SQL injection: parameterized queries and input validation.

Parameterized Queries:

Parameterized queries are a method of using parameter placeholders in SQL queries. These placeholders are then replaced with actual parameter values. This technique helps to separate SQL code from user input, preventing malicious input from being treated as part of the SQL statement.

Example in Python using SQLite:

```python
import sqlite3

# Unsafe way (vulnerable to SQL injection)
def unsafe_query(username):
    query = "SELECT * FROM users WHERE username = '" + username + "';"
    return query

# Safe way (using parameterized query)
def safe_query(username):
    query = "SELECT * FROM users WHERE username = ?;"
    conn = sqlite3.connect("example.db")
    cursor = conn.cursor()
    cursor.execute(query, (username,))
    result = cursor.fetchall()
    conn.close()
    return result
```

In the unsafe example, the username is directly concatenated into the SQL query, making it susceptible to SQL injection. In the safe example, a parameterized query is used, and the actual parameter value is passed separately, reducing the risk of injection.

Example in PHP using MySQLi:

```php
// Unsafe way (vulnerable to SQL injection)
function unsafe_query($username) {
    $query = "SELECT * FROM users WHERE username = '" . $username . "';";
    return $query;
}

// Safe way (using parameterized query)
function safe_query($username) {
    $query = "SELECT * FROM users WHERE username = ?;";
    $conn = new mysqli("localhost", "username", "password", "database");
    $stmt = $conn->prepare($query);
    $stmt->bind_param("s", $username);
    $stmt->execute();
    $result = $stmt->get_result()->fetch_all(MYSQLI_ASSOC);
```

```
    $stmt->close();
    $conn->close();
    return $result;
}
```

In the PHP examples, the unsafe query directly includes user input, while the safe query uses parameterized queries with the `bind_param` function to bind variables securely.

Input Validation:

Input validation involves checking user input to ensure it meets certain criteria (e.g., length, type, format) before processing it. This helps to filter out malicious input and prevent it from being used in SQL injection attacks.

Example in JavaScript (Node.js) using Express and Joi:

```javascript
const express = require('express');
const Joi = require('joi');

const app = express();
app.use(express.json());

// Unsafe endpoint (vulnerable to SQL injection)
app.post('/unsafe', (req, res) => {
    const username = req.body.username;
    const query = `SELECT * FROM users WHERE username = '${username}'`;
    // Execute the query...
});

// Safe endpoint (using input validation)
app.post('/safe', (req, res) => {
    const schema = Joi.object({
        username: Joi.string().alphanum().min(3).max(30).required(),
    });

    const { error, value } = schema.validate(req.body);

    if (error) {
        return res.status(400).send(error.details[0].message);
    }

    const username = value.username;
    const query = 'SELECT * FROM users WHERE username = ?';
    // Execute the query...
});

const PORT = 3000;
app.listen(PORT, () => {
    console.log(`Server is running on port ${PORT}`);
```

});
```

In the unsafe endpoint, the username is directly used in the SQL query without validation. In the safe endpoint, Joi is used for input validation to ensure that the username meets certain criteria before processing the query.

**Example in C# using ASP.NET Core and Data Annotations:**

```csharp
public class UserController : Controller
{
    private readonly MyDbContext _context;

    public UserController(MyDbContext context)
    {
        _context = context;
    }

    // Unsafe action method (vulnerable to SQL injection)
    public IActionResult Unsafe(string username)
    {
        var query = $"SELECT * FROM Users WHERE Username = '{username}'";
        // Execute the query...
    }

    // Safe action method (using input validation)
    public IActionResult Safe([StringLength(30, MinimumLength = 3)] string username)
    {
        if (ModelState.IsValid)
        {
            var query = "SELECT * FROM Users WHERE Username = {0}";
            // Execute the query...
        }
        else
        {
            return BadRequest(ModelState);
        }
    }
}
```

Certainly! Here are more examples using Python for SQL injection prevention with parameterized queries and input validation:

More Examples in Python:

Parameterized Queries:

1. **Using MySQL Connector:**
```

```python
import mysql.connector

# Unsafe way (vulnerable to SQL injection)
def unsafe_query(user_id):
    query = "SELECT * FROM users WHERE id = " + str(user_id) + ";"
    return query

# Safe way (using parameterized query)
def safe_query(user_id):
    query = "SELECT * FROM users WHERE id = %s;"
    conn = mysql.connector.connect(host="localhost", user="username", password="password",
database="database")
    cursor = conn.cursor()
    cursor.execute(query, (user_id,))
    result = cursor.fetchall()
    conn.close()
    return result
```

In the unsafe example, the `user_id` is directly concatenated into the SQL query. In the safe example, a parameterized query is used with the `%s` placeholder.

2. **Using SQLite with `sqlite3` module:**

```python
import sqlite3

# Unsafe way (vulnerable to SQL injection)
def unsafe_query(user_email):
    query = "SELECT * FROM users WHERE email = '" + user_email + "';"
    return query

# Safe way (using parameterized query)
def safe_query(user_email):
    query = "SELECT * FROM users WHERE email = ?;"
    conn = sqlite3.connect("example.db")
    cursor = conn.cursor()
    cursor.execute(query, (user_email,))
    result = cursor.fetchall()
    conn.close()
    return result
```

In the unsafe example, the `user_email` is directly concatenated into the SQL query. In the safe example, a parameterized query is used with the `?` placeholder.

Input Validation:

1. **Using `validate_email` library:**

```python
from validate_email_address import validate_email

# Unsafe function (vulnerable to SQL injection)
def unsafe_function(email):
    query = f"SELECT * FROM users WHERE email = '{email}';"
    # Execute the query...

# Safe function (using input validation)
def safe_function(email):
    if validate_email(email):
        query = "SELECT * FROM users WHERE email = ?;"
        # Execute the query...
    else:
        print("Invalid email address.")
```

In the safe function, the `validate_email` library is used to check if the email address is valid before using it in the SQL query.

2. **Using Regular Expressions for Username Validation:**

```python
import re

# Unsafe function (vulnerable to SQL injection)
def unsafe_function(username):
    query = f"SELECT * FROM users WHERE username = '{username}';"
    # Execute the query...

# Safe function (using input validation)
def safe_function(username):
    if re.match("^[a-zA-Z0-9_]*$", username):
        query = "SELECT * FROM users WHERE username = ?;"
        # Execute the query...
    else:
        print("Invalid username.")
```

In the safe function, a regular expression is used to ensure that the username contains only alphanumeric characters and underscores before using it in the SQL query.