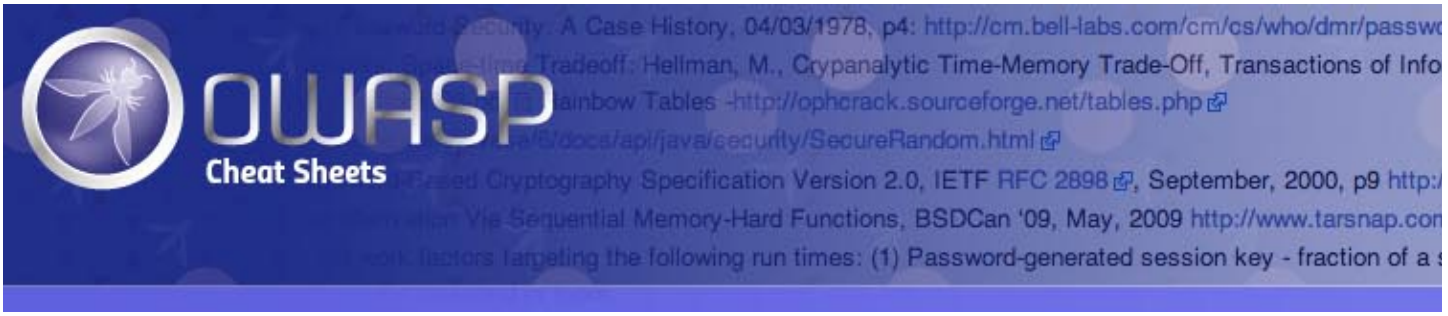
SQL Injection Prevention Cheat Sheet

From OWASP



Last revision (mm/dd/yy): **06/7/2014**

Introduction

This article is focused on providing clear, simple, actionable guidance for preventing SQL Injection flaws in your applications. SQL Injection attacks are unfortunately very common, and this is due to two factors:

1. the significant prevalence of SQL Injection vulnerabilities, and
2. the attractiveness of the target (i.e., the database typically contains all the interesting/critical data for your application).

It's somewhat shameful that there are so many successful SQL Injection attacks occurring, because it is EXTREMELY simple to avoid SQL Injection vulnerabilities in your code.

SQL Injection flaws are introduced when software developers create dynamic database queries that include user supplied input. To avoid SQL injection flaws is simple. Developers need to either: a) stop writing dynamic queries; and/or b) prevent user supplied input which contains malicious SQL from affecting the logic of the executed query.

This article provides a set of simple techniques for preventing SQL Injection vulnerabilities by avoiding these two problems. These techniques can be used with practically any kind of programming language with any type of database. There are other types of databases, like XML databases, which can have similar problems (e.g., XPath and XQuery injection) and these techniques can be used to protect them as well.

Primary Defenses:

- **Option #1: Use of Prepared Statements (Parameterized Queries)**
- **Option #2: Use of Stored Procedures**
- **Option #3: Escaping all User Supplied Input**

Additional Defenses:

- **Also Enforce: Least Privilege**
- **Also Perform: White List Input Validation**

Unsafe Example

SQL injection flaws typically look like this:

The following (Java) example is UNSAFE, and would allow an attacker to inject code into the query that would be executed by the database. The unvalidated "customerName" parameter that is simply appended to the query allows an attacker to inject any SQL code they want. Unfortunately, this method for accessing databases is all too common.

```
String query = "SELECT account_balance FROM user_data WHERE user_name = "
  + request.getParameter("customerName");

try {
       Statement statement = connection.createStatement( … );
       ResultSet results = statement.executeQuery( query );
}
```

Primary Defenses

Defense Option 1: Prepared Statements (Parameterized Queries)

The use of prepared statements (aka parameterized queries) is how all developers should first be taught how to write database queries. They are simple to write, and easier to understand than dynamic queries. Parameterized queries force the developer to first define all the SQL code, and then pass in each parameter to the query later. This coding style allows the database to distinguish between code and data, regardless of what user input is supplied.

