SQL Injection Prevention Cheat Sheet

Introduction

This cheat sheet will help you prevent SQL injection flaws in your applications. It will define what SQL injection is, explain where those flaws occur, and provide four options for defending against SQL injection attacks. SQL Injection attacks are common because:

1. SQL Injection vulnerabilities are very common, and
2. The application's database is a frequent target for attackers because it typically contains interesting/critical data.

What Is a SQL Injection Attack?

Attackers can use SQL injection on an application if it has dynamic database queries that use string concatenation and user supplied input. To avoid SQL injection flaws, developers need to:

1. Stop writing dynamic queries with string concatenation or
2. Prevent malicious SQL input from being included in executed queries.

There are simple techniques for preventing SQL injection vulnerabilities and they can be used with practically any kind of programming language and any type of database. While XML databases can have similar problems (e.g., XPath and XQuery injection), these techniques can be used to protect them as well.

Anatomy of A Typical SQL Injection Vulnerability

A common SQL injection flaw in Java is below. Because its unvalidated "customerName" parameter is simply appended to the query, an attacker can enter SQL code into that query and the application would take the attacker's code and execute it on the database.

```
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

- **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: STRONGLY DISCOURAGED: Escaping All User Supplied Input**

Defense Option 1: Prepared Statements (with Parameterized Queries)

When developers are taught how to write database queries, they should be told to use prepared statements with variable binding (aka parameterized queries). Prepared statements are simple to write and easier to understand than dynamic queries, and parameterized queries force the developer to define all SQL code first and pass in each parameter to the query later.

If database queries use this coding style, the database will always distinguish between code and data, regardless of what user input is supplied. Also, prepared statements ensure that an attacker cannot change the intent of a query, even if SQL commands are inserted by an attacker.

Safe Java Prepared Statement Example

In the safe Java example below, if an attacker were to enter the userID as `tom' or '1'='1`, the parameterized query would look for a username that literally matches the entire string `tom' or '1'='1`. Thus, the database would be protected against injections of malicious SQL code.

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

In .NET, the creation and execution of the query doesn't change. Just pass the parameters to the query using the `Parameters.Add()` call as shown below.

```
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
}
```

While we have shown examples in Java and .NET, practically all other languages (including Cold Fusion and Classic ASP) support parameterized query interfaces. Even SQL abstraction layers, like the Hibernate Query Language (HQL) with the same type of injection problems (called HQL Injection) support parameterized queries as well:

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

```
// This is an unsafe HQL statement
Query unsafeHQLQuery = session.createQuery("from Inventory where productID='"+userSu
// 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);
```

Other Examples of Safe Prepared Statements

If you need examples of prepared queries/parameterized languages, including Ruby, PHP, Cold Fusion, Perl, and Rust, see the Query Parameterization Cheat Sheet or this site.

Generally, developers like prepared statements because all the SQL code stays within the application, which makes applications relatively database independent.

Defense Option 2: Stored Procedures

Though stored procedures are not always safe from SQL injection, developers can use certain standard stored procedure programming constructs. This approach has the same effect as using parameterized queries, as long as the stored procedures are implemented safely (which is the norm for most stored procedure languages).

Safe Approach to Stored Procedures

If stored procedures are needed, the safest approach to using them requires the developer to build SQL statements with parameters that 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, then called from the application. Since prepared statements and safe stored procedures are equally effective in preventing SQL injection, your organization should choose the approach that makes the most sense for you.

When Stored Procedures Can Increase Risk

Occasionally, stored procedures can increase risk when a system is attacked. For example, on MS SQL Server, you have three main default roles: `db_datareader`, `db_datawriter` and `db_owner`. Before stored procedures came into use, DBAs 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 not available by default. In some setups where user management has been centralized, but is limited to those 3 roles, web apps would have to run as `db_owner` so stored procedures could 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 Java's implementation of the stored procedure interface (`CallableStatement`) to execute the same database query. The `sp_getAccountBalance` stored procedure has to be predefined in the database and use the same functionality as the query 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 must be predefined in the database and use 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

If you are faced with parts of SQL queries that can't use bind variables, such as table names, column names, or sort order indicators (ASC or DESC), input validation or query redesign is the most appropriate defense. When table or column names are needed, ideally those values come from the code and not from user parameters.

Sample Of Safe Table Name Validation

WARNING: Using user parameter values to target table or column names is a symptom of poor design and a full rewrite should be considered if time allows. If that is not possible, developers should map the parameter values to the legal/expected table or column names to make sure unvalidated user input doesn't end up in the query.

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

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

Safest Use Of Dynamic SQL Generation (DISCOURAGED)

When we say a stored procedure is "implemented safely," that means it 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.

Sample of Safer Dynamic Query Generation (DISCOURAGED)

For something simple like a sort order, it is 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 discussed earlier in this article. More techniques on how to implement strong input validation is described in the Input Validation Cheat Sheet.

Defense Option 4: STRONGLY DISCOURAGED: Escaping All User-Supplied Input

In this approach, the developer will escape all user input before putting it in a query. It is very database specific in its implementation. This methodology is frail compared to other defenses, and we CANNOT guarantee that this option will prevent all SQL injections in all situations.

If an application is built from scratch or requires low risk tolerance, it 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.

Additional Defenses

Beyond adopting one of the four primary defenses, we also recommend adopting all of these additional defenses 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. 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. DO NOT ASSIGN DBA OR ADMIN TYPE ACCESS 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.

Minimizing Application and OS Privileges

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.

Details Of Least Privilege When Developing

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 of 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.

Least Admin Privileges For Multiple DBs

The designers of web applications should avoid using the same owner/admin account in the web applications to connect to the database. Different DB users should be used for different web applications.

In general, each separate web application that requires access to the database should have a designated database user account that the application 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 only what it needs, 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.

Enhancing Least Privilege with SQL 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 have additional benefits.

For example, if 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. They could 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