

# Injection Attacks

by Ralfe Poisson



# Overview

- **What** are Injection Attacks?
- **Types** of Injection Attacks
- **Prevention** of Injection Attacks
- **Demonstration** of some Injection Attacks

# Code Injection Exploitation

**Code Injection** is the exploitation of a software vulnerability that is caused by **processing invalid data**.

# Types of Injection Attacks

1) SQL Injection

2) Script Injection

3) Dynamic Evaluation Vulnerabilities

4) Shell Injection

# 1. SQL Injection

**SQL injection is a technique employed to manipulate a legitimate database query in order to return falsified data.**

# Example

## Standard login form

Username :

Password :

To authenticate against this form, a programmer might do something like :

```
SELECT * FROM `users` WHERE `user` = 'someusername'  
AND `pass` = 'somepassword'
```

# But .....

What would happen if somepassword or  
someusername happen to be something other than  
a

username and password which we were expecting?

What if, for instance, they happen to be SQL  
commands?



# The Beginning of the End

What if we enter the following into the form:

Username :   
Password :

Username : **anything**  
Password : **' OR 1=1 #**

The resulting SQL would be:

```
SELECT * FROM `users`  
WHERE `user` = 'anything'  
AND `pass` = " OR 1=1 #";
```

# So what would that do?



**With the resulting SQL ...**

We are retrieving all the information from the users table where  $1=1$ , in other words, **EVERYTHING**.

We would effectively become the first user in the table.

That is quite scary.

# Admin Access ..... not good

Now, what would happen if the following login details were used?

Username :   
Password :

Username : **admin' #**  
Password : **\_**

The resulting SQL would be:

```
SELECT * FROM `users`  
WHERE `user` = 'admin' #';
```



# It Gets Much Much Worse

## Retrieving Plaintext Passwords

**Step 1** : Find the table with the login details

Username : **admin**

Password : ' **UNION**

```
SELECT CONVERT (table_name USING latin1)  
FROM INFORMATION_SCHEMA.TABLES  
WHERE table_name LIKE 'u%'  
AND NOT table_name = 'USER_PRIVILEGES
```



# It Gets Much Much Worse .....

The resulting SQL is :

```
SELECT * FROM `users`  
WHERE `user` = 'admin' AND `password` = "  
UNION  
SELECT CONVERT (table_name USING latin1)  
FROM INFORMATION_SCHEMA.TABLES  
WHERE table_name LIKE 'u%' AND NOT table_name =  
'USER_PRIVILEGES'.
```

From the output we can determine the table with the login data.

# It Gets Much Much Worse .....

## Step 2 : Get the Password

Username : **admin'**

Password : **' UNION SELECT  
CONCAT(`user`, '=', `pass`)  
FROM `users`  
WHERE `user` = 'admin**



# It Gets Much Much Worse .....

The Resulting SQL is :

```
SELECT * FROM `users`  
WHERE `user` = 'admin' AND `password` = "  
UNION  
SELECT CONCAT (`user`, '=', `pass`)  
FROM `users`  
WHERE `user` = 'admin'.
```

We then will see something like this on the landing page:

**" Welcome admin=AdminPassword "**



# So How Do We Prevent This?

## # 1 : Escape Received Strings

Properly escape the strings we receive from the users. Alternatively, we could strip out characters we know we shouldn't be receiving, such as quotation marks, semi-colons etc...

# So How Do We Prevent This?

## # 2 : Password Hash Codes

Store hash codes only for passwords. Thus, the plaintext password is never used or stored or compared within an SQL query. In the script (perhaps the PHP script), you would generate a hash of the password entered by the user, and compare the resulting hash to the hash stored in the database. If the hash codes match, then authentication has occurred, if not, then the passwords do not match, and the user should not have access to further information. This will negate the possibility of the above hack for retrieving passwords.

# So How Do We Prevent This?

## # 3 : Database Specific User Priviledges

From the webapp, only access the database with a user with database-specific priviledges. You do not want to be using the root user account to be accessing the database. If you are foolish enough to do this, you are opening yourself up for someone to either wipe out your entire database server, or retrieve every single scrap of data on your SQL server....  
NOT GOOD.

# So How Do We Prevent This?

## **#4 : Turn on Magic-Quotes**

For system administrators, simply by turning on the `magic_quotes` flag in the `php.ini` file will automatically escape any suspicious quotation or apostrophe marks.

## 2. Script Injection Attacks

**Various types of code injection attacks which allow an attacker to supply code to the server side scripting engine.**

# Script Injection Examples

## Cross Site Scripting (XSS)

Occurs when HTML code and client-side scripts are able to be inserted into web applications and viewed by other users.

### Types of XSS

- **DOM-based**

- Type 0 XSS vulnerabilities; local XSS; exploits the DOM.

- **Non-Persistent**

- Type 1 XSS hole; reflected vulnerability; very common. Occurs when malicious HTML code is inserted into a form, whose data is redisplayed to the user; Social Engineering.

- **Persistent**

- Type 2 XSS vulnerability; stored/second-order vulnerability. A user's data is displayed to other users.

# Script Injection Examples

## Unvalidated File Upload Vulnerabilities

When a user is able to upload a file which is not validated by the server, a potential exists to upload malicious code to the server and execute it remotely.

### Example

Uploading a php script when asked to upload a profile picture. To execute this malicious php script, one simply needs to point the browser to the location of the php script in the publically accessible location where the other profile pictures are stored.

# 3. Dynamic Evaluation Attacks

## Dynamic Code Evaluation

Arbitrary code is inserted in place of standard input, resulting in that code being executed as part of the application.

### Example

Php's **eval()**; command will execute PHP code passed to it as a parameter. Also, watch out for dynamic function and variable evaluation.

**Warning : Do not think you are safe using preg\_replace or escaping and sanitizing input. There are ways of getting past this, such as using complex syntax (curly).**

# 4. Shell Injection Attacks

This class of attacks exploits applications which use input to formulate commands that are executed by the OS.

The severity of the attack depends on the access level of the user account underwhich the process is running which executes the command.

This can usually be sorted out through use of sanity checks, but it is best to avoid directly inserting user input into shell commands.

# Shell Injection Examples

- Server Side Scripts (PHP, Perl, etc..)
- KDE and Gnome launchers
- Buffer Overflow attacks



# Preventing Injection Attacks

- Sanitize all user input
- Validate user input
- Avoid directly using user input for shell commands
- Avoid dynamic evaluation where possible
  - Dynamic Code Evaluation
  - Dynamic Variable Evaluation
  - Dynamic Function Evaluation
- Implement and review system access controls