Prepared statements ensure that an attacker is not able to change the intent of a query, even if SQL commands are inserted by an attacker. In the safe example below, if an attacker were to enter the userID of tom' or '1'='1, the parameterized query would not be vulnerable and would instead look for a username which literally matched the entire string tom' or '1'='1.

Language specific recommendations:

- Java EE – use PreparedStatement() with bind variables
- .NET – use parameterized queries like SqlCommand() or OleDbCommand() with bind variables
- PHP – use PDO with strongly typed parameterized queries (using bindParam())
- Hibernate - use createQuery() with bind variables (called named parameters in Hibernate)
- SQLite - use sqlite3_prepare() to create a statement object (http://www.sqlite.org/c3ref/stmt.html)

In rare circumstances, prepared statements can harm performance. When confronted with this situation, it is best to either a) strongly validate all data or b) escape all user supplied input using an escaping routine specific to your database vendor as described below, rather than using a prepared statement. Another option which might solve your

performance issue is to use a stored procedure instead.

Safe Java Prepared Statement Example

The following code example uses a PreparedStatement, Java's implementation of a parameterized query, to execute the same database query.

```
String custname = request.getParameter("customerName"); // This should REALLY be validated too
// perform input validation to detect attacks
String query = "SELECT account_balance FROM user_data WHERE user_name = ? ";

PreparedStatement pstmt = connection.prepareStatement( query );
pstmt.setString( 1, custname);
ResultSet results = pstmt.executeQuery( );
```

Safe C# .NET Prepared Statement Example

With .NET, it's even more straightforward. The creation and execution of the query doesn't change. All you have to do is simply pass the parameters to the query using the Parameters.Add() call as shown here.

```
String query =
        "SELECT account_balance FROM user_data WHERE user_name = ?";
try {
       OleDbCommand command = new OleDbCommand(query, connection);
       command.Parameters.Add(new OleDbParameter("customerName", CustomerName Name.Text));
       OleDbDataReader reader = command.ExecuteReader();
       // …
} catch (OleDbException se) {
       // error handling
}
```

We have shown examples in Java and .NET but practically all other languages, including Cold Fusion, and Classic ASP, support parameterized query interfaces. Even SQL abstraction layers, like the Hibernate Query Language (http://www.hibernate.org/) (HQL) have the same type of injection problems (which we call HQL Injection (http://cwe.mitre.org/data/definitions/564.html)). HQL supports parameterized queries as well, so we can avoid this problem:

Hibernate Query Language (HQL) Prepared Statement (Named Parameters) Examples

```
First is an unsafe HQL Statement

Query unsafeHQLQuery = session.createQuery("from Inventory where productID='"+userSuppliedParameter+"'");

Here is a safe version of the same query using named parameters

Query safeHQLQuery = session.createQuery("from Inventory where productID=:productid");
safeHQLQuery.setParameter("productid", userSuppliedParameter);
```

For examples of parameterized queries in other languages, including Ruby, PHP, Cold Fusion, and Perl, see the Query Parameterization Cheat Sheet.

Developers tend to like the Prepared Statement approach because all the SQL code stays within the application. This makes your application relatively database independent. However, other options allow you to store all the SQL code in the database itself, which has both security and non-security advantages. That approach, called Stored Procedures, is described next.

Defense Option 2: Stored Procedures

Stored procedures have the same effect as the use of prepared statements when implemented safely*. They require the developer to define the SQL code first, and then pass in the parameters after. The difference between prepared statements and stored procedures is that the SQL code for a stored procedure is defined and stored in the database itself, and then called from the application. Both of these techniques have the same effectiveness in preventing SQL injection so your organization should choose which approach makes the most sense for you.

