SQL Injection Optimization and Obfuscation Techniques

By Roberto Salgado

Introduction

SQL Injections are without question one of the most dangerous web vulnerabilities around. With all of our information stored in databases, almost every detail about our lives is at the mercy of a simple HTTP request. As a solution, many companies implement Web Application Firewalls and Intrusion Detection/Prevention Systems to try to protect themselves. Unfortunately, these countermeasures are not sufficient and can easily be circumvented. This is all possible due to optimization and obfuscation techniques which have been perfected over the last 15 years since the discovery of this lethal vulnerability.

Even though firewalls cannot not be relied on to prevent all attacks, some firewalls can be effective when used as a monitoring tool. It is not unheard of for an attacker to be detected and blocked during mid-attack, due to firewall triggers and an alert security team. Because of this, a SQL Injection that has been optimized and obfuscated has a much higher probability of being successful; it will extract the data faster and remain undetected for longer.

In this paper we will discuss and compare a variety of optimization methods which can be highly effective when exploiting Blind SQL Injections. We will also introduce SQL queries which can be used to dump the whole database with just one request, making it an extremely easy to quickly retrieve data while going unnoticed. Furthermore, we will be reviewing several obfuscation techniques which can make a SQL Injection unrecognizable to firewalls. When combined, these techniques create a deadly attack which can be devastating.

Optimization

Optimization is often one of the most overlooked, yet important aspects when exploiting Blind SQL Injections. Not only will an optimized SQL Injection produce faster results, it also reduces the network congestion and is a lower burden on the server which in turn provides a lower risk of being detected. The difference in speed some methods offer is astonishing and can cut the amount of requests and time it takes to successfully extract data from a database by over a third when compared to traditional methods. In order to use these methods, we must understand how they work and how effective each one is, but before we begin to analyze and compare them, it is important to review some preliminary concepts beforehand.

Computers can only understand numbers, for this reason ASCII (American Standard Code for Information Interchange) was created. An ASCII Code is a numerical representation of a letter or character. Any ASCII character can be represented in 1 byte or 8 bits, also known as an octet. When dealing with SQL Injections, we are usually not interested in the full ASCII range, as not all characters are valid or permitted in a database. It is for this reason that we will only focus on the ASCII range 32 – 126 which leaves us with a set of 94 characters. This range can be represented with only 7 bits which leaves the most significant bit as 0. This will come in handy when further optimizing the techniques since it allows us to shave off a request. The following table shows the ASCII range in its different forms:

Decimal	**Hexadecimal**	**Binary**
0	00	00000000
127	7F	01111111

Table 1: The MSB (most significant bit) is always off, allowing us to save an additional request.

Analysis of Methods

These methods are intended to be used with Blind SQL Injections when error showing is disabled, in which we generally have to retrieve 1 character at a time.

Bisection Method

The bisection method, also commonly known as the binary search algorithm in Computer Science, is a logarithmic algorithm and the most widely used method for extracting data through a SQL Injection; it is versatile, effective and straight forward to implement which makes it a popular choice among hackers and programmers. Using a finite list, the bisection method works by splitting the list in half and searching each half of the branch. It continues the process by splitting the half branch were the item was found and repeats until finished, this is why it is a dichotomic divide-and-conquer algorithm. When described in terms performance, it has the same worst case and average case scenario of log2(N), which leaves this method usually on the high end of its requests. When applied to SQL Injections additional requests are required to make this method work.

Regex Method

First introduced in a paper called “Blind Sql Injection with Regular Expressions Attack” by Simone Quatrini and Marco Rondini [2], this method is simply a variation of the bisection method which instead uses regular expressions. For MySQL, the technique makes use of the REGEXP function and for MSSQL, the LIKE statement which has similar functionality, but works slightly different. A small disadvantage to this method is the use of two different functions for each type of database. A suggested alternative is to use the BETWEEN function which exists in both MySQL and MSSQL; this would simplify the implementation by being able to use a single function for multiple databases.

REGEXP '^[a-z]'	True
REGEXP '^[a-n]'	True
REGEXP '^[a-g]'	False
REGEXP '^[h-n]'	True
REGEXP '^[h-l]'	False

Table 2: An example of the REGEX method in use.

