How to Use SQL Calls to Secure Your Web Site

How to Secure Your Web Site
Supplementary Volume

March 2010

IT SECURITY CENTER (ISEC)
INFORMATION-TECHNOLOGY PROMOTION AGENCY, JAPAN

This guideline is available for download at:

How to Use SQL Calls to Secure Your Web Site

http://www.ipa.go.jp/security/vuln/websecurity.html

Contents

Preface

Since 2005, security incidents, such as information leak and web site alteration, have become frequent. Among the nine vulnerabilities discussed in the How to Secure Your Web Site, SQL injection is the one that needs a special attention and must be dealt with immediately considering the large impact and serious damage it could cause.

In addition, the use of open source software became active in the last few years in the field of the web application development. Some of those software, however, are being used without enough security verification. Depending on the programming language and type of database engine, software may be exploited by attackers through vulnerability in software-specific specifications or character encoding errors even if software is implemented with standard SQL injection countermeasures.

IPA pointed out the importance of implementing the countermeasures against SQL injection in the How to Secure Your Web Site and has been strongly recommending the use of the bind mechanism as a fundamental solution. Depending on software, however, some cases suggest that the bind mechanism itself may have a vulnerability. In addition, although some books and articles introduce the method of escaping instead of the use of the bind mechanism as a countermeasure against SQL injection, escaping may not be enough to build secure SQL statements depending on the type or configuration of database engines.

This guideline examines the requirements to securely use the bind mechanism and escaping method, and presents some examples of how to use which DBMS products to build and implement secure SQL statements.

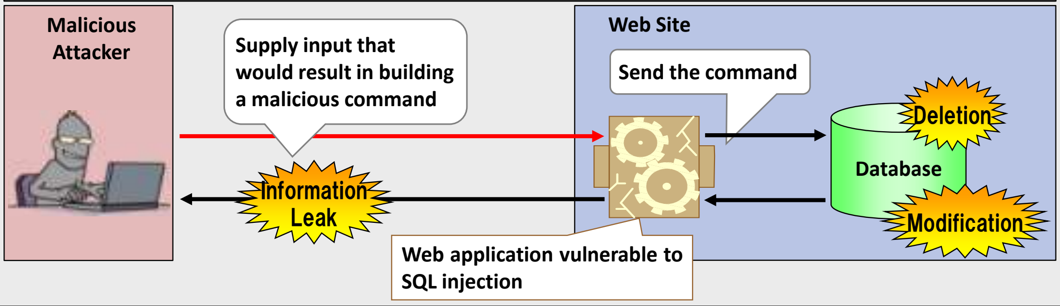
IPA hopes that this guideline will help you secure your web site from SQL injection.

1. Introduction

The version four of the How to Secure Your Web Site introduces the countermeasures against SQL injection as follows.

SQL Injection

SQL injection allows an attacker to manipulate the database with maliciously-crafted requests.


Malicious Attacker
Supply input that would result in building a malicious command
Information Leak
Web Site
Send the command
Deletion
Database
Modification
Web application vulnerable to SQL injection

■ **Fundamental Solutions**

1)-2 **If binding mechanism is not supported, perform escaping for everything that makes up SQL statements**

What does it mean?

This is a secondary solution that should be applied if binding mechanism suggested in 1)-1 is not available for use.

You should perform escaping for everything that makes up SQL statements, from user input and database values to every single variable and arithmetic operation outcomes. You should escape the characters that have a special meaning to SQL statements. For example, replace ‘ (single quote) with ‘ ‘ (two single quotes) and \ (backslash) with \\ (two backslashes) .

You need to escape the SQL special characters appropriate for database engine you use since they differ for each database engine.

How to Secure Your Web Site, 4th Version, p5-7

A fundamental solution for SQL injection is to use the bind mechanism. If it isn’t an option for some reason, escaping can be used instead. As mentioned above, however, the metacharacters that have a special meaning to SQL statements differ depending on database engines and you should escape them appropriate to your system environment. The How to Secure Your Web Site does not give a detailed explanation on how to do it.

This supplementary volume gives that. It explains the necessity of the countermeasures appropriate to database engine and shows the research results of whether the suggested countermeasures work correctly as expected with the DBMS products, as well as the impact the character encoding has on SQL injection. Based on these analyses, how to use what DBMS products to build and implement secure SQL statements is clarified.

In the next chapter, the literals, one of the potential causes of SQL injection, will be explained.

2. Literals and SQL Injection

2.1. Structure of SQL Statement

The structure of an SQL statement is explained using the following example.

```
SELECT a,b,c FROM atable WHERE name='YAMADA' and age>=20
```

SQL

An SQL statement is composed by the elements such as key words, operators, identifiers and literals.

Key Words (Reserved)	SELECT FROM WHERE AND
Operators	= >= ,
Identifiers	a b c atable name age
Literals	'YAMADA' 20

2.2. Literals

For example, to search an employee that has an “employee ID” of “052312”, you can execute the following SQL statement.

```
SELECT name, age FROM employee WHERE employee_id = '052312'
```

SQL

Here, a constant like “052312” in the SQL statement is called literals and if it is a string, it’s especially called string literals. Besides string literals, there are also numeric literals, boolean literals and date and time literals.

【Examples of Numeric Literals】

```
20
-17
0
3.14159
6.0221415E+23
```

SQL

【Examples of String Literals】

```
'Information-technology Promoting Agency'
'052312'
'O''Reilly'
```

SQL

【Examples of Date and Time Literals】

```
DATE '2009-11-04'
TIME '13:59:26'
```

SQL

2.3. Escaping of String Literals

When writing string literals or data and time literals, you bracket the literals in single quotes. It is called "quoting". A problem arises when a single quote appears in a string to be quoted because the program needs to distinguish whether it is a character and part of a literal or the metacharacter that suggests the end of the literal.

For example, if you build a SQL statement like the following to search a person named "O'Reilly", it results in syntax error.

```
SELECT * FROM employee WHERE name = 'O'Reilly'
```

SQL

It is because the program interprets the string literal ends at the second single quote 'O' and the following Reilly' part spills out of the literal. SQL syntax suggests if a single quote appears in a string literal, mark a single quote with another single quote to tell the program that it is a character and part of the string literal. The example above should have been written like the following. This processing is called "escaping" of string literals.

```
SELECT * FROM employee WHERE name = 'O''Reilly'
```

SQL

Note that the single quote is not the only metacharacter that needs to be escaped. What should be escaped differs depending on the type and configuration of database engines. The details are given in the chapter 4.1.

2.4. Numeric Literals

Numeric literals do not need to be quoted. The syntax for integer literals, which are part of numeric literals, is as follows.

Integer	← sign unsigned-integer or unsigned-integer
Sign	← the plus sign (+) or minus sign (-)
Unsigned-Integer	← more than one digit
Digit	← 0 - 9

As described above, an integer literal starts with a sign or digit and ends with the character before the first non-digit character. Note that SQL JIS/ISO standard (JIS X 3005、ISO/IEC 9075) defines that the numeric literal must be followed by a symbol, space or comment.

The following SQL statement is evaluated as JIS/ISO-compliant.

```
SELECT * FROM employee WHERE age >= 25--comment
```
SQL

On the other hand, the next SQL statement violates the JIS/ISO rule.

```
SELECT * FROM employee WHERE age >= 25and age <= 60
```
SQL

There needs to be a space or comment between `25` and `and`. Some database engine implementation (Microsoft SQL Server and PostgreSQL), however, accepts such SQL statement.

Since numeric literals do not need to be quoted, you do not have to do escaping like string literals.

2.5. Causes of SQL Injection

When calling SQL procedures with applications, it is common to use a parameter to substitute for the literal part of an SQL statement. If a literal is not built grammatically correctly when the literal is replaced with the parameter value, part of the parameter value will be cut off from the literal and interpreted as another SQL statement which follows the literal. This is how SQL injection occurs.

