Login Timing Attacks

For Mischief and Mayhem

Kiwicon 2012
Ohh Errr Research? Ok boss.

Got to thinking about side channel attacks

BEAST
CRIME
Can a timing attack be used on a remote web app to guess a hashed password faster than a simple brute force attack?
if password == storedPassword:
    loginOk()
else:
    LoginFail()
A more correct password takes longer to compare than a less correct password.

```python
def compare(str1, str2):
    if len(str1) != len(str2):
        return False
    for i in range(len(str1)):
        if str1[i] != str2[i]:
            return False
    return True;
```
How much longer?

Bugger all

$\approx 5 - 100 \text{ ns}$
Can we measure this over a fast network?

Not always, but sometimes yes
Can we measure this over a fast network?

Not always, but sometimes yes
The trick is taking multiple measurements and correctly filtering out crap measurements. 10th Percentile seem to be the best. Cheers S. A. Crosby et al.
PlainText Attack

The Process

1. Generate candidate passwords
2. Try each password, record how long it took
3. Work out if we have a slow outlier, if not GOTO step 2
4. If so, generate new passwords with known prefix, GOTO step 2
5. Stop if we have ALL THE CHARACTERS!!!!!
6. Laugh Manically (mostly optional)
PlainText Attack

The Test Process

1. Generate candidate passwords, add a known password but change the last character.
2. Try each password, record how long it took
3. Work out if we have a slow outlier, if not GOTO step 2
4. If we have a slow outlier and it's our password, WIN
5. Laugh Manically (mostly optional)
Does it work though?

Let's take a look!
PlainText Attack

POC Python Socket Server

Reads password from the network
Compares it to a hard coded password
Responds with true or false
Known Password Test

<table>
<thead>
<tr>
<th>Password</th>
<th>Count</th>
<th>10thCentile</th>
<th>isTarget</th>
</tr>
</thead>
<tbody>
<tr>
<td>baaaaaaaa</td>
<td>94786</td>
<td>322573</td>
<td></td>
</tr>
<tr>
<td>caaaaaaaa</td>
<td>94785</td>
<td>322619</td>
<td></td>
</tr>
<tr>
<td>daaaaaaaaa</td>
<td>94786</td>
<td>322627</td>
<td></td>
</tr>
<tr>
<td>eaaaaaaaaaa</td>
<td>94785</td>
<td>322631</td>
<td></td>
</tr>
<tr>
<td>gaaaaaaaaaa</td>
<td>94786</td>
<td>322635</td>
<td></td>
</tr>
<tr>
<td>faaaaaaaaaa</td>
<td>94785</td>
<td>322722</td>
<td></td>
</tr>
<tr>
<td>mcartnea</td>
<td>94786</td>
<td>322808</td>
<td>&lt;--</td>
</tr>
</tbody>
</table>

Elapsed 00:09:08 (673506 total).
Current Candidate: mcartnea, Confidence: 63.01% (86/170)

mcartnea is significantly slower than others after 679106 requests. This server is probably VULNERABLE.
Complete in 0 hrs, 9 mins.

Tool output over 100mb network
Hash Attack

How about hashes passwords?

\texttt{sha1(qwerty) = b1b3773a05c0ed0176787a4f1574ff0075f7521e48c16c7184a6b61a5b7d1a8bd3bd49413d6827cb = sha1(????)}

Cryptographic Hash Properties:

- Easy to compute
- infeasible to create a message that has a given hash
- infeasible to modify a message without changing the hash
- infeasible to find two different messages with the same hash
Simple Hash Login

if hash(password) == storedHash:
    loginOk()
else:
    LoginFail()
Hash Attack

Why is this “not” vulnerable

Our string comparisons no longer make sense
Hash Attack

However!
What about hash prefix collisions?

<insert maniacal laughter here>
Hash Attacks

We generate a bunch of prefix collisions

And perform our timing attack on those

\[
\begin{align*}
\text{sha1}(40931246) &= 7d\text{de6b3a271e5ff852c941c62ee92804e89d1da3} \\
\text{sha1}(25751109) &= 7dd61668555a3e1a9fb1a22a9e62ebabbf7eb5cc \\
\text{sha1}(03076342) &= 7dddc57024e54636985336aee94e7c0317d8bb78 \\
\end{align*}
\]
So perhaps we can steal the hash?

Nope.

