**SQL Injection**

Many web developers are unaware of how SQL queries can be tampered with, and assume that an SQL query is a trusted command. It means that SQL queries are able to circumvent access controls, thereby bypassing standard authentication and authorization checks, and sometimes SQL queries even may allow access to host operating system level commands.

Direct SQL Command Injection is a technique where an attacker creates or alters existing SQL commands to expose hidden data, or to override valuable ones, or even to execute dangerous system level commands on the database host. This is accomplished by the application taking user input and combining it with static parameters to build an SQL query. The following examples are based on true stories, unfortunately.

Owing to the lack of input validation and connecting to the database on behalf of a superuser or the one who can create users, the attacker may create a superuser in your database.

**Example #1 Splitting the result set into pages ... and making superusers (PostgreSQL)**

<?php  
  
$offset = $argv[0]; // beware, no input validation!  
$query  = "SELECT id, name FROM products ORDER BY name LIMIT 20 OFFSET $offset;";  
$result = pg\_query($conn, $query);  
  
?>

SQL Injection is an attack on the database via the addition of malicious code, mostly in the form of SQL queries, passed into a string that reaches past the application on to the database and causes it to execute against the database. In most cases the hacker is able to obtain unauthorized information off of the database.

This article is intended to discuss some of the ways that MySQL databases can be exposed via SQL Injection techniques.

Simply put, the end goals of a SQL injection attack is to gain private (maybe confidential) data, perform a create or change on data which is not meant to be changed, or altogether purge the data, data object or dataset. Just a couple of months back, i.e. March 27th 2011 to be precise, Romanian Hackers by the name of “TinKode” and “Ne0h” attacked MySQL.com and Sun.com. They did this with a SQL injection attack, to gather table names, column names and email addresses stored in one of the tables.

“TinKode has been busy in recent days going after MySQL databases.” It is worth noting that the article does not blame MySQL, the open-source DBMS, for the vulnerability but rather the Website code.

**SQL Injection Examples**

Say you have a table called “STUDENT\_MARKS” which has columns - roll\_number, last\_name, first\_name, mark, and your php code is to output the marks given the student rollnumber. Each student will know their own unique10 digit rollnumber. Your code looks something like this and has no check on the inputs given to the application or website:

// connection made against mysql database

// given input of say…

$rollnumber= “4’ or ‘1’=’1”;

// given query code of say…

mysql\_query(“SELECT \* FROM STUDENT\_MARKS WHERE rollnumber=’{$rollnumber}’”);

mysql\_connect("$host", "$username", "$password")or die("cannot connect");

mysql\_select\_db("$db\_name")or die("cannot select DB");

// Define $myusername and $mypassword

$myusername=$\_POST['username'];

$mypassword=$\_POST['password'];

// To protect MySQL injection (more detail about MySQL injection)

$myusername = stripslashes($myusername);

$mypassword = stripslashes($mypassword);

$myusername = mysql\_real\_escape\_string($myusername);

$mypassword = mysql\_real\_escape\_string($mypassword);

$sql="SELECT \* FROM $tbl\_name WHERE username='$myusername' and password='$mypassword'";

$result=mysql\_query($sql);

// set variables

$\_SESSION['office\_name'] = mysql\_query("SELECT office FROM $tbl\_name WHERE username='$myusername' and password='$mypassword'");

// Mysql\_num\_row is counting table row

$count=mysql\_num\_rows($result);

// If result matched $myusername and $mypassword, table row must be 1 row

if($count==1){

// Register $myusername, $mypassword and redirect to file "login\_success.php"

session\_register("username");

session\_register("password");

header("location:myecl.php");

}

else {

header( 'Location: http://www.galactek.com/support/offmaint.html' );

}

?>

# Encryption:

Pad the plaintext using a sane padding strategy (e.g. PKCS#7)

Encrypt using AES, BlowFish, Serpent, TwoFish, or Triple-DES in CBC or CTR mode. You may also use GCM (which removes the need for a separate MAC). Additionally, ChaCha20 and Salsa20/8 (provided by [libsodium](https://github.com/jedisct1/libsodium-php)) are stream ciphers and do not need special modes.

Unless you chose GCM above, you should authenticate the ciphertext with HMAC-SHA-256 (or, for the stream ciphers, Poly1305 -- most libsodium APIs do this for you). **The MAC should cover the IV as well as the ciphertext!**

# Decryption:

Unless Poly1305 or GCM is used, recalculate the MAC of the ciphertext and compare it with the MAC that was sent using [hash\_equals()](https://php.net/manual/en/function.hash-equals.php). If it fails, abort.

Decrypt the message.

Remove the padding.