Prepared Statements

Topic 4 Lesson 4
Creating a prepared SQL statement

SQL Injection: motivation for security

SQL injection refers to the act of someone inserting or modifying a SQL statement to be run on your database without your knowledge.

Injection usually occurs when you ask a user for input, like their name, and instead of a name they give you a MySQL statement that you will unknowingly run on your database.

SQL injection provides access to data that a user should not have access to.

SQL Injection example

```
-- a valid user's name
SET @name = ” ‘timmy’ ";
set @query = CONCAT("SELECT * FROM customers WHERE username
   = " , @name,';');
SELECT   @query ;
```

SELECT * FROM customers WHERE username = ‘timmy’;

```
--  user input that uses SQL Injection
SET @name_bad = ”’timmy’  OR 1";
set @query_bad = CONCAT("SELECT * FROM customers WHERE
   username = ",@name_bad,';');

SELECT   @query_bad ;
```

SELECT * FROM customers WHERE username = ‘timmy' OR 1;

Benefits to a prepared statement

- A prepared statement protects the database against SQL injection, since you define the parts of the statement that can be replaced by a value
- Less overhead for parsing the statement each time it is executed. Statement is set up once (prepared) and executed with different values.
- The query plan is only determined once when the statement is prepared and is used for each execution.

Using a prepared statement

Use **PREPARE** to prepare a SQL statement

SYNTAX: PREPARE *statementname* from SQLStatement

Defines a name from the SQLStatement

Within SQLStatement, ? characters denote parameter markers to indicate where data values are to be bound to within the query when it is executed

Use **EXECUTE** to execute the command

SYNTAX:

EXECUTE SQLStatement [**USING** @***var_name*** [, @***var_name***] ...]

Parameter values can be supplied only by user variables, and the USING clause must name exactly as many variables as the number of parameter markers in the statement

Use **DEALLOCATE** to free resources associated with the statement

Example of a prepared statement

```
PREPARE stmt1 FROM 'SELECT
                    SQRT(POW(?,2) + POW(?,2)) AS hypotenuse';
SET @a = 3;
SET @b = 4;

EXECUTE stmt1 USING @a, @b;
```

Prepared statement example 2

```
use ap;
SET @c := 1;

SET @s :=
  "SELECT invoice_id FROM invoices WHERE vendor_id > ?;”

PREPARE stmt FROM @s;

EXECUTE stmt USING  @c;
```

Another approach to building queries

Dynamic SQL allows you to treat SQL as text and to build up the SQL statement by concatenating the strings together

Many people take this approach when creating the SQL statement within a host program, we do not recommend this approach but want to let you aware of the practice

We show you an example of dynamic SQL within a database procedure. In this example, the string containing the prepared statement must be a session variable as opposed to a local variable.

Dynamic SQL (1)

```
DELIMITER //

CREATE PROCEDURE select_invoices
(
  min_invoice_date_param   DATE,
  min_invoice_total_param  DECIMAL(9,2)
)
BEGIN
  DECLARE select_clause VARCHAR(200);
  DECLARE where_clause  VARCHAR(200);

  SET select_clause = "SELECT invoice_id, invoice_number,
                       invoice_date, invoice_total
                       FROM invoices ";
  SET where_clause =  "WHERE ";

  IF min_invoice_date_param IS NOT NULL THEN
    SET where_clause = CONCAT(where_clause,
       " invoice_date > '", min_invoice_date_param, "'");
  END IF;
```

Dynamic SQL (2)

```
IF min_invoice_total_param IS NOT NULL THEN
    IF where_clause != "WHERE " THEN
      SET where_clause = CONCAT(where_clause, "AND ");
    END IF;
    SET where_clause = CONCAT(where_clause,
       "invoice_total > ", min_invoice_total_param);
  END IF;

  IF where_clause = "WHERE " THEN
    SET @dynamic_sql = select_clause;
  ELSE
    SET @dynamic_sql = CONCAT(select_clause, where_clause);
  END IF;

  PREPARE select_invoices_statement
  FROM @dynamic_sql;

  EXECUTE select_invoices_statement;

  DEALLOCATE PREPARE select_invoices_statement;
END//
```

Summary

A prepared statement allows you to specify the structure of a SQL statement and identify what portions of the statement can be changed at execution time

It helps address the problem of SQL injection since the structure of the statement has already been specified to the database server before the statement is executed.