2.5.1. SQL Injection against String Literals

The following SQL statement written in Perl shows an example of how SQL injection takes place. The `id` is an SQL identifier and is a string type. The `$id` is a Perl variable and its value is given externally.

```
$q = "SELECT * FROM atable WHERE id='$id'";
```
Perl

Let’s suppose that the following value is given to the `$id`.

```
';DELETE FROM atable--
```
Txt

With the insertion, the resulting SQL statement will become as follows.

```
SELECT * FROM atable WHERE id='';DELETE FROM atable--'
```

SQL

As you can see, a DELETE statement is added after the SELECT statement and all the database contents will be deleted upon execution. The hyphens (--) and the input after them are ignored as comments.

Like the example above, when using a parameter to substitute for the literal part of an SQL statement, attackers could change what the SQL statement is supposed to do by inserting another SQL statement. This vulnerability is called SQL injection.

2.5.2. SQL Injection against Numeric Literals

With numeric literals, caution is required when the application is implemented with a programming language that does not provide parameter types, such as Perl and PHP. The application developers write a application with the premise that only numeric value is inputted for the numeric parameters, but the application will process the input value as string when non-numeric value is entered if the programming language does not provide parameter types.

Let’s see an example. It is the same statement shown in 2.5.1 but here the `id` is a numeric type. `$id` does not need to be quoted since it is a numeric literal

```
$q = "SELECT * FROM atable WHERE id=$id";
```

Perl

Assume that the SQL process above is called and the following value is given to the `$id`.

```
0;DELETE FROM atable
```

Txt

With the insertion, the resulting SQL statement will become as follows.

```
SELECT * FROM atable WHERE id=0;DELETE FROM atable
```

SQL

Once again, a DELETE statement is added after the SELECT statement and all the database contents will be deleted upon execution.

2.5.3. Requirements to Securely Call SQL Procedure

As seen in the examples, it is necessary to replace the literals with the parameter values properly and securely to call SQL processes safely. To do that, make sure the following.

・For string literals, escape the characters that should be escaped

- For numeric literals, make sure that non-numeric value is not inserted as their value

The next chapter will classify the methods to call SQL procedure from applications in three types and show how literals are processed in each method.

3. How to Call SQL Procedure

When calling an SQL procedure with applications, it is common to use parameters to specify the search conditions. If the identifiers represent the employee name or employee ID like the examples in the previous chapter, the application needs to pass a name or employee ID of the person you wants to find to SQL. It is done in two main ways.

- Building an SQL statement by string concatenation
- Building an SQL statement using a placeholder

3.1. Building SQL Statement by String Concatenation

String concatenation is a method to insert a parameter values and build an SQL statement by joining two character strings end to end. The below is an example of the SQL statement written in PHP to search an employee whose name is specified in the CGI parameter `name`. Note that this example program is vulnerable to SQL injection.

```
$name = $_POST['name'];
//...
$sql = "SELECT * FROM employee WHERE name='" . $name . "'";
```

PHP

If the given value for the PHP variable $name is `YAMADA`, the following SQL statement will be built upon insertion.

```
SELECT * FROM employee WHERE name='YAMADA'
```

SQL

To eliminate the SQL injection vulnerability, you need to perform escaping on the value of `$name` when quoting and concatenating `$name`. The metacharacters that need to be escaped differ depending on the type and configuration of database engines.

3.2. Building SQL Statement using Placeholder

Placeholder is a method to insert a parameter value and build an SQL statement by marking a place where a parameter value is later inserted with some symbol, such as `?`, and mechanically replacing it with the actual parameter value later on. The below is an example written in Java.

```java
PreparedStatement prep = conn.prepareStatement("SELECT * FROM employee WHERE name=?");
prep.setString(1, "YAMADA");
```

Java

The symbol of ? here that marks the parameter part in the statement is called a placeholder and replacing it with the actual input value is called "binding". The placeholder is also called the bind variable.

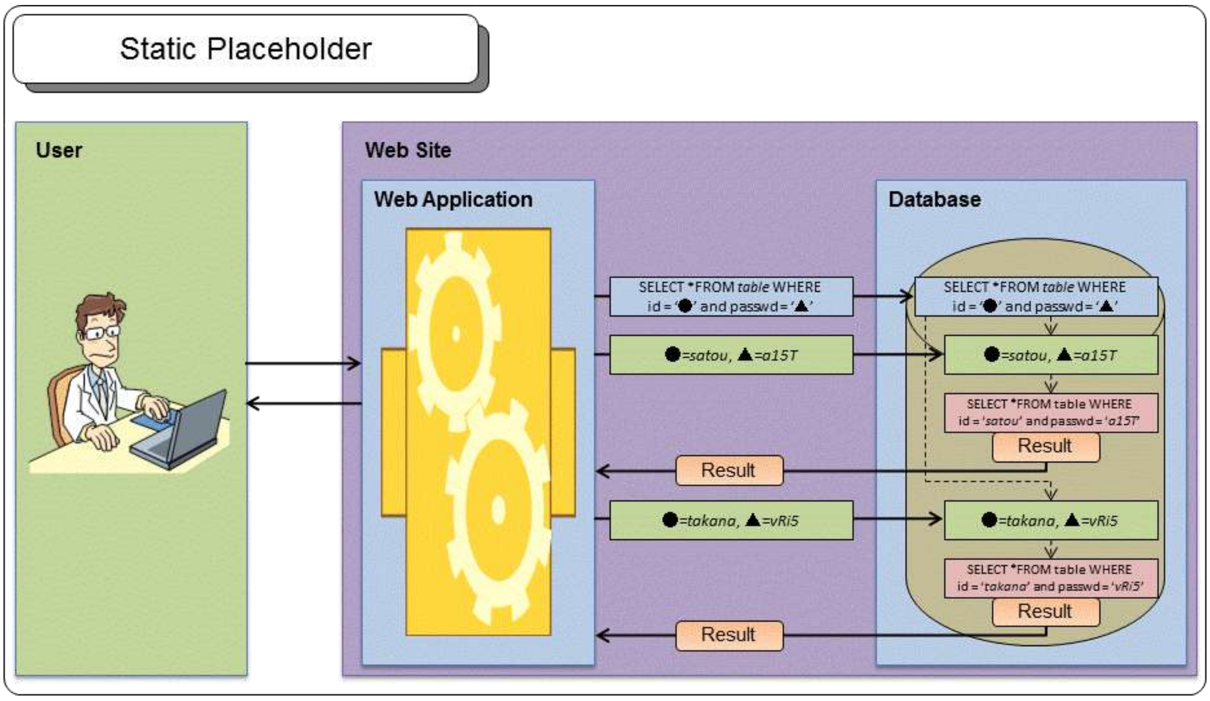
The method of building an SQL statement using a placeholder is classified into two types depending on the timing of when to perform biding.

- Static placeholder
- Dynamic placeholder

3.2.1. Static Placeholder

In the JIS/ISO standard, the static placeholder is defined as prepared statement. It is a method where an SQL statement containing a placeholder is sent to the database engine in advance and prepared for syntax analysis before execution. When executing the SQL statement, the actual parameter value is sent to the database engine and the database engine performs binding.

Applications repeatedly execute the same SQL statements with different values. Therefore, preparing for syntax analysis in advance will improve the execution efficiency. The static placeholder, however, may not be supported by some database engines and libraries.