*Note: 'Implemented safely' means the stored procedure does not include any unsafe dynamic SQL generation. Developers do not usually generate dynamic SQL inside stored procedures. However, it can be done, but should be avoided. If it can't be avoided, the stored procedure must use input validation or proper escaping as described in this article to make sure that all user supplied input to the stored procedure can't be used to inject SQL code into the dynamically generated query. Auditors should always look for uses of sp_execute, execute or exec within SQL Server stored procedures. Similar audit guidelines are necessary for similar functions for other vendors.

There are also several cases where stored procedures can increase risk. For example, on MS SQL server, you have 3 main default roles: db_datareader, db_datawriter and db_owner. Before stored procedures came into use, DBA's would give db_datareader or db_datawriter rights to the webservice's user, depending on the requirements. However, stored procedures require execute rights, a role that is not available by default. Some setups where the user management has been centralized, but is limited to those 3 roles, cause all web apps to run under db_owner rights so stored procedures can work. Naturally, that means that if a server is breached the attacker has full rights to the database, where previously they might only have had read-access. More on this topic here. http://www.sqldbatips.com/showarticle.asp?ID=8 (http://www.sqldbatips.com/showarticle.asp?ID=8)

Safe Java Stored Procedure Example

The following code example uses a CallableStatement, Java's implementation of the stored procedure interface, to execute the same database query. The "sp_getAccountBalance" stored procedure would have to be predefined in the database and implement the same functionality as the query defined above.

```
String custname = request.getParameter("customerName"); // This should REALLY be validated
try {
       CallableStatement cs = connection.prepareCall("{call sp_getAccountBalance(?)}");
       cs.setString(1, custname);
       ResultSet results = cs.executeQuery();
       // … result set handling
} catch (SQLException se) {
       // … logging and error handling
}
```

Safe VB .NET Stored Procedure Example

The following code example uses a SqlCommand, .NET's implementation of the stored procedure interface, to execute the same database query. The "sp_getAccountBalance" stored procedure would have to be predefined in the database and implement the same functionality as the query defined above.

```
Try
       Dim command As SqlCommand = new SqlCommand("sp_getAccountBalance", connection)
       command.CommandType = CommandType.StoredProcedure
       command.Parameters.Add(new SqlParameter("@CustomerName", CustomerName.Text))
       Dim reader As SqlDataReader = command.ExecuteReader()
       ' …
Catch se As SqlException
       ' error handling
End Try
```

We have shown examples in Java and .NET but practically all other languages, including Cold Fusion, and Classic ASP, support the ability to invoke stored procedures.

For organizations that already make significant or even exclusive use of stored procedures, it is far less likely that they have SQL injection flaws in the first place. However, you still need to be careful with stored procedures because it is possible, although relatively rare, to **create a dynamic query inside of a stored procedure that is subject to SQL injection**. If dynamic queries in your stored procedures can't be avoided, you can use bind variables inside your stored procedures, just like in a prepared statement. Alternatively, you can validate or properly escape all user supplied input to the dynamic query, before you construct it. For examples of the use of bind variables inside of a stored procedure, see the Stored Procedure Examples in the OWASP Query Parameterization Cheat Sheet.

There are also some additional security and non-security benefits of stored procedures that are worth considering. One security benefit is that if you make exclusive use of stored procedures for your database, you can restrict all database user accounts to only have access to the stored procedures. This means that database accounts do not have

permission to submit dynamic queries to the database, giving you far greater confidence that you do not have any SQL injection vulnerabilities in the applications that access that database. Some non-security benefits include performance benefits (in most situations), and having all the SQL code in one location, potentially simplifying maintenance of the code and keeping the SQL code out of the application developers' hands, leaving it for the database developers to develop and maintain.

Defense Option 3: Escaping All User Supplied Input

This third technique is to escape user input before putting it in a query. If you are concerned that rewriting your dynamic queries as prepared statements or stored procedures might break your application or adversely affect performance, then this might be the best approach for you. However, this methodology is frail compared to using parameterized queries and we cannot guarantee it will prevent all SQL Injection in all situations. This technique should only be used, with caution, to retrofit legacy code in a cost effective way. Applications built from scratch, or applications requiring low risk tolerance should be built or re-written using parameterized queries.