Bitwise Methods

There are several bitwise methods which are all slight variations from each other. When analyzing these methods, in most cases they all perform worse than the bisection method, for that reason this paper will not go into great length describing them. These methods use bitwise operations such as shifting [4] or ANDing for example, but these tend to be redundant since a binary is treated as a string in both MySQL and MSSQL, so it is possible to use the SUBSTRING function or similar to compare each individual bit to 1 or 0. The amount of requests is consistent with these methods and will always be equivalent to the size of the binary string, usually 7 bits. An example of a bitwise shifting method which finds the letter “a” (01100001 in binary and 97 in decimal) can be observed in the following table:

Binary of “a”	**Bit to be shifted**	**Binary result**	**Decimal result**
01100001	>> 7	00000000	0
01100001	>> 6	00000001	1
01100001	>> 4	00000110	6
01100001	>> 3	00001100	12
01100001	>> 2	00011000	24
01100001	>> 1	00110000	48
01100001	>> 0	01100001	97

Table 3: Bitwise shifting method shown retrieving the character “a”.

Bin2Pos Method

This method is my own invention and it is essentially an optimized searching technique which can retrieve a character with a minimum of only 1 request and a maximum of 6 requests. This method depends on a finite list of characters where each character gets mapped to its position on the list; mapping the characters to their position is important because it provides a smaller number to be retrieved when compared to the decimal value of the character. The position is then converted to binary and retrieved starting with the first occurrence of an on bit in the binary sequence. By converting the position to binary, we are effectively reducing the set of characters to look for to two (1 or 0). Additionally since we start with the first occurrence of an on bit, we can save one request since we know it will be a “1”.

Since the size of the binary will depend on the position of the character in the list, the closer the character is to the beginning of the list, the less amount of requests are needed to retrieve it. For this reason, we can order the list by the most frequent letters in the alphabet to optimize it even further. An example of the implementation in MySQL which uses a parameter with two different values (e.g. id=1, id=2) can be seen in the figure below using a hexadecimal set.

- `IF((@a:=MID(`**`BIN`**`(`**`POSITION`**`(MID((SELECT password FROM users WHERE id=2 LIMIT 1),1,1)IN(CHAR(`**`48,49,50,51,52,53,54,55,56,57,65,66,67,68,69,70`**`))),1,1))!=space(0),2-@a,0/0)`

Because the standard set size of 94 (32 – 126) uses up to 7 bits, this means the maximum size possible of the character to retrieve will be 7. It takes an additional request to determine that we have reached the end of the binary sequence. So by avoiding the first request and having to do an additional request at the end, this leaves us with a total of 7 requests. However, since we know 7 bits is the maximum length possible to be retrieved, if we already made 6 requests, we don’t have to send an additional request to know when we have reached the end since we will have already reached the maximum length possible. This gives a worst case scenario of just 6 requests. The best case scenario is when the character is found in the first position of the list which would only require 1 request. The following illustration demonstrates this method using a set ordered alphabetically.

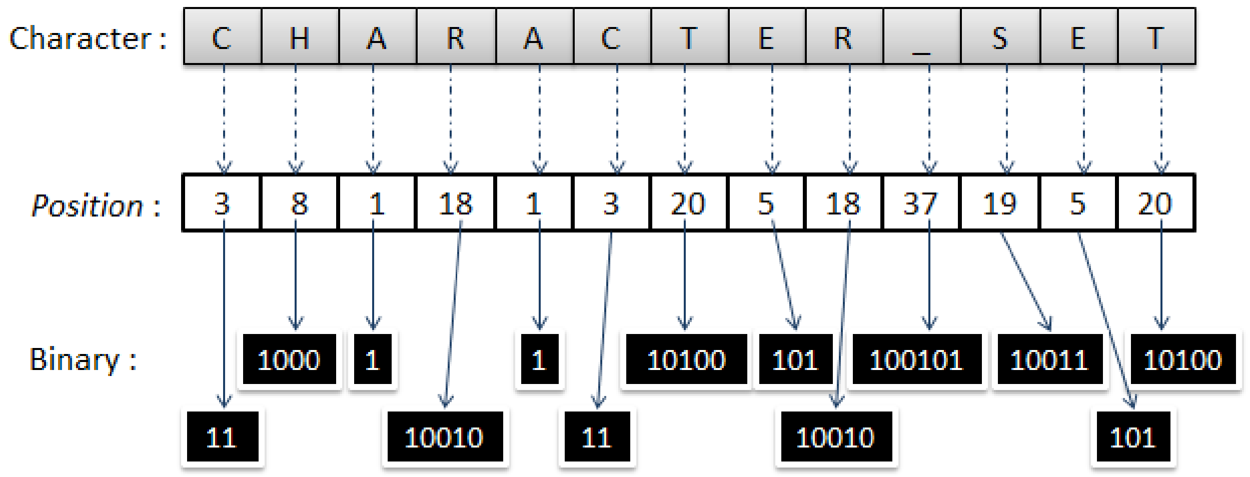

Figure 1: Only 2 requests are required to retrieve the character "C".

