Bethuel Kilonzo - 191636

Report on Prepared Statements and SQL Injection Prevention in PHP

1. Running Queries Using Prepared Statements

Prepared statements (also known as parameterized queries) allow developers to separate SQL code from user data,
ensuring data is safely handled. They are supported both in MySQLi and PDO in PHP.

Example using MySQLi:

```
$conn = new mysqli($host, $user, $pass, $db);
$stmt = $conn->prepare("SELECT * FROM users WHERE email = ? AND status = ?");
$stmt->bind_param("si", $email, $status);
$email = $_POST['email'];
$status = 1;
$stmt->execute();
$result = $stmt->get_result();
while ($row = $result->fetch_assoc()) {
   // process $row
}
$stmt->close();
$conn->close();
```

Example using PDO:

```
$pdo = new PDO("mysql:host=$host;dbname=$db;charset=utf8mb4", $user, $pass,
          [ PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION ]);
$pdo->setAttribute(PDO::ATTR_EMULATE_PREPARES, false);

$sql = "INSERT INTO orders (user_id, total, created_at) VALUES (:uid, :total, NOW())";
$stmt = $pdo->prepare($sql);
$stmt->bindParam(':uid', $userId, PDO::PARAM_INT);
$stmt->bindParam(':total', $orderTotal);
$userId = 123;
$orderTotal = 49.99;
```

```
$stmt->execute();
```

Prepared statements improve both security and performance by pre-compiling the SQL query. The workflow involves preparing,
binding parameters, executing, and retrieving results.

2. How Prepared Statements Thwart SQL Injection

SQL Injection occurs when untrusted input is merged into a SQL query, allowing an attacker to modify its logic.
Prepared statements mitigate this by separating query structure from data. The placeholders ensure the query structure is fixed,
and user data cannot change it.

Key benefits:
- Separation of code and data.
- Automatic quoting and escaping by the database driver.
- Elimination of risky string concatenation.
- Prevention of malicious input from altering SQL logic.

Prepared statements protect against injection only in value contexts, not when dynamic SQL fragments (like table names)
are built from user input. Developers must still validate input and disable PDO emulated prepares to use native ones.

3. Running Transactions Using PDO

Transactions ensure multiple operations execute atomically — either all succeed or none do.
PDO provides transaction control methods: beginTransaction(), commit(), and rollBack().

Example:

```
try {
   $pdo->beginTransaction();
```

```
    $stmt1 = $pdo->prepare("UPDATE accounts SET balance = balance - :amt WHERE id =
:id");
    $stmt1->execute([':amt' => $amount, ':id' => $fromId]);

    $stmt2 = $pdo->prepare("UPDATE accounts SET balance = balance + :amt WHERE id =
:id");
    $stmt2->execute([':amt' => $amount, ':id' => $toId]);

    $pdo->commit();
} catch (Exception $e) {
    $pdo->rollBack();
    throw $e;
}
```

Transactions maintain data integrity and consistency, ensuring that in case of an error, all changes are reverted.
PDO automatically rolls back uncommitted transactions when the connection closes.

4. Moving from Development to Production

Transitioning a PHP application to production requires configuration and security adjustments:

- Disable error display and enable error logging (display_errors=Off).
- Use `$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION)` for controlled error handling.
- Disable PDO emulated prepares (`PDO::ATTR_EMULATE_PREPARES = false`).
- Specify UTF-8 (utf8mb4) charset in connections to prevent encoding attacks.
- Use transactional engines like InnoDB for MySQL.
- Use environment variables for database credentials and limit database user privileges.
- Enable slow query logs and security monitoring.

Ensure all prepared statements are used consistently, avoid dynamic SQL with user input, and wrap critical multi-step operations in transactions.
Proper rollback mechanisms, exception handling, and logging should be added before going live.

Conclusion

Prepared statements are a critical defense mechanism against SQL injection in PHP. They maintain a clear separation
between SQL logic and data, eliminating risks from untrusted input. Combined with transactions, proper configurations,
and secure deployment practices, they enhance application reliability and data security.

References (APA)

- OWASP. (n.d.). SQL Injection Prevention Cheat Sheet. https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html
- WebsiteBeaver. (n.d.). Prepared Statements in PHP MySQLi to Prevent SQL Injection. https://websitebeaver.com/prepared-statements-in-php-mysqli-to-prevent-sql-injection
- WebsiteBeaver. (n.d.). PHP PDO Prepared Statements Tutorial to Prevent SQL Injection. https://websitebeaver.com/php-pdo-prepared-statements-to-prevent-sql-injection
- PHP Manual. (n.d.). PDO Transactions. https://www.php.net/manual/en/pdo.transactions.php
- SecurityJourney. (n.d.). How to Prevent SQL Injection Vulnerabilities. https://www.securityjourney.com/post/how-to-prevent-sql-injection-vulnerabilities-how-prepared-statements-work
- Medium. (n.d.). How to Use Parameterized Queries or Prepared Statements in PHP. https://medium.com/@rashid.khitilov/how-to-use-parameterized-queries-or-prepared-statements-in-php-e20d9790bfd8