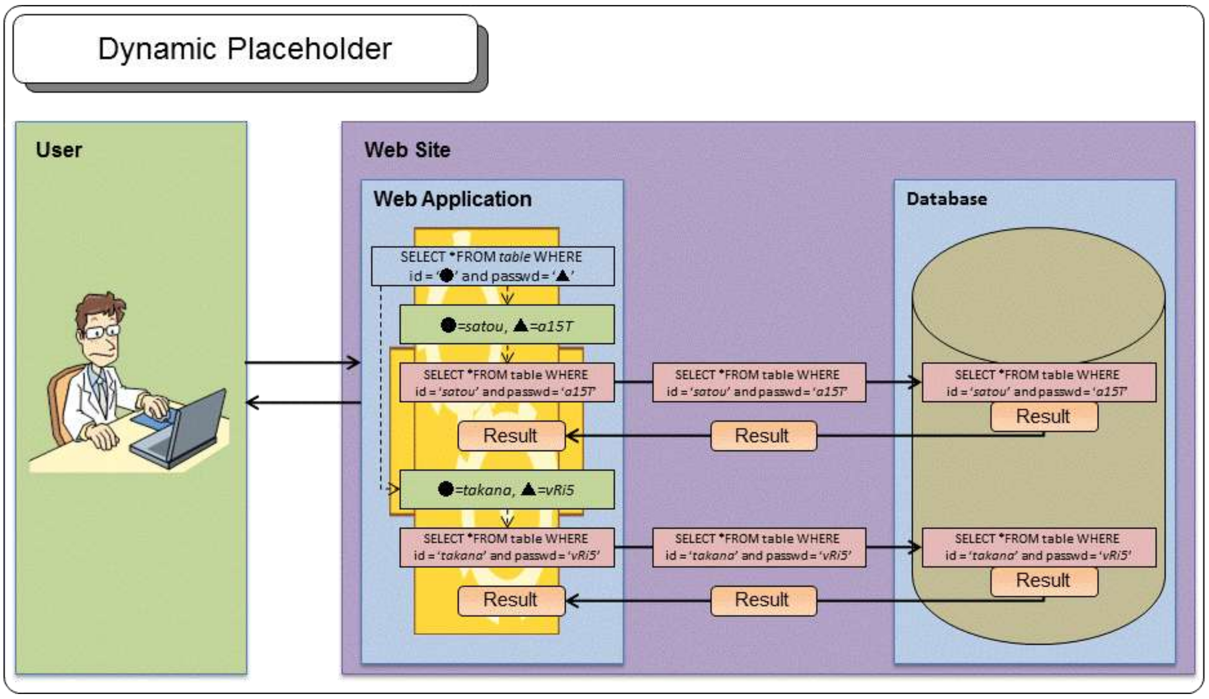

Static Placeholder
User
Web Site
Web Application
Database
SELECT *FROM table WHERE id='●' and passwd='▲'
●=satou, ▲=a15T
SELECT *FROM table WHERE id='satou' and passwd='a15T'
Result
●=takana, ▲=vRi5
SELECT *FROM table WHERE id='takana' and passwd='vRi5'
Result

With the method using the static placeholder, the strings that are to be passed to the placeholder do not need to be quoted because the syntax of the SQL statement is fixed before binding. In turn, escaping on single quotes is unnecessary. Numeric literals are properly bound without intervention as well.

From these reasons, it can say that using the static placeholder is the most secure. With this method, the syntax of the SQL statement is fixed when preparing the statement and does not change later, the parameter value will not spill out of the expected literal. As a result, the method is free of SQL injection vulnerability.

3.2.2. Dynamic Placeholder

The dynamic placeholder is different from prepared statements. Although the dynamic placeholder uses a placeholder, this method performs parameter binding on the application side using libraries instead of on the database engine side.


Dynamic Placeholder
User
Web Site
Web Application
Database
SELECT *FROM table WHERE id = '●' and passwd = '▲'
●=satou, ▲=a15T
SELECT *FROM table WHERE id = 'satou' and passwd = 'a15T'
SELECT *FROM table WHERE id = 'satou' and passwd = 'a15T'
SELECT *FROM table WHERE id = 'satou' and passwd = 'a15T'
Result
Result
Result
●=takana, ▲=vRi5
SELECT *FROM table WHERE id = 'takana' and passwd = 'vRi5'
SELECT *FROM table WHERE id = 'takana' and passwd = 'vRi5'
SELECT *FROM table WHERE id = 'takana' and passwd = 'vRi5'
Result
Result
Result

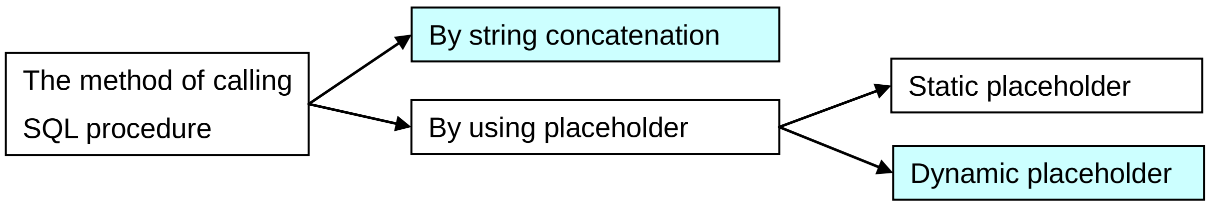
As shown in the figure above, an SQL statement is sent to the database engine after parameter binding every time an SQL procedure is called. For that, the execution efficiency is lower than the static placeholder method. Yet some of the database engines offer the dynamic placeholder feature instead of the static placeholder feature. Sometimes the dynamic placeholder feature is called client-side prepared statements but remember that they are different from so-called prepared statements defined by JIS/ISO.

From security perspective, because insertion of the parameter values is done through binding using a placeholder mechanically by the library, it is expected that escaping errors by application developers are prevented more effectively compared to building the SQL statement through string concatenation.

The dynamic placeholder, however, is different from the static placeholder and there is a possibility that some libraries - the very mechanism that performs binding - may have a vulnerability and enables SQL injection attacks.

3.3. Chapter Summary

The methods of calling SQL procedure discussed in this chapter are summarized below.



The method of using the static placeholder is free of SQL injection vulnerability because the syntax of the SQL statement is fixed when preparing the statement and does not change later.

On the other hand, with the method of using the dynamic placeholder, there is a possibility that some libraries - the very mechanism that performs binding - may have a vulnerability and enables SQL injection attacks.

The method of building a statement by string concatenation has a risk of escaping errors by application developers. It is a challenge since the metacharacters that need to be escaped differ depending on the type and configuration of database engines and it is difficult to develop each application accordingly and appropriately to them.

In the next chapter, we closely look into the problems the string concatenation method and dynamic placeholder method would present.

4. Building SQL Statement Appropriately to Database Engine

In this chapter we look into what we should do to safely call SQL procedure by string concatenation or using the dynamic placeholder.

4.1. Using the quote Method for Concatenation

To safely call SQL procedure through building an SQL statement by string concatenation, you must satisfy the following requirements.

- For string literals, escape the characters that should be escaped
- For numeric literals, make sure that non-numeric value is not inserted as their value

As Appendix A.1 shows, however, the metacharacters that need to be escaped when generating string literals differ depending on the type of database engine and sometimes on its configuration as well. If you do not implement the escaping process accordingly and appropriately to the database engine in use, you may jeopardize your system with the risk of SQL injection vulnerability.

Since implementing the escaping process can be troublesome, some programming languages and database engines provide a special method or function to generate SQL literals as strings.

The `quote` method offered by Perl DBI, PHP Pear::MDB2 and PDO (PHP Data Objects) is a versatile method dedicated to generate SQL literals and application developers can use it to implement the escaping process properly without regard to the type and configuration of database engines.

【Calling the `quote` Method with PHP Pear::MDB2】

```
require_once 'MDB2.php';//Load libraries

// Connect to DB (in the case of PostgreSQL)
$db = MDB2::connect('pgsql://dbuser:password@hostname/dbname?charset=utf8');

// Specify the string type and get a quoted string for the string literal
(snip) $db->quote($s, 'text') (snip)

// Specify the numeric type and get a string for the numeric literal
(snip) $db->quote($n, 'decimal') (snip)
```

PHP

As shown above, specify the type of literal to be generated in the second parameter of the `quote` method.