This technique works like this. Each DBMS supports one or more character escaping schemes specific to certain kinds of queries. If you then escape all user supplied input using the proper escaping scheme for the database you are using, the DBMS will not confuse that input with SQL code written by the developer, thus avoiding any possible SQL injection vulnerabilities.

- Full details on ESAPI are available here on OWASP.
- The javadoc for ESAPI is available here at its Google Code repository (http://owasp-esapi-java.googlecode.com/svn/trunk_doc/index.html).
- You can also directly browse the source at Google (http://code.google.com/p/owasp-esapi-java/source/browse/#svn/trunk/src/main/java/org/owasp/esapi), which is frequently helpful if the javadoc isn't perfectly clear.

To find the javadoc specifically for the database encoders, click on the ‘Codec’ class on the left hand side. There are lots of Codecs implemented. The two Database specific codecs are OracleCodec, and MySQLCodec.

Just click on their names in the ‘All Known Implementing Classes:’ at the top of the Interface Codec page.

At this time, ESAPI currently has database encoders for:

- Oracle
- MySQL (Both ANSI and native modes are supported)

Database encoders for:

- SQL Server
- PostgreSQL

Are forthcoming. If your database encoder is missing, please let us know.

Database Specific Escaping Details

If you want to build your own escaping routines, here are the escaping details for each of the databases that we have developed ESAPI Encoders for:

Oracle Escaping

This information is based on the Oracle Escape character information found here: http://www.orafaq.com/wiki/SQL_FAQ#How_does_one_escape_special_characters_when_writing_SQL_queries.3F

Escaping Dynamic Queries

To use an ESAPI database codec is pretty simple. An Oracle example looks something like:

```
ESAPI.encoder().encodeForSQL( new OracleCodec(), queryparam );
```

So, if you had an existing Dynamic query being generated in your code that was going to Oracle that looked like this:

```
String query = "SELECT user_id FROM user_data WHERE user_name = '" + req.getParameter("userID")
+ "' and user_password = '" + req.getParameter("pwd") +"'";
try {
    Statement statement = connection.createStatement( … );
    ResultSet results = statement.executeQuery( query );
}
```

You would rewrite the first line to look like this:

```
Codec ORACLE_CODEC = new OracleCodec();
String query = "SELECT user_id FROM user_data WHERE user_name = '" +
  ESAPI.encoder().encodeForSQL( ORACLE_CODEC, req.getParameter("userID")) + "' and user_password = '"
  + ESAPI.encoder().encodeForSQL( ORACLE_CODEC, req.getParameter("pwd")) +"'";
```

And it would now be safe from SQL injection, regardless of the input supplied.

For maximum code readability, you could also construct your own OracleEncoder.

```
Encoder oe = new OracleEncoder();
String query = "SELECT user_id FROM user_data WHERE user_name = '"
  + oe.encode( req.getParameter("userID")) + "' and user_password = '"
  + oe.encode( req.getParameter("pwd")) +"'";
```

With this type of solution, all your developers would have to do is wrap each user supplied parameter being passed in into an **ESAPI.encoder().encodeForOracle( )** call or whatever you named it, and you would be done.

Turn off character replacement

Use SET DEFINE OFF or SET SCAN OFF to ensure that automatic character replacement is turned off. If this character replacement is turned on, the & character will be treated like a SQLPlus variable prefix that could allow an attacker to retrieve private data.