Collisions get expensive

<table>
<thead>
<tr>
<th>Number of Chars</th>
<th>Time to Calculate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 to 4</td>
<td>&lt; 1 second</td>
</tr>
<tr>
<td>5</td>
<td>8 secs</td>
</tr>
<tr>
<td>6</td>
<td>4 mins</td>
</tr>
<tr>
<td>7</td>
<td>2 hrs</td>
</tr>
<tr>
<td>8</td>
<td>2.5 days</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>20</td>
<td>&gt; 3,570,000,000,000,000,000 yrs</td>
</tr>
</tbody>
</table>
Hash Attacks

What about if we don't need ALL the hash?

Prefixes are enough to be useful on large password lists

We can use the prefix to reduce the password list size

Then fall back to brute force
If hash prefix doesn't match prefix of hashed password, remove it from the list.
Theoretical Attack Time

Assuming 50 reqs/s, 32mil requests per character, correct password 30% in to list of 215 million words
Hash Attack

Cool eh?

But can we measure this?
Comparison Measurement

String Comparison

Ruby

```c
const long len = RSTRING_LEN(str1);
const char *ptr1, *ptr2;

if (len != RSTRING_LEN(str2)) return Qfalse;
if (!rb_str_comparable(str1, str2)) return Qfalse;
if ((ptr1 = RSTRING_PTR(str1)) == (ptr2 = RSTRING_PTR(str2)))
    return Qtrue;
if (memcmp(ptr1, ptr2, len) == 0)
    return Qtrue;
return Qfalse;
```
Ruby String Comparison

“Pseudo Code”

```ruby
def comp(str1, str2):
    if len(str1) != len(str2):
        return False

    for i in range(len(str1)):
        if str1[i] != str2[i]:
            return False

    return True
```
if (Py_SIZE(a) == Py_SIZE(b)
    && (a->ob_sval[0] == b->ob_sval[0]
    && memcmp(a->ob_sval, b->ob_sval, Py_SIZE(a)) == 0)) {
    result = Py_True;
} else {
    result = Py_False;
}
Python String Comparison

“Pseudo Code”

```python
def comp(str1, str2):
    if len(str1) != len(str2):
        return False

    for i in range(len(str1)):
        if str1[i] != str2[i]:
            return False

    return True;
```

← Extra Step
Comparison Measurement

This means first char is easier to guess in python

≈ 100ns first char vs < 20ns second char
Hash Attack

Does it work?

POC Python Socket Server
Test Mode

\[
\text{sha1(mcartney)} = 038cba2fbdd1cdc8209136e9df8b26fd007e371c
\]
\[
\text{sha1(44706014)} = 038cb6cc6a5c2bfaed8ec7c3b1e2c19b2c0a9935
\]

Generate collision for known password
so we don't follow the “correct login” code path
# Hash Attack

## Does it work?

### POC Python Socket Server

### Test Mode

<table>
<thead>
<tr>
<th>Password</th>
<th>Hash Prefix</th>
<th>Count</th>
<th>10thCentile</th>
</tr>
</thead>
<tbody>
<tr>
<td>46324565</td>
<td>5ae</td>
<td>60188</td>
<td>225058</td>
</tr>
<tr>
<td>31078427</td>
<td>895</td>
<td>60187</td>
<td>225238</td>
</tr>
<tr>
<td>32055653</td>
<td>489</td>
<td>60187</td>
<td>225409</td>
</tr>
<tr>
<td>14351275</td>
<td>752</td>
<td>60188</td>
<td>225467</td>
</tr>
<tr>
<td>24139348</td>
<td>60b</td>
<td>60187</td>
<td>225712</td>
</tr>
<tr>
<td>31307226</td>
<td>156</td>
<td>60187</td>
<td>225818</td>
</tr>
<tr>
<td>99409750</td>
<td>9e9</td>
<td>60188</td>
<td>225852</td>
</tr>
<tr>
<td><strong>44706014</strong></td>
<td><strong>038</strong></td>
<td><strong>60187</strong></td>
<td>226549</td>
</tr>
</tbody>
</table>

Elapsed 00:04:26 (481501 requests).
Current Candidate: 44706014, Confidence: 103.62% (697/794)
### Hash Attack