If the character string type is specified as 'text', the method executes the escaping process accordingly and appropriately to the type and configuration of the database engine in use and returns a single-quoted string. If a numeric type such as 'decimal' is specified, it returns a string appropriate as a numeric literal.

The table below shows the examples of the return values when some data are given to the quote method of PHP Pear::MDB2.

Input Data	**Type**	**Return Value**
abc	'text'	'abc' (PHP character string-typed value, incl. quotes)
O'Reilly	'text'	'O''Reilly' (PHP character string-typed value, incl. quotes)
-123	'decimal'	-123 (PHP character string-typed value)
123abc	'decimal'	123 (PHP character string-typed value)
-123	'integer'	-123 (PHP integer-typed value)
123abc	'integer'	123 (PHP integer-typed value)

These are the literals generated from the correct escaping process[1]. By using this method, you can prevent SQL syntax errors and in turn, SQL injection vulnerability.

However, according to IPA's study, the quote method may fail to return the correct, expected value depending on the combination between the programming language and the database engine. We will look into this problem in Chapter 5.

4.2. Dynamic Placeholder and Database Engine

The dynamic placeholder method executes string concatenation mechanically by using the libraries or drivers instead of by applications. First, we see how an SQL statement is built using the dynamic placeholder.

One example that is implemented with the dynamic placeholder method is the DBD::mysql module of Perl DBI. With the placeholder feature provided by the DBD::mysql module, you can explicitly specify whether you want to use the dynamic placeholder or static placeholder as an option when connecting to the database.

[1] For PHP's Pear::MDB2, when you specify 'integer' as a type with the quote method, the return value is not a character string but a PHP integer-typed value. The value is later converted into a character string when concatenated as a part of the SQL statement.

```
my $db = DBI->connect("DBI:mysql:$dbname:$host;mysql_server_prepare=0",
                      $user, $pwd)  || die $DBI::errstr;
my $sql = "SELECT * FROM test3 where age >= ? and name = ?";
my $sth = $db->prepare($sql);
$sth->bind_param(1, 27, SQL_INTEGER);
$sth->bind_param(2, 'YAMAMOTO', SQL_VARCHAR);
my $rt = $sth->execute();
```

Perl

In this example, the dynamic placeholder is specified with the option `mysql_server_prepare=0`.

The SQL statement built from the above program is as follows.

```
SELECT * FROM test3 where age >= 27 and name = 'YAMAMOTO'
```

SQL

With the same program, if the value `O'reilly` and `\100` are given to the name identifier, then the resulting SQL statements are as follows.

```
SELECT * FROM test3 where age >= 27 and name = 'O\'reilly'
SELECT * FROM test3 where age >= 27 and name = '\\100'
```

SQL

Since the default settings of MySQL define that ' and \ are to be escaped, the results become just as seen above. If you set the `NO_BACKSLASH_ESCAPES` option in MySQL, then the program performs escaping just on ' as defined by the JIS/ISO standard (see Appendix A.1). In this case, the resulting SQL statement will become as follows.

```
SELECT * FROM test3 where age >= 27 and name = 'O''reilly'
SELECT * FROM test3 where age >= 27 and name = '\100'
```

SQL

Like this, the DBD::mysql module performs escaping accordingly and appropriately depending on the MySQL settings.

Next, we see what will happen when giving a non-numeric value to the first placeholder. When `1 or 1=1` is inputted, the following will occur.

【SQL Statement Generated】

```
SELECT * FROM aTable where age >= 1 and name = 'Yamamoto'
```

SQL

【Error Message】

```
DBD::mysql::st bind_param failed: Binding non-numeric field 1, value '1 or 1=1' as
a numeric! at C:¥・・・file name・・・ line 23.
```

Txt

Because the error message says `failed`, it looks as if the SQL statement was not executed. But it indeed was executed. In this case, the character string `1 or 1=1` was converted into a digit (1) and bound to the first placeholder, and then the statement was executed. Since the string had been converted into a numeric value, an SQL injection attempt was prevented.

In Chapter 5, we will look into whether the processing related to the dynamic placeholder behaves correctly and expectedly with some combination between the programming language and database engine.

5. DBMS Study

5.1. Study Purpose

Based on what we presented in the previous chapters, we studied and clarified the following points for the combinations between some programming languages and database engines often used to develop web applications.

- Whether implementation of the placeholder is static (prepared statement) or dynamic
- Whether the escaping process with the dynamic placeholder is correctly and expectedly done or not
- Whether the `quote` method is correctly and expectedly done or not
- What character encoding can be used

5.2. Java + Oracle

When calling Oracle stored procedure from Java, it is common to use JDBC, and there are some JDBC implementations for Oracle. In this study, IPA picked up ojdbc6.jar provided by Oracle Corporation and used the character encoding for the database as UTF-8 (see Appendix A.4).

【Study Results】

Study Item	**Results**
Placeholder Implementation	Static placeholder only
Escaping Process with Dynamic Placeholder	N/A (the dynamic placeholder feature not provided)
The `quote` Method	N/A (the `quote` method not provided)
Character Encoding	Use UTF-8 when connecting to the database

Since only the static placeholder is used with the Java + Oracle + ojdbc6.jar combination, you do not have to do anything special as long as you use the JAVA `PreparedStatement` interface.

Because Java does not provide a library that is equivalent to the `quote` method and therefore you cannot perform escaping accordingly and appropriately to the type and configuration of database engines automatically, it is not recommended using string concatenation to build SQL statements.

5.2.1. Sample Code

```java
import java.sql.*;

public class OraclePrepared {
  public static void main(String[] args) {
    String url = "jdbc:oracle:thin:@SERVERNAME:1521:ORCL";
    try {
      // Load JDBC driver
      Class.forName("oracle.jdbc.OracleDriver");
      // Connect to the database
      db = DriverManager.getConnection(url, userName, password);

      String param = ....

      String sql = "SELECT * FROM atable WHERE name=?";
      PreparedStatement stmt = con.prepareStatement(sql);
      stmt.setString(1, param);   // Replace ? with the value
      ResultSet rs = stmt.executeQuery();
      while(rs.next()){
        int id = rs.getInt("id");
        String name = rs.getString("name");
        String address = rs.getString("address");
        String comment = rs.getString("comment");
        System.out.printf("id = %d  name = %s  address = %s  comment = %s\n",
 id, name, address, comment);
      }
      rs.close();
      stmt.close();
      con.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Java

5.3. PHP + PostgreSQL

There are various libraries that can call PostgreSQL procedure from PHP. IPA picked up PEAR::MDB2 based on the points below.

- Other than PostgreSQL, MDB2 provides versatile interface capability for calling SQL procedure to multiple DBMSs, such as MySQL, and Oracle.
- MDB2 still keeps being developed while other modules that offer the same capability, such as Pear::DB and Pear::MDB, will be no longer modified and updated.
- MDB2 gives consideration to character coding issues.
- The static placeholder is available with MDB2.
- MDB2 gives consideration to the data types when quoting and binding the values into the placeholders.

The study results are presented below.

【Study Results】

Study Item	**Result**
Placeholder Implementation	Static placeholder only
Escaping Process with Dynamic Placeholder	N/A (the dynamic placeholder feature not provided)
The `quote` Method (string literal generation)	Correctly done
The `quote` Method (numeric literal generation)	Correctly done
Character Encoding	Enable to specify when connecting to the database

Since only the static placeholder is used with the PHP + MDB2 + PostgreSQL combination, you do not have to do anything special as long as you use the placeholder.

You can use the `quote` method instead of the placeholders. The `quote` method in this combination generates both string literals and numeric literals correctly.

5.3.1. Sample Code

A sample SQL call program using MDB2 is shown below.

