# Login Timing Attacks

# For Mischief and Mayhem

Kiwicon 2012

Adrian Hayes

# Ohh Errr Research? Ok boss.

Got to thinking about side channel attacks

BEAST

CRIME

# Ohh Errr Research? Ok boss.

Can a timing attack be used

on a remote web app

to guess a hashed password

faster than a simple brute force attack?

# Password Timing Attacks

## Simple Login

```
if password == storedPassword:
  loginOk()
else:
  LoginFail()
```

# PlainText Attack

A more correct password
takes longer to compare
than a less correct password

```
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;
```

# PlainText Attack

How much longer?

Bugger all

≈ 5 – 100 ns

# PlainText Attack

Can we measure this over a fast network?

Not always, but sometimes yes

# PlainText Attack

Can we measure this over a fast network?

Not always, but sometimes yes

# PlainText Attack

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)

```
aaaaaaaa
baaaaaaa
caaaaaaa
daaaaaaa
eaaaaaaa
…
0aaaaaaa
```

# 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)

```
aaaaaaaa
baaaaaaa
caaaaaaa
daaaaaaa
eaaaaaaa
passwora
```

# PlainText Attack

Does it work though?

Lets take a look!

# PlainText Attack

POC Python Socket Server

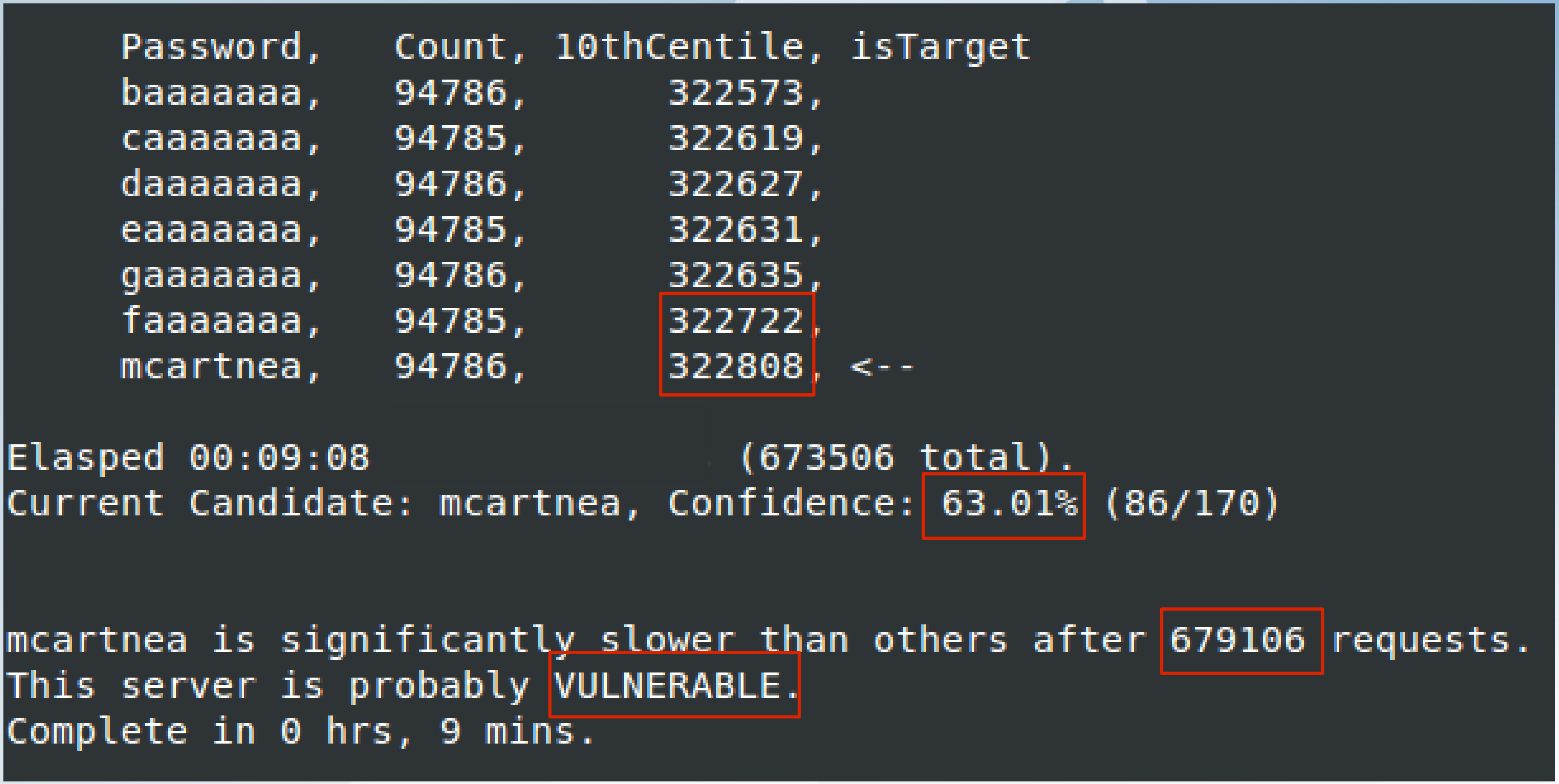
Reads password from the network

Compares it to a hard coded password

Responds with true or false

# PlainText Attack

## Known Password Test



```
     Password,    Count, 10thCentile, isTarget
     baaaaaaa,    94786,      322573,
     caaaaaaa,    94785,      322619,
     daaaaaaa,    94786,      322627,
     eaaaaaaa,    94785,      322631,
     gaaaaaaa,    94786,      322635,
     faaaaaaa,    94785,      322722,
     mcartnea,    94786,      322808, <--

Elasped 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?