See http://download.oracle.com/docs/cd/B19306_01/server.102/b14357/ch12040.htm#i2698854 and http://stackoverflow.com/questions/152837/how-to-insert-a-string-which-contains-an for more information

Escaping Wildcard characters in Like Clauses

The LIKE keyword allows for text scanning searches. In Oracle, the underscore '_' character matches only one character, while the ampersand '%' is used to match zero or more occurrences of any characters. These characters must be escaped in LIKE clause criteria. For example:

```
SELECT name FROM emp
WHERE id LIKE '%/_%' ESCAPE '/';
```

```
SELECT name FROM emp
WHERE id LIKE '%\%%' ESCAPE '\';
```

Oracle 10g escaping

An alternative for Oracle 10g and later is to place { and } around the string to escape the entire string. However, you have to be careful that there isn't a } character already in the string. You must search for these and if there is one, then you must replace it with }}. Otherwise that character will end the escaping early, and may introduce a

vulnerability.

MySQL Escaping

MySQL supports two escaping modes:

1. ANSI_QUOTES SQL mode, and a mode with this off, which we call
2. MySQL mode.

ANSI SQL mode: Simply encode all ' (single tick) characters with '' (two single ticks)

MySQL mode, do the following:

```
NUL (0x00) --> \0  [This is a zero, not the letter O]
BS  (0x08) --> \b
TAB (0x09) --> \t
LF  (0x0a) --> \n
CR  (0x0d) --> \r
SUB (0x1a) --> \Z
"   (0x22) --> \"
%   (0x25) --> \%
'   (0x27) --> \'
\   (0x5c) --> \\
_   (0x5f) --> \_
all other non-alphanumeric characters with ASCII values less than 256  --> \c
where 'c' is the original non-alphanumeric character.
```

This information is based on the MySQL Escape character information found here:
http://mirror.yandex.ru/mirrors/ftp.mysql.com/doc/refman/5.0/en/string-syntax.html

SQL Server Escaping

We have not implemented the SQL Server escaping routine yet, but the following has good pointers to articles describing how to prevent SQL injection attacks on SQL server

- http://blogs.msdn.com/raulga/archive/2007/01/04/dynamic-sql-sql-injection.aspx

DB2 Escaping

This information is based on DB2 WebQuery special characters found here: https://www-304.ibm.com/support/docview.wss?uid=nas14488c61e3223e8a78625744f00782983 as well as some information from Oracle's JDBC DB2 driver found here:
http://docs.oracle.com/cd/E12840_01/wls/docs103/jdbc_drivers/sqlescape.html

Information in regards to differences between several DB2 Universal drivers can be found here:
http://publib.boulder.ibm.com/infocenter/db2luw/v8/index.jsp?topic=/com.ibm.db2.udb.doc/ad/rjvjcsqc.htm

Additional Defenses

Beyond adopting one of the three primary defenses, we also recommend adopting all of these additional defenses in order to provide defense in depth. These additional defenses are:

- **Least Privilege**
- **White List Input Validation**

Least Privilege

To minimize the potential damage of a successful SQL injection attack, you should minimize the privileges assigned to every database account in your environment. Do not assign DBA or admin type access rights to your application accounts. We understand that this is easy, and everything just 'works' when you do it this way, but it is very dangerous. Start from the ground up to determine what access rights your application accounts require, rather than trying to figure out what access rights you need to take away. Make sure that accounts that only need read access are only granted read access to the tables they need access to. If an account only needs access to portions of a table, consider creating a view that limits access to that portion of the data and assigning the account access to the view instead, rather than the underlying table. Rarely, if ever, grant create or delete access to database accounts.

If you adopt a policy where you use stored procedures everywhere, and don't allow application accounts to directly execute their own queries, then restrict those accounts to only be able to execute the stored procedures they need. Don't grant them any rights directly to the tables in the database.

SQL injection is not the only threat to your database data. Attackers can simply change the parameter values from one of the legal values they are presented with, to a value that is unauthorized for them, but the application itself might be authorized to access. As such, minimizing the privileges granted to your application will reduce the likelihood of such unauthorized access attempts, even when an attacker is not trying to use SQL injection as part of their exploit.

While you are at it, you should minimize the privileges of the operating system account that the DBMS runs under. Don't run your DBMS as root or system! Most DBMSs run out of the box with a very powerful system account. For example, MySQL runs as system on Windows by default! Change the DBMS's OS account to something more appropriate, with restricted privileges.

