



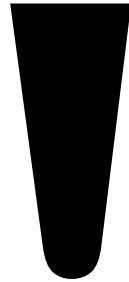
Injection Flaws





The OWASP Foundation

<http://www.owasp.org>





Anatomy of SQL Injection Attack

```
sql = "SELECT * FROM user_table WHERE username = '' &  
Request("username") & '' AND password = '' & Request  
("password") & """
```

What the developer intended:

username = john

password = password

SQL Query:

```
SELECT * FROM user_table WHERE username = 'john' AND  
password = 'password'
```

Anatomy of SQL Injection Attack



The OWASP Foundation

<http://www.owasp.org>

```
sql = "SELECT * FROM user_table WHERE username = '' &  
Request("username") & "' AND password = ' " & Request("password") &  
" ' "
```



(This is DYNAMIC SQL and Untrusted Input)

What the developer did not intend is parameter values like:

username = john

password = blah' or '1'='1 --

SQL Query:

```
SELECT * FROM user_table WHERE username = 'john' AND password =  
'blah' or '1'='1' --
```

or '1' = '1' causes all rows in the users table to be returned!



Example Attacks

```
SELECT first_name, last_name FROM users WHERE  
user_id = " UNION ALL SELECT  
load_file('C:\\app\\htdocs\\webapp\\.htaccess'), '1'
```

```
SELECT first_name, last_name FROM users WHERE  
user_id =" UNION SELECT ", '<?php  
system($_GET["cmd"]); ?>' INTO OUTFILE  
'C:\\app\\htdocs\\webapp\\exploit.php';#
```

Goto <http://bank.com/webapp/exploit.php?cmd=dir>



- String building can be done when calling stored procedures as well

```
sql = "GetCustInfo @LastName=" +  
request.getParameter("LastName");
```

- Stored Procedure Code

```
CREATE PROCEDURE GetCustInfo (@LastName VARCHAR(100))  
AS
```

```
exec('SELECT * FROM CUSTOMER WHERE LNAME=''' + @LastName + ''') (Wrapped Dynamic SQL)  
GO
```

What's the issue here.....

If **blah' OR '1'='1** is passed in as the LastName value, the entire table will be returned

- Remember Stored procedures need to be implemented safely. 'Implemented safely' means the stored procedure does not include any unsafe dynamic SQL generation.



Rails:

ActiveRecord/Database Security

Rails is designed with minimal SQL Injection problems.

It is not recommended to use user data in a database query in the following manner:

```
Project.where("name = '#{params[:name]}')
```

By entering a parameter with a value such as

```
' OR 1 --
```

Will result in:

```
SELECT * FROM projects WHERE name = " OR 1 --'
```



Active Record

Other Injectable examples:

Rails 2.X example:

```
@projects = Project.find(:all, :conditions => "name  
like #{params[:name]}")
```

Rails 3.X example:

```
name = params[:name]  
@projects = Project.where("name like ' " + name + "  
' ");
```




Active Record

Countermeasure

Ruby on Rails has a built-in filter for special SQL characters, which will escape ' , " , NULL character and line breaks.

Using `Model.find(id)` or `Model.find_by_something(something)` automatically applies this countermeasure.

`Model.where("login = ? AND password = ?", entered_user_name, entered_password).first`

The "?" characters are placeholders for the parameters which are **parameterised** and escaped automatically.

Important:

Many query methods and options in ActiveRecord which do not sanitize raw SQL arguments and are not intended to be called with unsafe user input.

A list of them can be found here and such methods should be used with caution.



Query Parameterization (PHP)

```
$stmt = $dbh->prepare("update users set  
email=:new_email where id=:user_id");
```

```
$stmt->bindParam(':new_email', $email);
```

```
$stmt->bindParam(':user_id', $id);
```



Query Parameterization (.NET)

```
SqlConnection objConnection = new
SqlConnection(_ConnectionString);
objConnection.Open();
SqlCommand objCommand = new SqlCommand(
    "SELECT * FROM User WHERE Name = @Name AND
    Password = @Password", objConnection);
objCommand.Parameters.Add("@Name",
    NameTextBox.Text);
objCommand.Parameters.Add("@Password",
    PassTextBox.Text);
SqlDataReader objReader =
objCommand.ExecuteReader();
```



Query Parameterization (Java)

```
String newName = request.getParameter("newName") ;  
String id = request.getParameter("id") ;
```

```
//SQL
```

```
PreparedStatement pstmt = con.prepareStatement("UPDATE  
EMPLOYEES SET NAME = ? WHERE ID = ?");  
pstmt.setString(1, newName);  
pstmt.setString(2, id);
```

```
//HQL
```

```
Query safeHQLQuery = session.createQuery("from  
Employees where id=:empId");  
safeHQLQuery.setParameter("empId", id);
```



Query Parameterization (Cold Fusion)

```
<cfquery name="getFirst"  
dataSource="cfsnippets">  
    SELECT * FROM #strDatabasePrefix#_courses  
WHERE intCourseID = <cfqueryparam  
value=#intCourseID# CFSQLType="CF_SQL_INTEGER">  
</cfquery>
```



Query Parameterization (PERL)

```
my $sql = "INSERT INTO foo (bar, baz) VALUES ( ?, ? )";  
my $sth = $dbh->prepare( $sql );  
$sth->execute( $bar, $baz );
```

Command Injection



The OWASP Foundation

<http://www.owasp.org>

Web applications may use input parameters as arguments for OS scripts or executables

Almost every application platform provides a mechanism to execute local operating system commands from application code

- Perl: `system()`, `exec()`, `backquotes(` `)`
- C/C++: `system()`, `popen()`, `backquotes(` `)`
- ASP: `wscript.shell`
- Java: `getRuntime.exec`
- MS-SQL Server: `master..xp_cmdshell`
- PHP : `include()` `require()`, `eval()` ,`shell_exec`

Most operating systems support multiple commands to be executed from the same command line. Multiple commands are typically separated with the pipe “|” or ampersand “&” characters



Where can I learn more?

LDAP Injection

- https://www.owasp.org/index.php/LDAP_injection
- https://www.owasp.org/index.php/Testing_for_LDAP_Injection

SQL Injection

- https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet
- https://www.owasp.org/index.php/Query_Parameterization?

Command Injection

- https://www.owasp.org/index.php/Command_Injection