```
<?php
require_once 'MDB2.php';//Load libraries

$db = MDB2::connect('pgsql://username:password@hostname/dbname' .
'?charset=utf8');
if(PEAR::isError($db)) {
   //Error processing
}

$stmt = $db->prepare('SELECT * FROM atable WHERE name=? and num=?',
            array('text', 'integer'), array('text', 'text', 'integer'));
$rs = $stmt->execute(array($name, $num)); //String-typed and integer-typed variables
if(PEAR::isError($rs)) {
    //Error processing
}

//Display search results
while($row = $rs->fetchRow()) {
    printf("%s:%s:%s\n", $row[0], $row[1], $row[2]);
}
```

PHP

【Points When Connecting to the Database】

1. Specifying character encoding

   Specify the character encoding to communicate with PostgreSQL as UTF-8.

```
charset=utf8
```

Txt

In PHP, you can specify the character encoding to be used in the program in the configuration file php.ini. If you specify the character encoding as Shift_JIS, the code of 5C, which may appear in the second byte for Shift_JIS, is interpret as \ and exploited in SQL injection attacks. For that reason, IPA recommend using UTF-8 or EUC-JP. IPA also suggests setting the same character encoding for both the PHP program and the database.

2. Specifying type of placeholder

   You can specify the type of the placeholder. If not specified, the database considers it as varchar (the type to store variable length character strings) and an implicit type conversion from the string type to the actual type will take place when executing the SQL statement. This may cause unexpected errors or performance degradation.

```
$stmt = $db->prepare('SELECT * FROM atable WHERE name=? and num=?',
            array('text', 'integer'), array('text', 'text', 'integer'));
```

PHP

In the example above, those specified in the second parameter are the type of the placeholders. The third parameter specifies the type of the return values and is optional.

【Escaping When Placeholder Is Unavailable】

If the placeholder is unavailable for use for some reason, you can use the `quote` method and successfully perform escaping that is appropriate to the data types and configuration of the database engine.

In PostgreSQL, the backslash \ is processed as one of the metacharacters that need to be escaped with the standard settings, but if you enable `standard_conforming_strings` (making it `on`), the backslash is no longer considered as a metacharacter (see Appendix A.1). Thus, escaping must be done accordingly to the content of the `standard_conforming_strings` setting. The `quote` method in MDB2 does it automatically.

In addition, you can specify the data type with the `quote` method in MDB2.

```
$sql = 'SELECT * FROM atable WHERE name=' . $db->quote($name, 'text') . ' and num=' .
$db->quote($num, 'integer');
```

PHP

`$db->quote ('10', 'integer')` returns a PHP integer-typed value of 10 that is to be interpreted as an SQL numeric literal of 10, and is later converted to a PHP character string of 10 when string concatenation occurs. Likewise, `$db->quote ('10', 'text')` returns a PHP character string of 10 that is to be interpreted appropriately as an SQL string literal later. By using this function, escaping that is appropriate to the data type will be achieved.

5.4. Perl + MySQL

When calling SQL procedure from Perl, it is common to use a DBI module. The DBI modules provide a versatile interface capability for calling SQL procedure and there is a DBI module appropriate for each DBMS.

When calling SQL procedure using a DBI module, you can use the placeholder with the `prepare` method, and the `quote` method is also available.

The study results of the combination of PerlDBI and DBD::MySQL are presented below.

【Study Results】

Study Item	Results
Placeholder Implementation	Dynamic or static
Escaping Process with Dynamic Placeholder	Correctly done
The `quote` Method (string literal generation)	Correctly done
The `quote` Method (numeric literal generation)	Incorrectly done (returns the input data as they are)
Character Encoding	Enable to specify UTF-8 explicitly when connecting to the database

5.4.1. Implementation of Placeholder

With the default settings, the dynamic placeholder is selected. If you want to use the static placeholder, you need to specify the parameter `mysql_server_prepare=1` when connecting to the database.

5.4.2. Metacharacters to Be Escaped

In MySQL, the backslash \ is processed as one of the metacharacters that need to be escaped with the standard settings, but if you enable the `NO_BACKSLASH_ESCAPES` setting, the backslash is no longer considered as a metacharacter (see Appendix A.1). Thus, escaping must be done accordingly to the content of this setting. The `quote` method in DBD::MySQL does it automatically and performs escaping for string literals correctly.

5.4.3. Numeric Literals with the quote Method

Even if the numeric type is specified in the second parameter of the `quote` method, the method does not check whether the input data are appropriate for the specified data type nor convert them to the numeric value, and just returns the input data as they are.

【Example】

```
$dbh->quote("1 or 1=1", SQL_INTEGER);  # → returns "1 or 1=1"
```

Perl

Thus, the quote method cannot be used as a countermeasure against SQL injection and it is not recommended using string concatenation to build SQL statements with the current specification of DBD::MySQL. Unless you have inevitable reasons, such as needing it as an ad-hoc countermeasure against SQL injection for an existing application, the use of the placeholders is recommended.

5.4.4. Character Encoding

After Perl 5.8, the use of UTF-8 is recommended as its character encoding within the programs. By specifying mysql_enable_utf8=1 when connecting to the database, you can set the same character encoding for both the Perl program and the database.

5.4.5. Sample Code

A sample SQL call program using DBI/DBD is shown below.

```
#!/usr/bin/perl
use CGI;
use DBI;
use DBI qw(:sql_types);
use strict;
use utf8;
use Encode 'decode', 'encode';

my $db = DBI->connect(
'DBI:mysql:database=xxxx;host=xxxx;mysql_server_prepare=1;mysql_enable_utf8=1',
'xxxx', 'xxxx');
if (! $db) {
    # Connection error process
}
my $sql = 'SELECT * FROM antable WHERE num=? AND name=?';
my $sth = $db->prepare($sql);

$sth->bind_param(1, $num, SQL_INTEGER);
$sth->bind_param(2, $name, SQL_VARCHAR);

my $rt = $sth->execute();
if (! $rt) {
    # SQL call error process
}
# search result fetching process
$sth->finish;
$db->disconnect;
```

Perl

【Points When Connecting to the Database】

1. Specifying type of placeholder as static

   Specify the type of placeholder as static. Otherwise, the dynamic placeholder will be specified and escaping will be performed within the database connection driver before SQL call.

   ```
   mysql_server_prepare=1
   ```

   Txt

By using the static placeholder, SQL statements and parameter values are sent separately to the database engine. Since syntax analysis of SQL statements is done before the placeholder symbol ? in the SQL statements is replaced, the risk of SQL injection can be avoided in principal.

2. Specifying character encoding
   Specify the character encoding as UTF-8.

```
mysql_enable_utf8=1
```

Txt

   The line above specifies the character encoding to connect to the database as UTF-8.
   After Perl 5.8, by using `Encode.pm`, character encoding for the internal process is performed with UTF-8. By also specifying character encoding to connect to the database as UTF-8, you can avoid garbled characters and the security issues caused by character encoding.

3. Specifying type of binding process
   Use the `bind_param` method for binding and specify the data type in the third parameter of the `bind_param` method.

