Maximizing the speed of time based SQL injection data retrieval

30c3, Hamburg, 29.12.2013

Arnim' ; DROP TABLE students;-- )
Introduction to SQL injection

- SQLi is #1 of OWASP Top 10 Web vulnerabilities
- Sample code of vulnerable php script:

```php
$sql_cmd = "SELECT * FROM user WHERE id = " . $_POST['id'];
```

- Form-Input: `42; UPDATE user SET type="admin" WHERE id=23;`

- Resulting Query:

```sql
SELECT * FROM user WHERE id =42; UPDATE user
SET type="admin" WHERE id=23;
```
SELECT author, subject FROM article WHERE ID=42 UNION SELECT login, password FROM user;

Very fast, can sometimes retrieve multiple strings in one request.
Boolean SQLi

- No output of the query can be seen
- There's an indication, if the result of a query is true or false because a certain string appears in the webpage, e.g. an error message
- Fastest retrieving method is binary search:
  - Is the ASCII-Code of the 1st character of the password of user 'admin' lower than 64?
  - If 'true': Is the ASCII-Code of the 1st character of the password of user 'admin' lower than 32?
  - ...
- Slow: Needs 7 request per ASCII character (but can be multithreaded)
Time based SQLi

- Neither output of the query can be seen nor any indication of it's result
- Only possible way to determine the result, is to let the database SLEEP() some seconds, if the query turns out false and continue immediately if it's true.
- In other databases time intense instructions have to be executed (effectively doing a short DoS)
- Very very slow and prone to errors because of hard distinction between SLEEP() and network lag
- Multithreading difficult to impossible
Common Pitfall

If a hammer is your only tool, every problem looks like a nail.
String to get: “1234”, only 4 possibilities

Is number <= 2 ?

- If true:
  - Is it 1?
    - If true: found 1
    - If false: found 2

- If false:
  - Is it 3?
    - If true: found 3
    - If false: found 4

To get 1234 it takes 8 request
Time based SQLi linear search

- Is number = 1?
  - If true: **found 1**
  - If false:
    - Is number = 2?
      - If true: **found 2**
      - If false:
        - Is number = 3?
          - If true: **found 3**
          - If false: **found 4**

To get 1234 it takes 9 request, 6 slow + 3 fast
Time based SQLi linear search 2

Let's learn from the game show “What’s my line?” aka “Heiteres Beruferaten”: some questions need to be reversed to avoid the answer “no” (e.g. "Is it something other than..." or "Can I rule out...")

- Is number $\neq 1$?
  - If false: found 1
  - If true:
    - Is number $\neq 2$?
      - If false: found 2
      - If true:
        - Is number $\neq 3$?
          - If false: found 3
          - If true: found 4

To get 1234 it takes $3$ slow + $6$ fast request

- Morse code was designed to use the short signals on the most common characters, dit = e
- According to Benford's law 1 is the most common number
Time based SQLi compare

