SQL Injection em Formulários: Exemplo Prático com PHP e MySQL

1. Contextualização

Para ilustrar a vulnerabilidade de SQL Injection, vamos criar um formulário de autenticação de usuário utilizando PHP, MySQL e HTML. Vamos simular um cenário onde um criminoso pode explorar essa falha para acessar dados de um sistema.

2. Configuração do Ambiente

Certifique-se de que o XAMPP está instalado e o servidor Apache e MySQL estão rodando. Crie um banco de dados no MySQL chamado `test_db` e uma tabela `usuarios` com os seguintes campos:

```sql
CREATE DATABASE test_db;
USE test_db;

CREATE TABLE usuarios (
    id INT AUTO_INCREMENT PRIMARY KEY,
    usuario VARCHAR(50) NOT NULL,
    senha VARCHAR(255) NOT NULL
);

INSERT INTO usuarios (usuario, senha) VALUES ('josemoura', '010203');
```

3. Criando o Formulário HTML

Vamos criar um formulário simples para a autenticação do usuário:

```html
<!DOCTYPE html>
<html lang="pt-BR">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Login</title>
</head>
<body>
    <h2>Login</h2>
    <form action="login.php" method="POST">
        <label for="usuario">Usuário:</label>
        <input type="text" id="usuario" name="usuario"><br><br>

        <label for="senha">Senha:</label>
        <input type="password" id="senha" name="senha"><br><br>

        <input type="submit" value="CONTINUAR">
    </form>
</body>
</html>
```

```

**4. Criando o Script PHP Vulnerável**

Vamos criar o script PHP que processa o login, vulnerável a SQL Injection:

```php
<?php
// login.php

$servername = "localhost";
$username = "root"; // Use o usuário root ou crie outro usuário para o MySQL
$password = ""; // Normalmente, a senha é vazia no XAMPP por padrão
$dbname = "test_db";

// Conectar ao banco de dados
$conn = new mysqli($servername, $username, $password, $dbname);

// Verificar conexão
if ($conn->connect_error) {
    die("Conexão falhou: " . $conn->connect_error);
}

// Obter dados do formulário
$usuario = $_POST['usuario'];
$senha = $_POST['senha'];

// Construir a query vulnerável
$sql = "SELECT * FROM usuarios WHERE usuario = '$usuario' AND senha = '$senha'";

$result = $conn->query($sql);

if ($result->num_rows > 0) {
    echo "Login realizado com sucesso!";
} else {
    echo "Usuário ou senha inválidos.";
}

$conn->close();
?>
```

5. Testando a Vulnerabilidade

Agora, vamos testar o login normal e a tentativa de SQL Injection:

1. Login Normal: Preencha o campo "Usuário" com `josemoura` e "Senha" com `010203`. Ao clicar em “CONTINUAR”, você verá a mensagem "Login realizado com sucesso!".

2. Tentativa de SQL Injection:
```

No campo "Usuário", insira o seguinte texto:

```
qualquercoisa' OR 1=1 #
```

Deixe o campo "Senha" em branco e clique em “CONTINUAR”.

Explicação:

- O campo `usuário` agora contém a expressão `'qualquercoisa' OR 1=1 #`.
- A consulta SQL resultante será:

```sql
SELECT * FROM usuarios WHERE usuario = 'qualquercoisa' OR 1=1; # AND senha = ''
```

- A expressão `OR 1=1` é sempre verdadeira, e o `#` faz com que o resto da consulta seja ignorado. Isso retorna todos os usuários da tabela, permitindo que o atacante faça login sem a senha correta.

6. Como Prevenir SQL Injection

A forma mais segura de evitar SQL Injection é utilizar **Prepared Statements** com **bind parameters**. Veja como ficaria o código utilizando essa técnica:

```php
<?php
// login.php (Versão Segura)

$servername = "localhost";
$username = "root";
$password = "";
$dbname = "test_db";

$conn = new mysqli($servername, $username, $password, $dbname);

if ($conn->connect_error) {
    die("Conexão falhou: " . $conn->connect_error);
}

$usuario = $_POST['usuario'];
$senha = $_POST['senha'];

// Usando Prepared Statements para evitar SQL Injection
$stmt = $conn->prepare("SELECT * FROM usuarios WHERE usuario = ? AND senha = ?");
$stmt->bind_param("ss", $usuario, $senha);
$stmt->execute();

$result = $stmt->get_result();

if ($result->num_rows > 0) {
```

```
    echo "Login realizado com sucesso!";
} else {
    echo "Usuário ou senha inválidos.";
}

$stmt->close();
$conn->close();
?>
```

7. Conclusão

Este exemplo prático demonstra como a vulnerabilidade de SQL Injection pode ser explorada e como preveni-la utilizando técnicas seguras de codificação. Evitar SQL Injection é crucial para proteger suas aplicações contra ataques maliciosos.