```
$sth->bind_param(1, $num, SQL_INTEGER);
$sth->bind_param(2, $name, SQL_VARCHAR);
```

Perl

   You can also provide parameter values using the `execute` method. In that case, however, the data type of the parameters are all considered as `varchar` and an implicit type conversion from the string type to the actual type will take place when executing the SQL statement. Since the type conversion from the string type to the numeric type in MySQL is performed using the floating point type in the process, the accuracy can be impaired. For this reason, IPA recommends you explicitly specify the data type in the `bind_param` method for binding in MySQL.

5.5. Java + MySQL

To call SQL procedure from Java, you can use JDBC (MySQL Connector/J). JDBC does not provide the `quote` method, therefore, you need to use the `PrepeardStatement` interface to call SQL procedure.

【Study Result】

Study Item	Result
Placeholder Implementation	Dynamic or static
Escaping Process with Dynamic Placeholder	Correctly done (depending on the version, however, the problem addressed in Appendix A.3 will occur)
The `quote` Method	N/A (the `quote` method not provided)
Character Encoding	Enable to specify when connecting to the database

5.5.1. Implementation of Placeholder

With the default settings, the dynamic placeholder is selected. If you want to use the static placeholder, you need to specify the parameter `useServerPrepStmts=true` when connecting to the database.

5.5.2. Metacharacters to Be Escaped

In MySQL, the backslash \ is processed as one of the metacharacters that need to be escaped with the standard settings, but if you enable the `NO_BACKSLASH_ESCAPES` setting, the backslash is no longer considered as a metacharacter (see Appendix A.1). Thus, escaping must be done accordingly to the content of this setting. If you use the dynamic placeholder in MySQL Connector/J, it is done automatically and escaping for string literals is performed correctly.

5.5.3. Character Encoding

You can specify what character encoding to use with the `characterEncoding` parameter when connecting to the database. In Appendix A.3, IPA recommends the use of UTF-8.

5.5.4. Sample Code

```
import java.sql.*;

public class MysqlPrepared {
  public static void main(String[] args) {
    String url = "jdbc:mysql://HOSTNAME/DBNAME?user=USERNAME&password=PASSWORD&"
              + "useUnicode=true&characterEncoding=utf8&useServerPrepStmts=true";
    try {
      Class.forName("com.mysql.jdbc.Driver");
      Connection con = DriverManager.getConnection(url);

      String param = ....

      String sql = "SELECT * FROM atable WHERE name=?";
      PreparedStatement stmt = con.prepareStatement(sql);
      stmt.setString(1, param);   // Replace ? with the value.
      ResultSet rs = stmt.executeQuery();
      while(rs.next()){
        int id = rs.getInt("id");
        String name = rs.getString("name");
        String address = rs.getString("address");
        String comment = rs.getString("comment");
        System.out.printf("id = %d  name = %s  address = %s  comment = %s\n",
                          id, name, address, comment);
      }
      stmt.close();
      con.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Java

5.6. ASP.NET + Microsoft SQL Server

When calling SQL procedure in the combination of Microsoft SQL Server and ASP.NET, you can use ADO.NET. Other than calling SQL procedure through ODBC and OLE.DB, ADO.NET can use the drivers for Microsoft SQL Server and Oracle as well.

【Study Result】

Study Item	**Result**
Placeholder Implementation	Static placeholder only
Escaping Process with Dynamic Placeholder	N/A (the dynamic placeholder feature not provided)
The `quote` Method	N/A (the `quote` method not provided)
Character Encoding	Use UTF-16 when connecting to the database

Since only the static placeholder is used with the ASP.NET + Microsoft SQL Server combination, you do not have to do anything special as long as you use the placeholder.

5.6.1. Sample Code

A sample SQL call program using ADO.NET with the Microsoft SQL Server driver (.NET Framework Data Provider for SQL Server) written in Visual Basic .NET is shown below.

```
Imports System.Data.SqlClient

Partial Class SqlSample
Inherits System.Web.UI.Page

'Page loading process
Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles
Me.Load

Dim dbcon As SqlConnection
Dim dbcmd As SqlCommand
Dim dataRead As SqlDataReader
Dim sqlStr As String

'Create DB connection
dbcon = New SqlConnection(
       "Server=HOSTNAME; database=DBNAME;userid=USERID;password=PASSWORD")

'Connect DB
dbcon.Open()

'SQL statement
sqlStr = "select * from aTalbe where name=@s1"

'Build SQL command
dbcmd = New SqlCommand(sqlStr, dbcon)

Dim param As String = ....

'Set paramaters
Dim p1 As SqlParameter = New SqlParameter("@s1",param)
dbcmd.Parameters.Add(p1)

'Execute SQL statement
dataRead = dbcmd.ExecuteReader()

'Fetch result(snip)

'Closing process
dataRead.Close()
dbcmd.Dispose()
dbcon.Close()
dbcon.Dispose()

end sub
End class
```

Visual Basic .NET

Because .NET Framework does not provide a library that is equivalent to the `quote` method and therefore you cannot perform escaping accordingly and appropriately to the type and configuration of database engines automatically, it is not recommended using string concatenation to build SQL statements.

5.7. Summary

The summary of the study is shown in the table below.

	Java +Oracle	PHP +MDB2 +PostgreSQL	Perl +MySQL	Java +MySQL	ASP.NET +SQL Server
Placeholder Implementation	Static Only	Static Only	Static or Dynamic	Static or Dynamic	Static Only
Escaping Process with Dynamic Placeholder	—	—	○	△	—
The quote Method (string literal generation)	—	○	○	—	—
The quote Method (numeric literal generation)	—	○	×	—	—
Character Encoding	Defined as UTF-8	Able to Specify	Able to Explicitly Specify UTF-8	Able to Specify	Defined as UTF-16

If the implementation of the placeholder limits that only the static placeholder can be used, the developers do not need to do anything special as long as using the placeholder.

Note that some methods like `prepare`, whose name sounds like being implemented with the static placeholder (prepared statement), use the dynamic placeholder with the default setting and you must be careful when using them.

When using the dynamic placeholder, you must be careful with some specific conditions. For example, SQL injection vulnerability may be built in due to the character encoding problems depending on the implementation of the drivers and libraries, like the risk posed by using the older version of MySQL Connector/J with the Java + MySQL combination.

If you need to use string concatenation to build SQL statements for some inevitable reasons, you can use the `quote` method and expect it to perform escaping accordingly and appropriately to the type and configuration of database engines. Note that in some cases like the Perl + MySQL combination, however, generation of numeric literals may not be done correctly.

The study results not shown this time suggest that in some other environments, such as Perl DBD::PgPP (Pure Perl PostgreSQL driver for the DBI), the dynamic placeholder and the `quote` method may not perform escaping accordingly and appropriately to the type and configuration of database engines (confirmed with DBD::PgPP version 0.08).

As for character encoding when connecting to the database, UTF-8 or UTF-16 is predefined in some environments, and in other environments, it may be better specified explicitly. The study results not shown this time suggest that in some environments, such as PHP PDO (PHP Data Objects), the character encoding specified may be ignored (confirmed with PDO version 1.0.4dev).

Appendix A.Technical Information

A.1. Escaping of Backslash

Depending on the database products, characters other than single quotes must be escaped. Typical example are MySQL and PostgreSQL. With these databases, backslash is interpreted as one of the metacharacters to escape a single quote. Thus, if a backslash is contained in string literals, the backslash itself needs to be escaped as well.

Escape Target	Escape Method
’	’’ (\’ can be used as well)
\	\\

You can also change the settings not to interpret backslash as a metacharacter.

MySQL Option	`NO_BACKSLASH_ESCAPES` Do not interpret backslash as a metacharacter when escaping
PostgreSQL Option	`standard_conforming_strings` Do not interpret backslash as a metacharacter when escaping

If you use the database products that require escaping of backslash but neglect it, you expose your system to the risk of SQL injection vulnerability. We will show it using a SQL call example below.

```
$q = ″SELECT * FROM atable WHERE a=’$s’″;
```
Perl

Here, a string below is given as an attack code.

```
\’;DELETE FROM atable--
```
Txt

Performing escaping of just single quotes gives the following result.

```
\’’;DELETE FROM atable--
```
Txt

Now, the result will be put into the original SQL statement.

```
SELECT * FROM atable WHERE a=’\’’;DELETE FROM atable--’
```
SQL

As mentioned above, \’ is interpreted as the metacharacters to escape a single quote and the following another single quote escapes the escaping process.

```
SELECT * FROM atable WHERE a='\'';DELETE FROM atable--'
                                 ↑single quote missed to escape
```

SQL

As a result, the string after ;DELETE spills out of the intended literal and is interpreted as another SQL statement.