- Binary search: 4x fast + 4x slow
- Linear search: 6x fast + 3x slow
- True usually returns after ~100ms, false after 1 second
- Binary search: 4.4 seconds
- Linear search: 3.6 seconds (and only 3x tiny DoS for DBMS without SLEEP)
- Break even for binary search on 4 choices: 1x slow = 2x fast
- Break even for binary search on 8 choices: 1x slow = 3.2x fast
<table>
<thead>
<tr>
<th>Dec</th>
<th>ASCII</th>
<th>Dec</th>
<th>ASCII</th>
<th>Dec</th>
<th>ASCII</th>
<th>Dec</th>
<th>ASCII</th>
<th>Dec</th>
<th>ASCII</th>
<th>Dec</th>
<th>ASCII</th>
<th>Dec</th>
<th>ASCII</th>
<th>Dec</th>
<th>ASCII</th>
<th>Dec</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NUL</td>
<td>16</td>
<td>DLE</td>
<td>32</td>
<td>SP</td>
<td>48</td>
<td>0</td>
<td>64</td>
<td>@</td>
<td>80</td>
<td>P</td>
<td>96</td>
<td>`</td>
<td>112</td>
<td>p</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>SOH</td>
<td>17</td>
<td>DC1</td>
<td>33</td>
<td>!</td>
<td>49</td>
<td>1</td>
<td>65</td>
<td>A</td>
<td>81</td>
<td>Q</td>
<td>97</td>
<td>a</td>
<td>113</td>
<td>q</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>STX</td>
<td>18</td>
<td>DC2</td>
<td>34</td>
<td>&quot;</td>
<td>50</td>
<td>2</td>
<td>66</td>
<td>B</td>
<td>82</td>
<td>R</td>
<td>98</td>
<td>b</td>
<td>114</td>
<td>r</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>ETX</td>
<td>19</td>
<td>DC3</td>
<td>35</td>
<td>#</td>
<td>51</td>
<td>3</td>
<td>67</td>
<td>C</td>
<td>83</td>
<td>S</td>
<td>99</td>
<td>c</td>
<td>115</td>
<td>s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>EOT</td>
<td>20</td>
<td>DC4</td>
<td>36</td>
<td>$</td>
<td>52</td>
<td>4</td>
<td>68</td>
<td>D</td>
<td>84</td>
<td>T</td>
<td>100</td>
<td>d</td>
<td>116</td>
<td>t</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>ENQ</td>
<td>21</td>
<td>NAK</td>
<td>37</td>
<td>%</td>
<td>53</td>
<td>5</td>
<td>69</td>
<td>E</td>
<td>85</td>
<td>U</td>
<td>101</td>
<td>e</td>
<td>117</td>
<td>u</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>ACK</td>
<td>22</td>
<td>SYN</td>
<td>38</td>
<td>&amp;</td>
<td>54</td>
<td>6</td>
<td>70</td>
<td>F</td>
<td>86</td>
<td>V</td>
<td>102</td>
<td>f</td>
<td>118</td>
<td>v</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>BEL</td>
<td>23</td>
<td>ETB</td>
<td>39</td>
<td>'</td>
<td>55</td>
<td>7</td>
<td>71</td>
<td>G</td>
<td>87</td>
<td>W</td>
<td>103</td>
<td>g</td>
<td>119</td>
<td>w</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>BS</td>
<td>24</td>
<td>CAN</td>
<td>40</td>
<td>(</td>
<td>56</td>
<td>8</td>
<td>72</td>
<td>H</td>
<td>88</td>
<td>X</td>
<td>104</td>
<td>h</td>
<td>120</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>HT</td>
<td>25</td>
<td>EM</td>
<td>41</td>
<td>)</td>
<td>57</td>
<td>9</td>
<td>73</td>
<td>I</td>
<td>89</td>
<td>Y</td>
<td>105</td>
<td>i</td>
<td>121</td>
<td>y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>LF</td>
<td>26</td>
<td>SUB</td>
<td>42</td>
<td>*</td>
<td>58</td>
<td>:</td>
<td>74</td>
<td>J</td>
<td>90</td>
<td>Z</td>
<td>106</td>
<td>j</td>
<td>122</td>
<td>z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>VT</td>
<td>27</td>
<td>ESC</td>
<td>43</td>
<td>+</td>
<td>59</td>
<td>;</td>
<td>75</td>
<td>K</td>
<td>91</td>
<td>[</td>
<td>107</td>
<td>k</td>
<td>123</td>
<td>{</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>FF</td>
<td>28</td>
<td>FS</td>
<td>44</td>
<td>,</td>
<td>60</td>
<td>&lt;</td>
<td>76</td>
<td>L</td>
<td>92</td>
<td>\</td>
<td>108</td>
<td>l</td>
<td>124</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>CR</td>
<td>29</td>
<td>GS</td>
<td>45</td>
<td>-</td>
<td>61</td>
<td>=</td>
<td>77</td>
<td>M</td>
<td>93</td>
<td>]</td>
<td>109</td>
<td>m</td>
<td>125</td>
<td>}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>SO</td>
<td>30</td>
<td>RS</td>
<td>46</td>
<td>.</td>
<td>62</td>
<td>&gt;</td>
<td>78</td>
<td>N</td>
<td>94</td>
<td>^</td>
<td>110</td>
<td>n</td>
<td>126</td>
<td>~</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>SI</td>
<td>31</td>
<td>US</td>
<td>47</td>
<td>/</td>
<td>63</td>
<td>?</td>
<td>79</td>
<td>O</td>
<td>95</td>
<td>_</td>
<td>111</td>
<td>o</td>
<td>127</td>
<td>_</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Avoid using non standard functions like REGEXP or RLIKE, use ASCII-ranges
Testing speedup

