Idaho Technology Authority (ITA)

ENTERPRISE GUIDELINES G500 – SECURITY PROCEDURES

Category: G591B – SQL Injection Attacks: Information and Avoidance

CONTENTS:

I. DEFINITIONS

A. SQL Server – System that runs a database accessed with Structured Query Language (SQL).
B. SQL Injection Attack – An attack in which a malicious user attempts to insert database commands in place of expected data during a SQL query.

II. RATIONALE

The purpose of this guideline is twofold: to provide background information regarding SQL Injection Attacks so agency staff understand related risks, and to provide techniques to ensure systems are not vulnerable to SQL Injection Attacks.

SQL Injection Attacks take advantage of vulnerabilities in applications which allow bad actors to execute queries against a target database. These queries can expose confidential data, change data for other customers, or even delete entire data tables.

Attempts to exploit this particular vulnerability are very common, and there have been state systems with this vulnerability in the past.

Because of the prevalence of this vulnerability, the extreme risk if an exploit occurs, and the frequency of attacks throughout the public Internet, it is critical that agencies are aware of risks and mitigation strategies for SQL Injection Attacks.

III. GUIDELINE

This guideline is part of the G590 series and it addresses SQL Injection Attacks. Implementing this guideline will better secure all state-used database servers in accordance with *ITA Enterprise Standard S3230 – Server Security Requirements*.

IV. PROCEDURE REFERENCE

The following pages will address specific procedures to mitigate risks of SQL Injection Attacks:

A. Stored Procedures

1. **Details**: Stored procedures not requiring parameters can be susceptible to SQL Injection if they allow unfiltered input from the user or dynamically generated query statements.

2. **Description**: Implement the use of the CallableStatement (Java) or SqlCommand (.NET) stored procedure interfaces in applications. Confine database action statements (QUERY, DROP, UPDATE, etc.) within stored procedures whenever possible.

3. **Solution:** Using stored procedures with parameterization enables the Developer to clearly specify data types as well as the final communication statement generated to the database. If dynamically generated queries cannot be avoided within the stored procedure, include parameter validation for the input and carefully consider the other mitigation procedures that follow.

4. **References**:
    a. Dynamic SQL and SQL injection http://blogs.msdn.com/b/raulga/archive/2007/01/04/dynamic-sql-sql-injection.aspx
    b. OWASP SQL Injection Prevention Cheat Sheet https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet

B. Data Access Architecture

1. **Details:** Review all Data Access components, Service Agents and applications to determine the best way to map application entities to Data Structures.

2. **Description**: All aspects of the data access layer must meet the requirements of the application, perform efficiently and securely, and be easy to maintain. Data Access components abstract the logic required to access the underlying data stores. Service Agents implement data access components that isolate the varying requirements for calling services from the application and provide basic mapping between the format of the data exposed by the service and the format the application requires.

   Applications can help prevent SQL Injection attacks by preventing Meta characters from being passed to the data structure and by not displaying Server error messages.

3. **Solution:** Review and implement Data Layer Guidelines for Data Access components and Service Agents. Update the current Software Development Life Cycle to include the filtering of Meta characters and detailed error messages for applications.

4. **References**:

   a. Microsoft Application Architectural Guide, Ch.8: Data Layer Guidelines
      http://msdn.microsoft.com/en-us/library/ee658127.aspx
   b. Analyzing SQL Meta Characters and Preventing SQL Injection Attacks Using Meta Filter
      http://www.ipcsit.com/vol6/33-E080.pdf
   c. Creating Custom ASP Error Messages
      http://support.microsoft.com/default.aspx?scid=kb;EN-US;q224070

C. Prepared Statements

1. **Details:** Database queries stored and maintained within an application may be modified with user input at runtime. A malicious user may attempt to insert database commands in place of expected data.

2. **Description**: A Prepared Statement is a special database query maintained within the application. The execution of the prepared statement is supported by the database connection provider or the ODBC driver located on the client or web server.

   Preparing a statement provides additional security **only** when parameter markers are used. The application variables bound to parameter markers are type checked, not interpreted literally and not executed as database commands.