On the other side, if you use the database products that do not require escaping of backslash yet you escape it, double backslashes will be stored in the database, corrupting the accuracy of the data as the result of wrong escaping.

A.2. SQL Injection Risk with Shift_JIS

When you implement escaping for string literals, you must take into account character encoding issues. SQL injection vulnerability especially and likely occurs in an environment where you use Shift_JIS as character encoding and you need to escape backslash \, but problems may occur when other conditions are met.

We will see it using a SQL call example below.

```
$q = "SELECT * FROM atable WHERE a='$id'";
```

Perl

Here, a string below is given to $id.

```
表';DELETE FROM atable--
```

Txt

The Japanese character code for the 表' part is as below:

表		'
0x95	0x5c	0x27

0x5c represents a backslash in US-ASCII and ISO-8859-1. If you perform escaping for this string without consideration of character encoding, 0x5c will be interpreted as one of the escaping targets. The result of the escape process may become as follows:

0x95	0x5c	0x5c	0x27	0x27

Decode it by Shift_JIS, it becomes like below.

表		\	'	'
0x95	0x5c	0x5c	0x27	0x27

Now, the result will be put into the original SQL statement.

```
SELECT * FROM atable WHERE a='表\'';DELETE FROM atable--'
```
SQL

Since the program interprets that the backslash \' escapes the first single quote, the string literal ends with the second single quote. As the result, the string after the semicolon spills out of the intended literal and is interpreted as another SQL statement.

For this reason, you should take into account the following points when implementing character encoding.

- Use a database engine, placeholder, library like the `quote` method, application programming language with which string processing is properly implemented with regard to character encoding
- Avoid the use of Shift_JIS which increases the risk of SQL injection vulnerability due to character encoding errors

A.3. SQL Injection Risk with Unicode

When using the combination of MySQL and Java (MySQL Connector/J), even if you use the placeholder, SQL injection may occur if the following conditions are met.

- MySQL Connector/J version 5.1.7 or before is used
- Shift_JIS or EUC-JP is used as character encoding for the connection between the database and application
- The dynamic placeholder is used

Let's look at a concrete example.

```
Connection con = DriverManager.getConnection(
"jdbc:mysql://HOSTNAME/DBNAME?user=USERNAME&password=PASSWORD&useUnicode=true&ch
aracterEncoding=sjis");
```
Java

The example above uses the dynamic placeholder by not specifying the `useServerPrepStmts` parameter (or setting `false` to the parameter) when connecting to the database. It also uses Shift_JIS or EUC-JP as character encoding for the connection between the database and application by specifying `characterEncoding=sjis` (or not specifying the `characterEncoding` parameter but by setting the server-side character to `sjis or ujis` in `my.ini`).

If the following string is given and bound in this environment, SQL injection will occur.

```
"\u00a5'or 1=1#"
```
Java

`\u00a5` in the string literal written in Java programming language means `U+00A5` in Unicode and assigned with the Yen symbol ¥. In Unicode, the Yen symbol ¥ and the backslash symbol \ (`U+005C`) exist separately. Shift_JIS and EUC-JP, however, do not differentiate them and convert both into ¥ (0x5C). As a result, the following phenomenon occurs when building SQL statements.

【Input Strings (Unicode)】

Code Point	00A5	0027	006F	0072	0020	0031	003D	0031	0023
Character	¥	’	o	r	SP	1	=	1	#

【Output String after Escape Process (Unicode)】

Code Point	00A5	005C	0027	006F	0072	0020	0031	003D	0031	0023
Character	¥	\	’	o	r	SP	1	=	1	#

【String after Converted into Shift_JIS】

Character Code	5C	5C	27	6F	72	20	31	3D	31	23
Character	¥	¥	’	o	r	SP	1	=	1	#

【SQL Statement (input value is bound to the dynamic placeholder)】

```
SELECT * FROM test WHERE name='¥¥' or 1=1#'
```

SQL

Here, ¥¥ is interpreted as a metacharacter that escaped the backslash and the string literal ends with the following single quote. As a result, the subsequent string of or 1=1# is interpreted as a part of the SQL statement and hereby occurs a successful SQL injection attack.

The cause of this problem is that the character encoding used in the escaping process while building an SQL statement using the dynamic placeholder and the character encoding used when the SQL statement is executed are different, and two different characters are assigned to the same character when the character encoding conversion from the one used in the escaping process to the one used in execution.

This problem is identified as a vulnerability in MySQL Connector/J (5.1.7 and before) and was fixed in July 2009[2]。

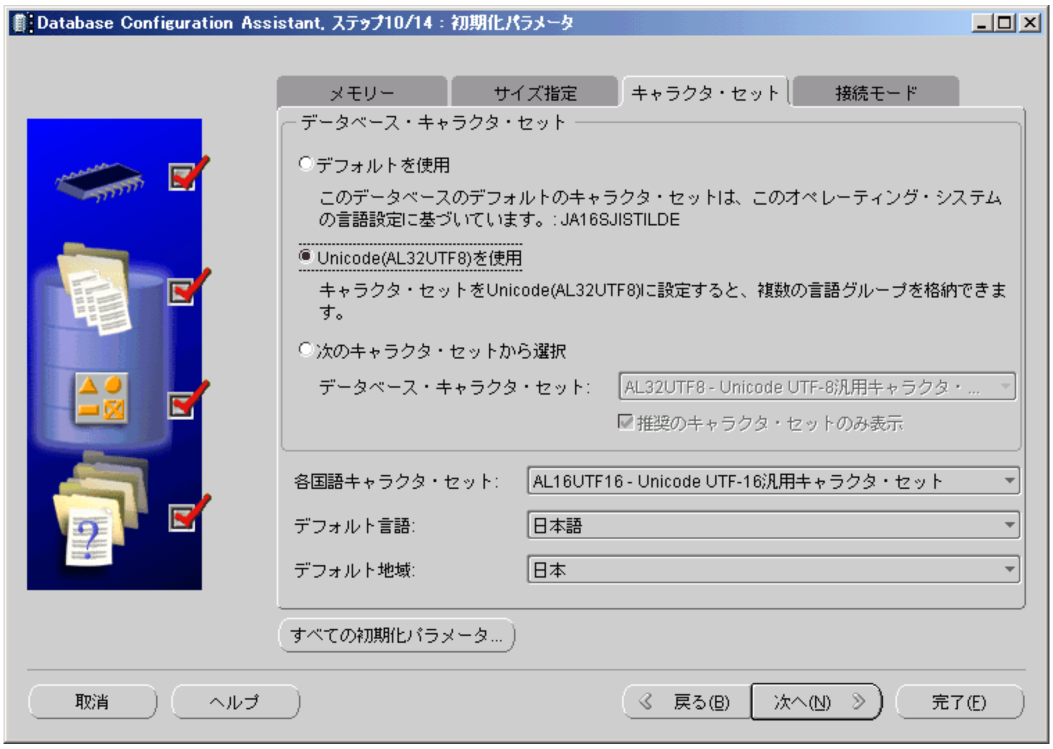
To fix this vulnerability, you must implement one or more of the following. IPA recommends you implement all of them.

- Specify Unicode (UTF-8) as the character encoding when connecting to MySQL
  (Set the connection character string parameter characterEncoding=utf8)
- Use the static placeholder
  (Set the connection character string parameter useServerPrepStmts=true)
- Use the latest version of MySQL Connector/J

[2] JVN#59748723: MySQL Connector/J vulnerable to SQL injection
http://jvn.jp/jp/JVN59748723/index.html

A.4. Creating Oracle Database with Unicode

You cannot choose the character encoding for each table or column with Oracle but only one for the database. Since you cannot change the character encoding for an existing database, you must think over and decide what character encoding to use when creating a new database.