```
sha1(qwerty) = b1b3773a05c0ed0176787a4f1574ff0075f7521e

48c16c7184a6b61a5b7d1a8bd3bd49413d6827cb = 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

# Hash Attack

## Simple Hash Login

```
if hash(password) == storedHash:
  loginOk()
else:
  LoginFail()
```

# Hash Attack

Why is this “not” vulnerable

```
sha1(aaaaaa) = f7a9e24777ec23212c54d7a350bc5bea5477fdbb
sha1(baaaaa) = 259b874393d7f04c76824057912ba33b2e4cebf4
sha1(caaaaa) = 48c16c7184a6b61a5b7d1a8bd3bd49413d6827cb
```

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

```
sha1(40931246) = 7dde6b3a271e5ff852c941c62ee92804e89d1da3
sha1(25751109) = 7dd61668555a3e1a9fb1a22a9e62ebabbf7eb5cc
sha1(03076342) = 7dddc57024e54636985336aee94e7c0317d8bb78
...
```

# Hash Attacks

So perhaps we can steal the hash?

Nope.

Collisions get expensive

| Number of Chars | Time to Calculate |
|---|---|
| 1 to 4 | < 1 second |
| 5 | 8 secs |
| 6 | 4 mins |
| 7 | 2 hrs |
| 8 | 2.5 days |
| … | ... |
| 20 | > 3,570,000,000,000,000 yrs |

# Hash Attacks

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

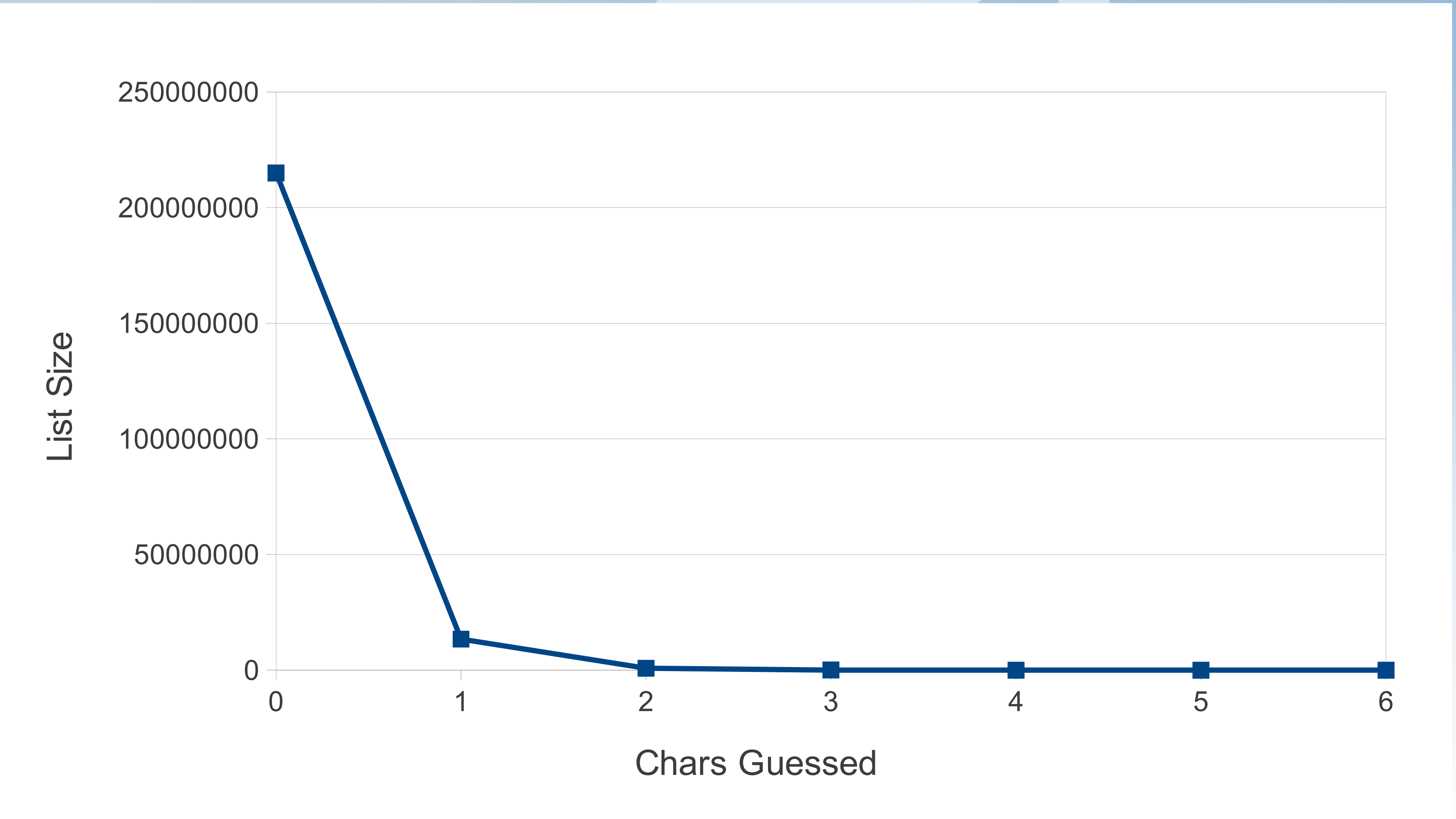
Prefixes are enough to be useful
on large password lists

