SQL Injection Prevention Cheat Sheet

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).

SQL Injection flaws are introduced when software developers create dynamic database queries constructed with string concatenation which includes user supplied input. To avoid SQL injection flaws is simple. Developers need to either: a) stop writing dynamic queries with string concatenation; 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 (with Parameterized Queries)**
- **Option 2: Use of Properly Constructed Stored Procedures**
- **Option 3: Allow-list Input Validation**
- **Option 4: Escaping All User Supplied Input**

Additional Defenses:

- **Also: Enforcing Least Privilege**
- **Also: Performing Allow-list Input Validation as a Secondary Defense**

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 (with Parameterized Queries)

The use of prepared statements with variable binding (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

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.

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.

```
// This should REALLY be validated too
String custname = request.getParameter("customerName");
// 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 (HQL) have the same type of injection problems (which we call HQL Injection). 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='"+userSup

//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 or this site.

Developers tend to like the Prepared Statement approach because all the SQL code stays within the application. This makes your application relatively database independent.

Defense Option 2: Stored Procedures

Stored procedures are not always safe from SQL injection. However, certain standard stored procedure programming constructs have the same effect as the use of parameterized queries when implemented safely which is the norm for most stored procedure languages.

They require the developer to just build SQL statements with parameters which are automatically parameterized unless the developer does something largely out of the norm. 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.

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.

```
// This should REALLY be validated
String custname = request.getParameter("customerName");
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
```

Defense Option 3: Allow-list Input Validation

Various parts of SQL queries aren't legal locations for the use of bind variables, such as the names of tables or columns, and the sort order indicator (ASC or DESC). In such situations, input validation or query redesign is the most appropriate defense. For the names of tables or columns, ideally those values come from the code, and not from user parameters.

But if user parameter values are used for targeting different table names and column names, then the parameter values should be mapped to the legal/expected table or column names to make sure unvalidated user input doesn't end up in the query. Please note, this is a symptom of poor design and a full rewrite should be considered if time allows.

Here is an example of table name validation.

```
String tableName;
switch(PARAM):
  case "Value1": tableName = "fooTable";
                 break;
  case "Value2": tableName = "barTable";
                 break;
  ...
  default      : throw new InputValidationException("unexpected value provided"
                                                  + " for table name");
```

The `tableName` can then be directly appended to the SQL query since it is now known to be one of the legal and expected values for a table name in this query. Keep in mind that generic table validation functions can lead to data loss as table names are used in queries where they are not expected.

For something simple like a sort order, it would be best if the user supplied input is converted to a boolean, and then that boolean is used to select the safe value to append to the query. This is a very standard need in dynamic query creation.

For example:

```
public String someMethod(boolean sortOrder) {
 String SQLquery = "some SQL ... order by Salary " + (sortOrder ? "ASC" : "DESC");`

  ...
```

Any time user input can be converted to a non-String, like a date, numeric, boolean, enumerated type, etc. before it is appended to a query, or used to select a value to append to the query, this ensures it is safe to do so.

Input validation is also recommended as a secondary defense in ALL cases, even when using bind variables as is discussed later in this article. More techniques on how to implement strong input validation is described in the Input Validation Cheat Sheet.

Defense Option 4: Escaping All User-Supplied Input

This technique should only be used as a last resort, when none of the above are feasible. Input validation is probably a better choice as this methodology is frail compared to other defenses and we cannot guarantee it will prevent all SQL Injection in all situations.

This technique is to escape user input before putting it in a query. It is very database specific in its implementation. It's usually only recommended to retrofit legacy code when implementing input validation isn't cost effective. Applications built from scratch, or applications requiring low risk tolerance should be built or re-written using parameterized queries, stored procedures, or some kind of Object Relational Mapper (ORM) that builds your queries for you.

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.

The OWASP Enterprise Security API (ESAPI) is a free, open source, web application security control library that makes it easier for programmers to write lower-risk applications. The ESAPI libraries are designed to make it easier for programmers to retrofit security into existing applications. The ESAPI libraries also serve as a solid foundation for new development:

- Full details on ESAPI are available here on OWASP.

- The javadoc for ESAPI 2.x (Legacy) is available. This code was migrated to GitHub in November 2014.
- The legacy ESAPI for Java at GitHub helps understand existing use of it when Javadoc seems insufficient.
- An attempt at another ESAPI for Java GitHub has other approaches and no tests or concrete codecs.

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 are forthcoming for:

- SQL Server
- PostgreSQL

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
- SQL Server
- DB2

ORACLE ESCAPING

This information is based on the Oracle Escape character information.

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, you would need only to wrap each user-supplied parameter being passed into an `ESAPI.encoder().encodeForOracle( )` call or whatever you named the call 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 here and here 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.

SQL SERVER ESCAPING

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

DB2 ESCAPING

This information is based on DB2 WebQuery special characters as well as some information from Oracle's JDBC DB2 driver.

Information in regards to differences between several DB2 Universal drivers.

Hex-encoding all input

A somewhat special case of escaping is the process of hex-encode the entire string received from the user (this can be seen as escaping every character). The web application should hex-encode the user input before including it in the SQL statement. The SQL statement should take into account this fact, and accordingly compare the data.

For example, if we have to look up a record matching a sessionID, and the user transmitted the string abc123 as the session ID, the select statement would be:

