avoid SQL Injection Vulnerabilities

See the OWASP SQL Injection Prevention Cheat Sheet. See the OWASP Query Parameterization Cheat Sheet.

How to Review Code for SQL Injection Vulnerabilities

See the OWASP Code Review Guide article on how to Review Code for SQL Injection vulnerabilities.

How to Test for SQL Injection Vulnerabilities

See the OWASP Testing Guide for information on testing for SQL Injection vulnerabilities.

How to Bypass Web Application Firewalls with SQLi

See the OWASP Article on using SQL Injection to bypass a WAF

Description

SQL injection attack occurs when:

1. An unintended data enters a program from an untrusted source.
2. The data is used to dynamically construct a SQL query

The main consequences are:

- **Confidentiality**: Since SQL databases generally hold sensitive data, loss of confidentiality is a frequent problem with SQL Injection vulnerabilities.
- **Authentication**: If poor SQL commands are used to check user names and passwords, it may be possible to connect to a system as another user with no previous knowledge of the password.
- **Authorization**: If authorization information is held in a SQL database, it may be possible to change this information through the successful exploitation of a SQL Injection vulnerability.

- **Integrity**: Just as it may be possible to read sensitive information, it is also possible to make changes or even delete this information with a SQL Injection attack.

Risk Factors

The platform affected can be:

- Language: SQL
- Platform: Any (requires interaction with a SQL database)

SQL Injection has become a common issue with database-driven web sites. The flaw is easily detected, and easily exploited, and as such, any site or software package with even a minimal user base is likely to be subject to an attempted attack of this kind.

Essentially, the attack is accomplished by placing a meta character into data input to then place SQL commands in the control plane, which did not exist there before. This flaw depends on the fact that SQL makes no real distinction between the control and data planes.

Examples

Example 1

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

The following C# code dynamically constructs and executes a SQL query that searches for items matching a specified name. The query restricts the items displayed to those where owner matches the user name of the currently-authenticated user.

```
...
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

This example examines the effects of a different malicious value passed to the query constructed and executed in Example 1. 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';
```

One traditional approach to preventing SQL injection attacks is to handle them as an input validation problem and either accept only characters from an allow list of safe values or identify and escape a deny list of potentially malicious values. An allow list can be a very effective means of enforcing strict input validation rules, but parameterized SQL statements require less maintenance and can offer more guarantees with respect to security. As is almost always the case, deny listing is riddled with loopholes that make it ineffective at preventing SQL injection attacks. For example, attackers can:

- Target fields that are not quoted
- Find ways to bypass the need for certain escaped meta-characters
- Use stored procedures to hide the injected meta-characters

Manually escaping characters in input to SQL queries can help, but it will not make your application secure from SQL injection attacks.

Another solution commonly proposed for dealing with SQL injection attacks is to use stored procedures. Although stored procedures prevent

some types of SQL injection attacks, they fail to protect against many others. For example, the following PL/SQL procedure is vulnerable to the same SQL injection attack shown in the first example.

```
procedure get_item (
    itm_cv IN OUT ItmCurTyp,
    usr in varchar2,
    itm in varchar2)
is
    open itm_cv for ' SELECT * FROM items WHERE ' ||
            'owner = '''|| usr ||
            ' AND itemname = ''' || itm || '''';
end get_item;
```

Stored procedures typically help prevent SQL injection attacks by limiting the types of statements that can be passed to their parameters. However, there are many ways around the limitations and many interesting statements that can still be passed to stored procedures. Again, stored procedures can prevent some exploits, but they will not make your application secure against SQL injection attacks.