データベース・キャラクタ・セット

○ デフォルトを使用

このデータベースのデフォルトのキャラクタ・セットは、このオペレーティング・システムの言語設定に基づいています。: JA16SJISTILDE

● Unicode(AL32UTF8)を使用

キャラクタ・セットをUnicode(AL32UTF8)に設定すると、複数の言語グループを格納できます。

○ 次のキャラクタ・セットから選択

データベース・キャラクタ・セット: AL32UTF8 - Unicode UTF-8汎用キャラクタ・...

☑ 推奨のキャラクタ・セットのみ表示

各国語キャラクタ・セット: AL16UTF16 - Unicode UTF-16汎用キャラクタ・セット

デフォルト言語: 日本語

デフォルト地域: 日本

すべての初期化パラメータ...

The figure above is the Oracle Database Configuration Assistant wizard with which the character set is being chosen. Here, the character set is changed from the default database character set, Shift_JIS (JA16SJISTILDE), to .Unicode（AL32UTF8）.

It does not mean that not choosing Unicode as the database character set always results in SQL injection vulnerability, but there is the risk that characters stored in the database may be converted into other characters, causing new and unexpected bugs. IPA recommends using Unicode.

A.5. Microsoft SQL Server and Character Encoding

Microsoft .NET, uses UTF-16 as the character encoding for internal processing. On the other hand, the character encoding used when storing character strings to the Microsoft SQL Server tables depends on the code page used in the operating environment where Microsoft SQL Server is installed. In the Japanese environment, the code page 932 (CP932) is used.

UTF-16 encoded strings passed from the applications to Microsoft SQL Server are converted into CP932 when being stored to the database. Therefore, a character that is not defined in CP932 is turned into garbage and cannot be stored in the database correctly.

Microsoft SQL Server provides a way to store Unicode encoded strings. When storing Unicode encoded strings, use nchar or nvachar as the data type instead of char or vachar. For string literals, precede all Unicode string literals with a prefix N that identifies they are Unicode encoded.

【Example: NVARCHAR】

```
CREATE TABLE aTable (name NVARCHAR(30), city NVARCHAR(30));
```

SQL

【Example: Unicode String Literal】

```
INSERT INTO aTable VALUES (N'Sato', N'Yokohama');
```

SQL

Characters may be garbled in some cases. For example, Unicode U+00A5 ¥ (Yen symbol) may be converted into CP932 5C \ (backslash).

This does not become a cause of SQL injection vulnerability because this character encoding conversion takes place after syntax analysis of SQL statements is done. It may, however, cause unexpected bugs or other vulnerabilities.

【References】

Microsoft MSDN – Using Unicode Data
http://msdn.microsoft.com/en-us/library/aa223981(SQL.80).aspx

Produced and Copyrighted by Information-technology Promotion Agency, Japan (IPA)

Editor Hideaki Kobayashi

Author Hiroshi Tokumaru, Yukinobu Nagayasu, Motokuni Soma, Naoto Katsumi
Hiromitsu Takagi National Institute of Advanced Industrial Science and Technology (AIST)

Advisor Hiroyuki Itabashi, Shunsuke Taniguchi, Shingo Ootani, Masashi Ohmori
Tadashi Yamagishi Hitachi, Ltd.

*Affiliation omitted for the personnel of IPA

How to Secure Your Web Site Supplementary Volume

How to Use SQL Calls to Secure Your Web Site

[Publication] Mar. 18, 2010 First Edition, First Printing

[Produced and Copyrighted by] IT Security Center, Information-technology Promotion Agency, Japan (IPA)
[Collaborated with] Research Center for Information Security, National Institute of Advanced Industrial Science and Technology

How to Report Information Security Issues to IPA

Designated by the Ministry of Economy, Trade and Industry, IPA IT Security Center collects information on the discovery of computer viruses and vulnerabilities, and the security incidents of virus infection and unauthorized access.

Make a report via web form or email. For more detail, please visit the web site:
URL: http://www.ipa.go.jp/security/todoke/ (Japanese only)

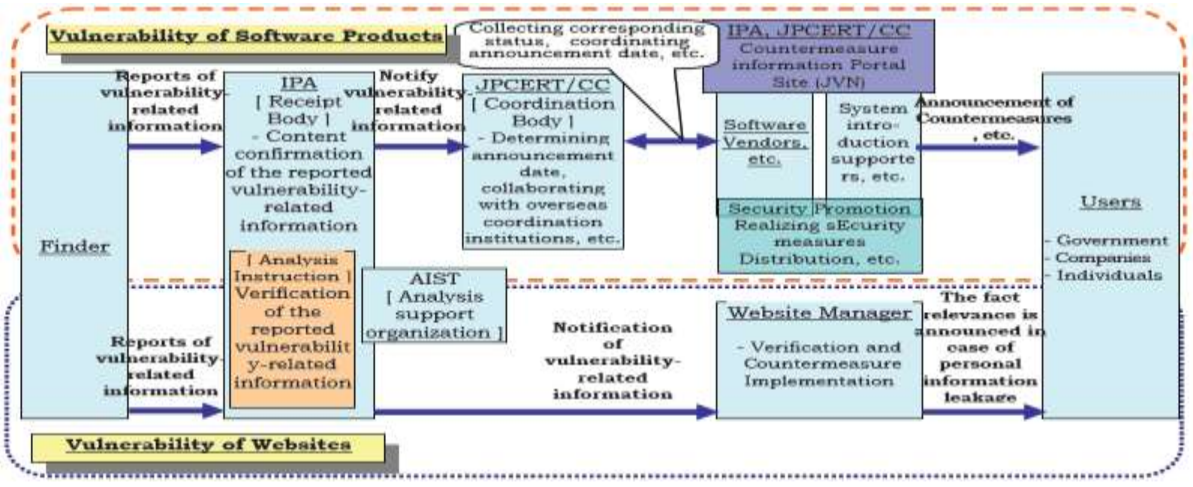
Computer Viruses
When you discover computer viruses or notice that your PC has been infected by viruses, please report to IPA.

Unauthorized Access
When you detect unauthorized access to your network, such as intranets, LANs, WANs and PC communications, please report to IPA.

Software Vulnerability and Related Information
When you discover vulnerabilities in client software (ex. OS and browser), server software (ex. web server) and hardware embedded software (ex. printer and IC card) , please report to IPA.

Web Application Vulnerability and Related Information
When you discover vulnerabilities in systems that provide their customized services to the public, such as web sites, please report to IPA.

Framework for Handling Vulnerability-Related Information
～ Information Security Early Warning Partnership ～


Vulnerability of Software Products
Collecting corresponding status, coordinating announcement date, etc.
IPA, JPCERT/CC Countermeasure information Portal Site (JVN)
Finder
Reports of vulnerability-related information
IPA [Receipt Body] - Content confirmation of the reported vulnerability-related information
Notify vulnerability-related information
JPCERT/CC [Coordination Body] - Determining announcement date, collaborating with overseas coordination institutions, etc.
Software Vendors, etc.
System intro-duction supporters, etc.
Security Promotion Realizing sEcurity measures Distribution, etc.
Announcement of Countermeasures , etc.
Users
- Government
- Companies
- Individuals
[Analysis Instruction] Verification of the reported vulnerability-related information
AIST [Analysis support organization]
Reports of vulnerability-related information
Notification of vulnerability-related information
Website Manager
- Verification and Countermeasure Implementation
The fact relevance is announced in case of personal information leakage
Vulnerability of Websites

JPCERT/CC: Japan Computer Emergency Response Team Coordination Center, AIST: National Institute of Advanced Industrial Science and technology

IPA®

INFORMATION-TECHNOLOGY PROMOTION AGENCY, JAPAN
2-28-8 Honkomagome, Bunkyo-ku, Tokyo 113-6591 JAPAN
http://www.ipa.go.jp/index-e.html

IT SECRITY CENTER
Tel: +81-3-5978-7527 FAX: +81-3-5978-7518
http://www.ipa.go.jp/security/english/