Optimizing Queries

Now that we have overviewed some of the known methods for extracting data through a Blind SQL Injection, we will look at some optimized queries which can greatly expedite data extraction from the database. Ionut Maroiu demonstrated a technique for MySQL which allows the retrieval of all databases, tables and columns with a single request [1]. A similar technique exist for PostgreSQL and Oracle which were demonstrated by Dmitriy Serebryannikov [5] and another one for MSSQL discovered by Daniel Kachakil [3]. The following table shows an example of each of the described queries:

MySQL	SELECT (@) FROM (SELECT(@:=0x00),(SELECT (@) FROM (information_schema.columns) WHERE (table_schema>=@) AND (@)IN (@:=CONCAT(@,0x0a,' [',table_schema,'] >',table_name,' > ',column_name))))x
MSSQL	SELECT table_name %2b ', ' FROM information_schema.tables FOR XML PATH('')
PostgreSQL	SELECT array_to_json(array_agg(tables))::text FROM (SELECT schemaname, relname FROM pg_stat_user_tables) AS tables LIMIT 1;
Oracle	SELECT xmlagg(xmlelement("user", login||':'||pass) ORDER BY login).getStringVal() FROM users;

Table 5: Different queries which retrieve multiple table & column entries with a single request.

There are also more devastating single request attacks [6] which are possible because of stacked queries in MSSQL or even MySQL when using PDO or the new and "improved" extension mysqli for PHP. For example, the following query for MSSQL will check to see if xp_cmdshell is loaded, if it is enabled, it checks to see if it is active and if it is active it proceeds to run a command of your choice.

- `' IF EXISTS (SELECT 1 FROM INFORMATION_SCHEMA.TABLES WHERE TABLE_NAME='TMP_DB') DROP TABLE TMP_DB DECLARE @a varchar(8000) IF EXISTS(SELECT * FROM dbo.sysobjects WHERE id = object_id (N'[dbo].[xp_cmdshell]') AND OBJECTPROPERTY (id, N'IsExtendedProc') = 1) BEGIN CREATE TABLE %23xp_cmdshell (name nvarchar(11), min int, max int, config_value int, run_value int) INSERT %23xp_cmdshell EXEC master..sp_configure 'xp_cmdshell' IF EXISTS (SELECT * FROM %23xp_cmdshell WHERE config_value=1)BEGIN CREATE TABLE %23Data (dir`

```
varchar(8000)) INSERT %23Data EXEC master..xp_cmdshell 'dir' SELECT
@a='' SELECT @a=Replace(@a%2B'<br></font><font
color="black">'%2Bdir,'<dir>','</font><font color="orange">') FROM
%23Data WHERE dir>@a DROP TABLE %23Data END ELSE SELECT @a='xp_cmdshell
not enabled' DROP TABLE %23xp_cmdshell END ELSE SELECT @a='xp_cmdshell
not found' SELECT @a AS tbl INTO TMP_DB--
```

For both a penetration tester and an attacker, testing for SQL Injections can become tedious. Some web applications can have endless amount of modules with infinite number of parameters. Furthermore, at least three separate tests are required for each possible type of SQL Injection: single, double and no quotations. With optimized queries, it is possible to reduce it all to one request.

Injection type	**Normal**	**Optimized**
No quotes	OR 1=1	OR 1#"OR"'OR''='"="'OR''='
Single quotes	‘ OR ‘’=’	OR 1#"OR"'OR''='"="'OR''='
Double quotes	“ OR “”=”	OR 1#"OR"'OR''='"="'OR''='

Table 6: The optimized injection allows us to test all three variations with a single request.

Another example using AND:

Injection type	**Normal**	**Optimized**
No quotes	AND 1=1	!=0--+"!="'!='
Single quotes	‘ AND ‘’=’	!=0--+"!="'!='
Double quotes	“ AND “”=”	!=0--+"!="'!='

Table 7: A variation of the optimized injection using AND logic.

