An Alternative Approach for Real-Life SQLi Detection

Reto Ischi

OWASP AppSec Europe Research 2013
August 23, 2013
• Reto Ischi
• Lead security engineer/developer of Airlock WAF at Ergon Informatik AG
• 10+ years experience in the area of web application security/WAF
• IT interests: Web and OS security, crypto, theoretical computer science, ...
Content

• SQLi Not Yet Boring
• Real-Life SQLi Detection
• Classical Approach to Filter SQLi
• Libinjection
  – Recap
  – Weaknesses
  – Combined with Classical Approach
• New Approach Detect False Positives
• Prototype Demo
known since 15 years, remains a significant thread
Real-Life SQLi Detection
• Easy: block everything

• Even blocking any request containing SQL terms/symbols/... is difficult because of several obfuscation techniques
e.g. Roberto Salgado:

• Good classical blacklist rules tend to be very complex
• Can’t decide correctly in all cases without additional information
• Strict rules => extensive exception handling
• Block this...?

SELECT * FROM users WHERE name = '%USER%' and password = '%INJECT%';

%INJECT% (MySQL):

'='  ' &'  ' <<'  ' >>'  ' |'  ' ^'  ' like'  ' %'  ' *'  ' /'  '1
...even worse without quotes:

```
SELECT * FROM users WHERE
name = '%USER%' and PIN = %INJECT%

%INJECT%: x,y = <any number>  z = <any string>
  x or y        x||y       x<y      x=0
  'z' or y      'z'=0      0<=>>0   x<>x
  x < y         x<=y      x != y    PIN
```
Classical Approach to Filter SQLi
• Moderate complex Regex

• Categorize attack types, consider DBMS
  – Eliminating conditions (Comment symbols, ...)
  – Extending query results (UNION SELECT, string concat, ...)
  – Start of new Commands (; UPDATE...)
  – Change expression evaluation (tautologies, ...)
  – ...
Extending query result with UNION SELECT

Injection:
SELECT id,name FROM users WHERE name = 'tom' and password = hash('') UNION SELECT id, name from users WHERE (username = 'Administrator');

Obvious trivial filter:
select
Reduce false positives by adding conditions:
[\s'' ]union[\s]+(all[\s]+)?select(--|[#''\s])
Libinjection Recap
C++/python library for SQLi detection through lexical analysis

https://github.com/client9/libinjection

BSD open source license

Author: Nick Galbreath

2012@Black Hat USA
1. As-is

SELECT * FROM users WHERE id = %INJECT%

2. Inside a single quoted string

SELECT * FROM users WHERE name = ' %INJECT% '

3. Inside a double quoted string

SELECT * from users WHERE name = " %INJECT% "
<table>
<thead>
<tr>
<th>k</th>
<th>keyword</th>
<th>(</th>
<th>open brace</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>logic operator</td>
<td>)</td>
<td>close brace</td>
</tr>
<tr>
<td>1</td>
<td>number</td>
<td>B</td>
<td>group/order by</td>
</tr>
<tr>
<td>o</td>
<td>regular operator</td>
<td>n</td>
<td>none/name</td>
</tr>
<tr>
<td>U</td>
<td>union</td>
<td>f</td>
<td>function</td>
</tr>
<tr>
<td>s</td>
<td>string (quoted)</td>
<td>;</td>
<td>semicolon</td>
</tr>
<tr>
<td>v</td>
<td>at (@)</td>
<td>c</td>
<td>comment</td>
</tr>
</tbody>
</table>
' UNION SELECT * FROM pass WHERE user = 'admin'

as-is context: s n s

' ' UNION SELECT * FROM pass WHERE user = 'admin' '

single-quote context: s U k o k n k n o s

" ' UNION SELECT * FROM pass WHERE user = 'admin' "

double-quote context: s
• Folding numbers: 234.3e3 => 1
• Folding strings: “test a b c” => s
• Convert simple arithm. expressions: 7+5 => 1
• Remove comment: SELECT /* bla*/ id FROM test
• Merging: “IS”, “NOT” => “IS NOT” (single op)
• Function must be followed by a parenthesis
• …
• Fingerprints of length up to 5 to detect SQLi &1o1U, &1osU, &1ovU, &f()o, &f(1), &f(s) ....