- Changed sqlmap to use quick request for > on 1st and 2nd request, then on <
- Retrieve sample string: AZazme_5

Sqlmap original:
[12:17:18] [DEBUG] performed 71 queries in 121.29 seconds

Sqlmap patched:
[12:14:44] [DEBUG] performed 71 queries in 103.72 seconds
Mitigation of account stealing in the wild

- So your web app is state of the art and does:
  - Input sanitation and uses prepared statements
  - Salts and hashes passwords with 50,000 rounds of PBKDF2 (more would be subject to an easy DoS by multiple authentication attempts)
  - Enforces passwords with 10 digits containing upper, lower, number, symbol + no dictionary words

- What happens:
  - A SQLi-vuln gets introduced with a new feature or a lousy plugin (check out exploit-db ;)
  - Attacker dumps hashes and cracks 10-30% of them on his GPU-cluster/cloud service in a week (*Passw0rd1!* isn't in your 100,000 word dictionary but in his containing 100,000,000 entries)
Solution: Learn from Adobes one good example!

- 150 million password-tokens leaked and not a single one got cracked (though some got guessed by the password hints)
- Encryption of OS-Passwords is considered stupid because if you can read e.g. /etc/shadow, you can read the file containing the key as well
- Store the key outside the database, unreachable for SQLi. Attacker would need a 2nd vulnerability to get it
- Passwords should be salted, hashed AND encrypted
- That's 4-6 additional lines of code
- Cracking 3DES or AES is much harder than password cracking (168 to 256 bits of entropy vs 80 of a typical password makes it $1.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000+\ times\ harder\ to\ crack$)
Solution: Learn from Adobes one good example!

- Protects even the lousiest password, 123456 is safe (from offline cracking ...)
- Known cleartext doesn't help in cracking because of the salt
- Prevents using SQLi to overwrite existing passwords (e.g. admins) or insert new ones
- Protects passwords also from attacks which get a direct db connection on TCP 3306, 1526, ... or get access to the db backups
- Encryption beats “peppering” and keyed HMACs in flexibility for combining user databases from multiple systems because decrypt/encrypt is possible
- Allows to encrypt other sensitive data like password hints as well ;)

29.12.2013

Time based SQLi @30c3
Bonus slide: Key management for webapps

- Take it easy, even just setting $key=... in config.php is much safer than just hashing passwords
- If you need more security:
  - Local file inclusion (LFI) and other methods to read arbitrary files on the webserver are the dangers
  - Store key in file with fixed prefix and random end, e.g. secret_key_e2e4dEAdheAd30c3.txt because LFI mostly can only read a known name and not search for files
  - Remove OS read permissions on file after reading it, e.g. chmod 000 secret_key_e2e4dEAdheAd30c3.txt
  - Attacker needs remote code execution to read the key file
Bonus slide: Methods for speedup

- Start with >96? instead of 64 (?)
- On ASCII-values < 32 make a lucky guess on 0 aka end of string
- Predict length of string depending on previous results, e.g. the last 3 hashes where 40 bytes long => no need to use several request to check for 0x00
- Adopt to charsets of different columns:
  - Hashes => hex = just 16 possibilities (Close gap in ASCII between numbers and lowercase chars with `SELECT @a:=ORD('a'), IF(@a > 96 AND @a <=103, @a-39, @a);`
  - Emails => make string lowercase and don't ask for uppercase with `SELECT @a:=ORD('A'), IF(@a > 96 AND @a <=122, @a-32, @a);`
- Ask for ranges instead of just greater/smaller, e.g. is the ASCII-value between 48 and 57 (=a number)?
Bonus slide: methods for speedup 2

• Predict next chars:
  • htt => p
  • q => u
  • lengt => h
  • .ed => u

• Use network QoS settings on client and router to minimize network lag

• If there's a choice, pick fastest injection point (there might be different amounts of webapp- and SQL-code executed)