To further illustrate this optimized query in action consider the following injection:

- `SELECT * FROM Users WHERE username=”Admin” and password = “1 OR 1#"OR"'OR''='"="'OR''='”`
- `SELECT * FROM Articles WHERE id = 1!=0--+"!="'!='`

Obfuscation

Obfuscation is one of the main tools for bypassing firewalls. By tricking the firewall into thinking our attacks are legitimate traffic, we can successfully deliver our payload without being detected. There are several methods that can be used to obfuscate our injections. Through the use of fuzzers, we can discover different possibilities that can be used for obfuscation. Although sometimes simplicity can be a better option and if all else fails, we can always resort to a variety of encodings.

Fuzzers

It is important for firewall developers to be fully aware of all the databases specifications and oddities. Some of these oddities are documented, others can only be found through fuzzing. One example of peculiarities discovered through the use of fuzzers are the permitted whitespace characters for each type of database. Each different RDBMS allows a variety of different whitespace characters instead of the usual 0x20. By switching the standard whitespace with other allowed whitespace characters, we can make our injection unrecognizable to certain firewalls allowing us to effectively bypass them.

RDBMS	Allowed whitespaces
SQLite 3	0A, 0D, 0C, 09, 20
MySQL 5	09, 0A, 0B, 0C, 0D, A0, 20
Oracle 11g	00, 09, 0A, 0B, 0C, 0D, 20
MSSQL 2008	01, 02, 03, 04, 05, 06, 07, 08, 09, 0A, 0B, 0C, 0D, 0E, 0F, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 1A, 1B, 1C, 1D, 1E, 1F, 20, 25

Table 8: Valid whitespaces allowed in different RDBMS.

While replacing whitespaces can be a useful trick, any sophisticated firewall will be capable of detecting most allowed whitespaces and subsequently will handle them appropriately. When firewalls become really good at deobfuscating injections, sometimes it can be best to take the opposite route and simplify the injection as much as possible.

Just like many programming languages give us several different ways of achieving the same result, so does SQL. In some cases using the simplest approach can be the most effective; if our SQL Injection looks like plain English, it can be very difficult for the firewall to differentiate between normal text and SQL syntax. A good example is the following use of the CASE statement which can be appended after a WHERE statement:

- `CASE WHEN BINARY TRUE THEN TRUE END IS NOT UNKNOWN HAVING TRUE FOR UPDATE`

Encodings

Special encodings are another tool in the box to use when bypassing firewalls. Most of these encodings will depend on how the application processes the data. The web application may see the data in one way, while the proxy, firewall or database might interpret the data differently. It is because of these discrepancies between layers that encodings may work to bypass the firewall.

URL encode

Anyone who has browsed the web has seen this encoding before. URL Encoding is used to transform “special” characters, so they can be sent over HTTP. Characters get transformed to their hexadecimal equivalent, prefixed by a percent sign.

Double URL encode

This is simply the process of applying URL encode two times on the characters. All that is required is re-encoding of the percent sign. This encoding is successful if the data is decoded twice after having passed through the firewall and before reaching the database.

Unicode encode

Unicode is an industry standard for representing over 110,000 symbols and characters for a wide range of languages. It can be represented by different character encodings such as UTF-8, UTF-16, UTF-32 and others. This encoding generally works against IIS servers or applications built in ASP or .NET.

Encoding type	Transformation
URL encode	%61
Double URL encode	%2561
Unicode encode	%u0061

Table 9: The following table shows each encoding for the character “a”.

UTF-8 encode

UTF-8 is a form of encoding each one of the 1,114,112 code points (0 through 10FFFF) in the Unicode character set. The "UTF" stands for Unicode Transformation Format while the "8" means it uses 8-bit blocks to represent a character. The number of blocks required to represent a character vary from 1 to 4. Only 1 byte or block is required to represent characters from ASCII range 0-127, having the leading bit as 0. However, any code point higher than 127 needs to be represented with multi-byte sequences where the leading bits of the first byte, up to the first 0, represent the total number of following bytes to complete the sequence. The following bits after the first 0 in the first byte form part of character. Each consecutive byte has '10' in the high-order position, however these two bits are redundant. The xx's in the chart [8] below represent the actual bits of data.

Bits of code point	**First code point**	**Last code point**	**Bytes in sequence**	**Byte 1**	**Byte 2**	**Byte 3**	**Byte 4**	**Byte 5**	**Byte 6**
7	U+0000	U+007F	1	0xxxxxxx					
11	U+0080	U+07FF	2	110xxxxx	10xxxxxx				
16	U+0800	U+FFFF	3	1110xxxx	10xxxxxx	10xxxxxx			
21	U+10000	U+1FFFFF	4	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx		
26	U+200000	U+3FFFFFF	5	111110xx	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx	
31	U+4000000	U+7FFFFFFF	6	1111110x	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx

Table 10: Here are the representations of different UTF-8 multi-byte encodings.