• Compare only the first 5 tokens of the parsed strings seems to be enough
  – If yes, we're lucky => fast

• Fingerprints generated (learned) from a large list (> 47000) of SQLi (src: pentest tools, cheat sheets, forums,...)
Libinjection Weaknesses
• Bypass 5 token restriction with padding
• "as-is" context (MySQL):
  ...WHERE id = 1\1\1\1\1\1# AND...
  token representation: 10101
• Quoted context (Oracle):
  ...WHERE name = 'a'||'d'||'m'||'i'||'n'--' AND...
  token representation: s&s&s
• Mixed (MySQL):
  ...WHERE id = 1 ^ 'N' || 2# AND...
  ...WHERE id = (select @a or ( @a ))# AND...
False Positives and Lexical Analysis

1; from will create the case 7

vs.

99; UPDATE user SET type = 22

Parser may see the same:

<number>; <keyword> <name> <keyword> <name> <operator> <number>
Libinjection
+
Classical Approach
• Tokenize the full input string

• Pattern matching
  - Evasion by padding no longer possible
  - Fingerprint learning vs. human brain power

• More expensive?
Major benefit: Much simpler patterns
- Dozens of variants to separate terms (space, tab variants, CR, LF, null byte, ...)
- Long disjunction chains
  select|insert|update|delete|... => keyword

Example Pattern:
Classical Regex:
```
;.*\((execute|exec|insert|update|select|delete|
  drop|waitfor|create|alter|begin)(--|[#'(\h\v])
```
Token Regex:
```
;.*k[cs( ]
```

Slightly better detection rate but way more FP
New Approach to Detect False Positives Based on Lexical Analysis
• Which token combinations are not common in SQL?

... <NAME> <NAME> ...

... <NUMBER> <NAME> ...

... <OPERATOR> <OPERATOR> ...

...
Why not having two consecutive names in the token representation of SQLi?

Detect keywords:

```
UPDATE TEST SET A = 1 WHERE ID = 1
```

Consider strings:

```
SELECT "foo bla" FROM t
```

Remove comments:

```
SELECT id /* this is comment */ FROM t
```
Single quote context example:

Dancin' like a robot on fire

↓ ↓ ↓ ↓ ↓ ↓

string operator name name keyword name

↓ ↓

Two consecutive names => FP
Exception example for: <name> <name>

Column and table alias without AS keyword

```
SELECT id i, name FROM mytable
```

```
SELECT id, name FROM mytable m
```
• No need to consider SQLi and FP in a single complex regex:
  1) **Simple regex** to detect SQLi
     Please click on facebook's 'like' button
     Blacklist rule: `<quote><logical operator>`
  2) **Simple regex** to whitelist false positive
     Please click on facebook's 'like' button
     Whitelist rule: `<name> <name>`

• Can whitelisting step be used to evade filter?
Overall Process

input string

pre-processing: parsing, decoding, ...

tokenizing/folding

"as-is" check

single quote check

double quote check

no match

match: could be SQLi

white-list FP check

match

white-list FP exceptions check

match

no match

match

SQLi

reduce false positives

simplified figure!
Prototype Demo
Conclusions

• Real-life SQLi detection without additional info is hard

• False positives are a pain

• Lexical analysis
  – Simplifies blacklist rules
  – False positives reduction by whitelisting
  – Worth for further research

• Can we use lexical analysis to prevent other code injection attacks?
Special thanks to Erwin Huber and Thomas Lohmüller for their support

Feedback / suggestions welcome: reto.ischi@ergon.ch

http://www.ergon.ch/en/airlock