```
SELECT ... FROM session WHERE hex_encode(sessionID) = '616263313233'
```

`hex_encode` should be replaced by the particular facility for the database being used. The string 606162313233 is the hex encoded version of the string received from the user (it is the sequence of hex values of the ASCII/UTF-8 codes of the user data).

If an attacker were to transmit a string containing a single-quote character followed by their attempt to inject SQL code, the constructed SQL statement will only look like:

```
... WHERE hex_encode ( ... ) = '2720 ... '
```

`27` being the ASCII code (in hex) of the single-quote, which is simply hex-encoded like any other character in the string. The resulting SQL can only contain numeric digits and letters `a` to `f`, and never any special character that could enable an SQL injection.

Escaping SQLi in PHP

Use prepared statements and parameterized queries. These are SQL statements that are sent to and parsed by the database server separately from any parameters. This way it is impossible for an attacker to inject malicious SQL.

You basically have two options to achieve this:

1. Using PDO (for any supported database driver):

```
$stmt = $pdo->prepare('SELECT * FROM employees WHERE name = :name');
$stmt->execute(array('name' => $name));
foreach ($stmt as $row) {
    // do something with $row
}
```

1. Using MySQLi (for MySQL):

```
$stmt = $dbConnection->prepare('SELECT * FROM employees WHERE name = ?');
$stmt->bind_param('s', $name);
$stmt->execute();
```

```
$result = $stmt->get_result();
while ($row = $result->fetch_assoc()) {
    // do something with $row
}
```

PDO is the universal option. If you're connecting to a database other than MySQL, you can refer to a driver-specific second option (e.g. pg_prepare() and pg_execute() for PostgreSQL).

Additional Defenses

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

- **Least Privilege**
- **Allow-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.

Multiple DB Users

The designer of web applications should not only avoid using the same owner/admin account in the web applications to connect to the database. Different DB users could be used for different web applications.

In general, each separate web application that requires access to the database could have a designated database user account that the web-app will use to connect to the DB. That way, the designer of the application can have good granularity in the access control, thus reducing the privileges as much as possible. Each DB user will then have select access to what it needs only, and write-access as needed.

As an example, a login page requires read access to the username and password fields of a table, but no write access of any form (no insert, update, or delete). However, the sign-up page certainly requires insert privilege to that table; this restriction can only be enforced if these web apps use different DB users to connect to the database.

Views

You can use SQL views to further increase the granularity of access by limiting the read access to specific fields of a table or joins of tables. It could potentially have additional benefits: for example, suppose that the system is required (perhaps due to some specific legal requirements) to store the passwords of the users, instead of salted-hashed passwords.

The designer could use views to compensate for this limitation; revoke all access to the table (from all DB users except the owner/admin) and create a view that outputs the hash of the password field and not the field itself. Any SQL injection attack that succeeds in stealing DB information will be restricted to stealing the hash of the passwords (could even be a keyed hash), since no DB user for any of the web applications has access to the table itself.

Allow-list Input Validation

In addition to being a primary defense when nothing else is possible (e.g., when a bind variable isn't legal), input validation can also be a secondary defense used to detect unauthorized input before it is passed to the SQL query. For more information please see the Input Validation Cheat Sheet. Proceed with caution here. Validated data is not necessarily safe to insert into SQL queries via string building.

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:

- SQL Injection Cheat Sheet
- Bypassing WAF's with SQLi - SQL Injection Bypassing WAF

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 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

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