Because the first two high order bits after the first byte are not needed, applications may just read the six last bits allowing us to replace the first two most significant bits with 00, 01 or 11.

Byte Sequence	**Character "a" encoded**	**First two high order bits**
2 byte sequence	%c1%a1	10
2 byte sequence	%c1%21	00
2 byte sequence	%c1%61	01
2 byte sequence	%c1%e1	11
3 byte sequence	%e0%81%a1	10

Table 11: Several different UTF-8 representations of the character "a".

Nibble encode

A nibble is 4 bits, therefor there are sixteen (2**4) possible values, so a nibble corresponds to a single hexadecimal digit. Since every character from ASCII range 0-255 can be represented in 1 byte, we can URL encode the 4 most significant bits, the 4 least significant bits or both of them. This is commonly known as First Nibble, Second Nibble and Double Nibble encode

Type	**Transformation of %61**	**Result**
First Nibble	6 = %36	%%361
Second Nibble	1 = %31	%6%31
Double Nibble	6 = %36 1 = %31	%%36%31

Table 12: Examples of first, second and double nibble encodings.

Invalid Percent encode

Invalid percent encoding is specific to .NET/IIS applications. By adding the percent sign in between characters which are not valid hex, IIS will strip the character making the data valid. However any firewall that is unaware of this behavior will leave the percent sign which in turn mangles the SQL syntax making it undetectable to the firewall.

What the data looks like to the firewall:

- %**UNI**%**ON** %**SE**%**LE**%**CT** 1 %**FR**%**OM** %**D**%**U**%**A**%**L**

What the data looks like to IIS:

- UNION SELECT 1 FROM DUAL

Invalid Hex encode

Invalid Hex [7] isn't exactly an encoding, instead it is a way of abusing how some applications may handle the conversion of hexadecimal to decimal. The idea is to create invalid hex that results in the same decimal value as valid hex. Depending on how the application handles and transforms the data, it may see %2Ú as the same as %61.

Hex Character	**Conversion**	**Decimal**
%61	6 * 16 + 1	97
%2Ú	2 * 16 + 65	97

Table 13: Converting the invalid hex results in the same decimal value as valid hex when converted.

Decimal	**Valid Hex**	**Invalid Hex**
10	0A	0A
11	0B	0B
12	0C	0C
13	0D	0D
14	0E	0E
15	0F	0F
16	**10**	**0G**
17	**11**	**0H**

Table 14: Invalid hex uses a full alphabet where valid hex only uses A-F.

Conclusion

As a result of the advances in optimization and obfuscation, firewall developers and maintainers need to be more aware than ever when keeping their firewall's core rule set up to date. Because detecting all possible variations of SQL Injections is virtually an impossible task, firewalls can never be fully trusted as an adequate security measure to stop a SQL Injection attack and should only be relied on to alert when the attack is taking place. Even though firewalls can help detect and block attacks early on, with new optimization techniques allowing an attacker to execute his attack with just one request, it is imperative to resort to other measures. For these reasons, it is of utmost importance for the software to undergo rigorous security audits and only use the firewall as a first line of defense.

References

[1] "Advanced Data Mining in MySQL Injections using Subqueries & Custom Variables" by Ionut Maroiu - http://www.slideshare.net/DefCamp/advanced-data-mining-in-my-sql-injections-using-subqueries-and-custom-variables

[2] "Blind Sql Injection with Regular Expressions Attack" by Simone Quatrini and Marco Rondini - http://www.exploit-db.com/wp-content/themes/exploit/docs/17397.pdf

[3] "Fast data extraction using SQL injection and XML statements" by Daniel Kachakil - http://www.kachakil.com/papers/SFX-SQLi-Paper-en.pdf

[4] "Faster Blind MySQL Injection Using Bit Shifting" by Jelmer de Hen - http://www.exploit-db.com/papers/17073/

[5] "Faster exploitation methods in Oracle and PostgreSQL" by Dmitriy Serebryannikov - https://twitter.com/dsrbr/status/342132003270959104/photo/1

https://twitter.com/dsrbr/status/340018970054766592/photo/1

[6] SQL Injection Knowledge Base - http://www.websec.ca/kb/sql_injection

[7] "URI Encoding to Bypass IDS/IPS" - http://threatlandscape.blogspot.ca/2007/07/uri-encoding-to-bypass-idsips.html

[8] "UTF-8 Chart" - https://en.wikipedia.org/wiki/UTF-8

websec