3. **Solution:** Developers should be trained to write all database queries maintained by the application and incorporating user inputs as prepared statements. All user inputs should correspond to parameter markers.

   Binding the parameter to an application variable is the mechanism that grants protection from SQL injection. Developers must ensure that all USER inputs are sent as parameters and are not dynamically added to the prepared statement at run-time.

4. **References**:

   a. Preparing SQL Statements http://msdn.microsoft.com/en-us/library/ms175528.aspx
   b. OWASP Open Web Application Security Project SQL Injection Prevention Cheatsheet (*Prepared Statements*) https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet#Defense_Option_1:_Prepared_Statements_.28Parameterized_Queries.29
   c. Using Prepared Statements http://download.oracle.com/javase/tutorial/jdbc/basics/prepared.html

D. Escaping Characters

1. **Details:** Dynamically built SQL commands, if incorporating unexamined user strings, can include commands which will compromise the security of your data.

2. **Description:** Data strings received from users via GET or POST input, if accepted "as is," can be manipulated to include a command to provide the nefarious user with a data dump, to drop the database table, or any other SQL command.

   One POST example is manipulating a login form, with username and password fields, to deliver all user passwords by entering *anything* in the username field and *bar' OR ''='* in the password field.

   A GET example is changing URL from *http://domain.com/file.cfm?pid=101* to *http://domain.com/file.cfm?pid=101;DELETE+products* (where *products* is the name of a table).

3. **Solution:** Use functions to vet or evaluate any data received from users or which can be "touched" by users. Never trust user-supplied data strings.

4. **References:**

   a. Wikipedia page on SQL injection (see *Technical Implementations* section)
   http://en.wikipedia.org/wiki/SQL_injection
   b. OWASP Open Web Application Security Project
   SQL Injection Prevention Cheatsheet (*Escaping User Supplied Input*)
   https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet#Defense_Option_3:_Escaping_All_User_Supplied_Input
   c. Ben Forta's ColdFusion Blog: SQL Injection Attacks, Easy To Prevent, But Apparently Still Ignored
   d. Filtering SQL Injection from Classic ASP
   http://blogs.iis.net/nazim/archive/2008/04/28/filtering-sql-injection-from-classic-asp.aspx
   e. SQL Injection Attacks by Example
   http://unixwiz.net/techtips/sql-injection.html
   f. Detection SQL Injection and Cross Site Scripting Attacks (includes RegEx)
   http://www.symantec.com/connect/articles/detection-sql-injection-and-cross-site-scripting-attacks
   g. PHP online Manual page on SQL Injection
   http://php.net/manual/en/security.database.sql-injection.php

E. Least Privilege

1. **Details:** Provide the lowest level of privilege possible while still allowing customers and applications necessary access.

2. **Description**: Different types of data querying require different levels of access. If data is accessed using exclusively stored procedures, access to base tables can be restricted. Customers and applications can be provided access only to execute the stored procedures, and are thereby limited in what they can do.

   If permissions are limited in this way, it not only reduces the risk of SQL injection attacks, it also eliminates the risk of internal staff connecting to production tables with tools such as MS Excel and inadvertently causing harm.

3. **Solution:** If possible, do not provide customers or applications any direct permissions to the base tables. If all data access is performed using stored procedures, permission can be granted only to execute the stored procedures necessary. Moreover, always provide the least amount of permissions which enable customers and applications to do their work.

4. **References**:

   a. OWASP SQL Injection Prevention Cheat Sheet https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet

V. REFERENCE DOCUMENTS

In addition to this guideline, the following documents apply:

A. *ITA Enterprise Standard S3230 – Server Security Requirements*

B. *ITA Enterprise Guideline G590A – Server Operating System*

C. *ITA Enterprise Guideline G590B – Public-Facing SQL Server Setup*

VI. CONTACT INFORMATION

For more information, contact the ITA Staff at (208) 605-4064.

VII. REVIEW CYCLE

Twelve (12) months

VIII. TIMELINE

Date Established: November 22, 2011
Last Reviewed:
Last Revised:

IX. REVISION HISTORY

07/01/13 – Changed “ITRMC” to “ITA”.

11/22/11 New document.