**POC Python Socket Server**

**Attack Mode**

<table>
<thead>
<tr>
<th>Password</th>
<th>Hash Prefix</th>
<th>Count</th>
<th>10thCentile</th>
</tr>
</thead>
<tbody>
<tr>
<td>87633610</td>
<td>3e2</td>
<td>59791</td>
<td>236318</td>
</tr>
<tr>
<td>32055653</td>
<td>489</td>
<td>59791</td>
<td>236344</td>
</tr>
<tr>
<td>59000794</td>
<td>a8e</td>
<td>59791</td>
<td>236346</td>
</tr>
<tr>
<td>49903503</td>
<td>dff</td>
<td>59790</td>
<td>236363</td>
</tr>
<tr>
<td>46324565</td>
<td>5ae</td>
<td>59791</td>
<td>236369</td>
</tr>
<tr>
<td>31307226</td>
<td>156</td>
<td>59791</td>
<td>236382</td>
</tr>
<tr>
<td>02541214</td>
<td>bfc</td>
<td>59791</td>
<td>236399</td>
</tr>
<tr>
<td>99409750</td>
<td>9e9</td>
<td>59790</td>
<td>236405</td>
</tr>
<tr>
<td>72799748</td>
<td>fb9</td>
<td>59791</td>
<td>236407</td>
</tr>
<tr>
<td>58589661</td>
<td>e9d</td>
<td>59791</td>
<td>236446</td>
</tr>
<tr>
<td>85230885</td>
<td>0ac</td>
<td>59791</td>
<td>236521</td>
</tr>
</tbody>
</table>

Elapsed 00:07:10 at 1878 req/s average (807703 total).
Current Candidate: 85230885, Confidence: 89.70% (75/186)

Dropping fastest candidate: 87633610
Timing attack done. The guessed hash prefix is: 0
Creating filtered wordlist...
Done! Wordlist at: /home/adrian/Desktop/genwordlist.txt
Complete in 0 hrs, 8 mins.

← Correct prefix obtained!
Hash Attack

What about a real HTTP server?

Apache, fail
(not vulnerable)

Twisted Web, win!
### Hash Attack

**Twisted Web Server**

<table>
<thead>
<tr>
<th>Password, Hash Prefix,</th>
<th>Count, 10th Centile,</th>
</tr>
</thead>
<tbody>
<tr>
<td>02541214, bfc</td>
<td>59891, 549451,</td>
</tr>
<tr>
<td>32055653, 489</td>
<td>59891, 549668,</td>
</tr>
<tr>
<td>59000794, a8e</td>
<td>59891, 549691,</td>
</tr>
<tr>
<td>58589661, e9d</td>
<td>59891, 549693,</td>
</tr>
<tr>
<td>46324565, 5ae</td>
<td>59890, 549715,</td>
</tr>
<tr>
<td>14351275, 752</td>
<td>59891, 549742,</td>
</tr>
<tr>
<td>24139348, 60b</td>
<td>59891, 549744,</td>
</tr>
<tr>
<td>31078427, 895</td>
<td>59891, 549787,</td>
</tr>
<tr>
<td>99409750, 9e9</td>
<td>59890, 549797,</td>
</tr>
<tr>
<td>78375144, 2c9</td>
<td>59891, 549838,</td>
</tr>
<tr>
<td>85230885, 0ac</td>
<td>59891, 549937,</td>
</tr>
</tbody>
</table>

Elapsed 00:23:27 at 574 req/s average (808803 total). Current Candidate: 85230885, Confidence: 39.21% (99/563)

Dropping fastest candidate: 02541214
Timing attack done. The guessed hash prefix is: 0
Creating filtered wordlist...

Complete in 0 hrs, 23 mins.

 ← Correct prefix obtained!
Introducing...

Timing Intrusion Tool

5000
The Tool

Built to explore
network timing attacks

https://github.com/aj-code/TimingIntrusionTool5000
The Tool

Modes

- Hash and plaintext test mode
  - Test timing with a known password

- Plain text length mode
  - Find the length of a plaintext password

- Hash attack mode
The Tool

Solves problems for you

- Jitter filtering based on 10th percentile after multiple measurements.
- Accurate cross-platform timing (probably).
- Socket tuning, sending, receiving.
- Hash prefix collision generation.
- Statistical calculations including automatic winner classification.
- Multithreaded wordlist reduction and attacks.
The Tool

Limitations

- Most servers will not work, but some will
- Processing on all requests must be mostly equal
- Won't work on salted hashes
- Full plaintext attack not implemented
- Untested on slow networks (ie the internet)
The Tool

Where to from here?

- This technique could be tried all over the place
- Get the tool, try it out, extend it (opensource and all)
- Apply it to other protocols authentication
- Get creative

https://github.com/aj-code/TimingIntrusionTool5000
Can a timing attack be used on a remote web app to guess a hashed password faster than a simple brute force attack?

Yes

But it's fucking hard.
Comparison Measurement

The End.

https://github.com/aj-code/TimingIntrusionTool5000