White List Input Validation

Input validation can be used to detect unauthorized input before it is passed to the SQL query. For more information please see the Input Validation Cheat Sheet.

Related Articles

SQL Injection Attack Cheat Sheets

The following articles describe how to exploit different kinds of SQL Injection Vulnerabilities on various platforms that this article was created to help you avoid:

- Ferruh Mavituna : "SQL Injection Cheat Sheet" - http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/
- RSnake : "SQL Injection Cheat Sheet-Esp: for filter evasion" - http://ha.ckers.org/sqlinjection/

Description of SQL Injection Vulnerabilities

- OWASP article on SQL Injection Vulnerabilities
- OWASP article on Blind_SQL_Injection Vulnerabilities

How to Avoid SQL Injection Vulnerabilities

- OWASP Developers Guide article on how to Avoid SQL Injection Vulnerabilities
- OWASP article on Preventing SQL Injection in Java
- OWASP Cheat Sheet that provides numerous language specific examples of parameterized queries using both Prepared Statements and Stored Procedures
- The Bobby Tables site (inspired by the XKCD webcomic) has numerous examples in different languages of parameterized Prepared Statements and Stored Procedures (http://bobby-tables.com/)

How to Review Code for SQL Injection Vulnerabilities

- OWASP Code Review Guide article on how to Review Code for SQL Injection Vulnerabilities

How to Test for SQL Injection Vulnerabilities

- OWASP Testing Guide article on how to Test for SQL Injection Vulnerabilities

Authors and Primary Editors

Dave Wichers - dave.wichers[at]owasp.org
Jim Manico - jim[at]owasp.org
Matt Seil - mseil[at]acm.org

Other Cheatsheets

Developer Cheat Sheets (Builder)

- Authentication Cheat Sheet
- Choosing and Using Security Questions Cheat Sheet
- Clickjacking Defense Cheat Sheet
- C-Based Toolchain Hardening Cheat Sheet
- Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet
- Cryptographic Storage Cheat Sheet
- DOM based XSS Prevention Cheat Sheet
- Forgot Password Cheat Sheet
- HTML5 Security Cheat Sheet
- Input Validation Cheat Sheet
- JAAS Cheat Sheet
- Logging Cheat Sheet
- .NET Security Cheat Sheet
- Password Storage Cheat Sheet
- Pinning Cheat Sheet
- Query Parameterization Cheat Sheet
- Ruby on Rails Cheatsheet
- REST Security Cheat Sheet
- Session Management Cheat Sheet
- **SQL Injection Prevention Cheat Sheet**
- Transport Layer Protection Cheat Sheet
- Unvalidated Redirects and Forwards Cheat Sheet
- User Privacy Protection Cheat Sheet
- Web Service Security Cheat Sheet
- XSS (Cross Site Scripting) Prevention Cheat Sheet

Assessment Cheat Sheets (Breaker)

- Attack Surface Analysis Cheat Sheet
- XSS Filter Evasion Cheat Sheet
- REST Assessment Cheat Sheet

Mobile Cheat Sheets

- IOS Developer Cheat Sheet
- Mobile Jailbreaking Cheat Sheet

OpSec Cheat Sheets (Defender)

- Virtual Patching Cheat Sheet

Draft Cheat Sheets

- OWASP Top Ten Cheat Sheet
- Access Control Cheat Sheet
- Application Security Architecture Cheat Sheet
- Business Logic Security Cheat Sheet

- PHP Security Cheat Sheet
- Secure Coding Cheat Sheet
- Secure SDLC Cheat Sheet
- Threat Modeling Cheat Sheet
- Web Application Security Testing Cheat Sheet
- Grails Secure Code Review Cheat Sheet
- IOS Application Security Testing Cheat Sheet
- Key Management Cheat Sheet
- Insecure Direct Object Reference Prevention Cheat Sheet
- Content Security Policy Cheat Sheet