We can use the prefix to
reduce the password list size

Then fall back to brute force

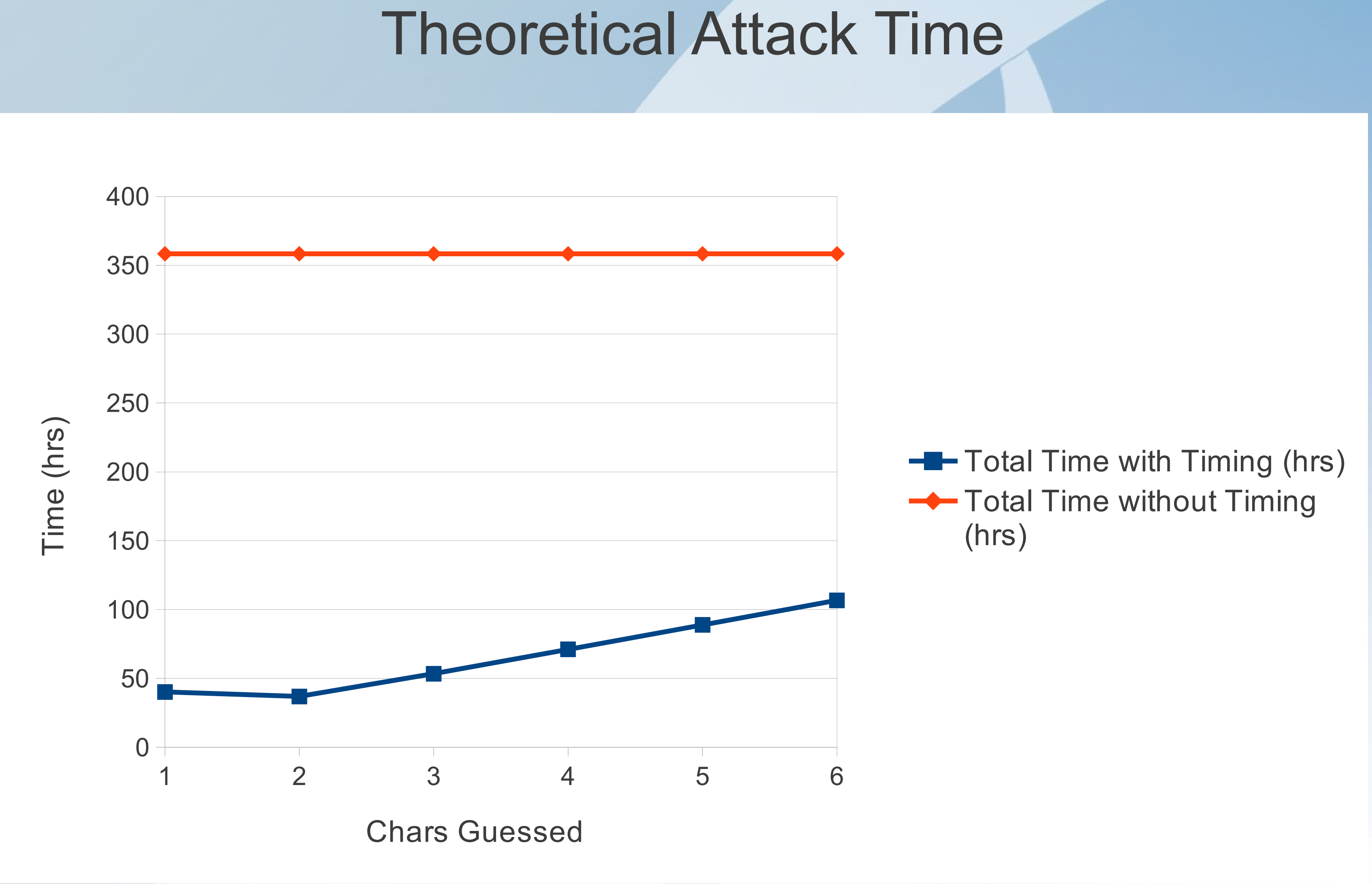
# Hash Attacks

## List Reduction on Hash Prefix



If hash prefix  doesn't match prefix of hashed password, remove it from the list

# Hash Attacks



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

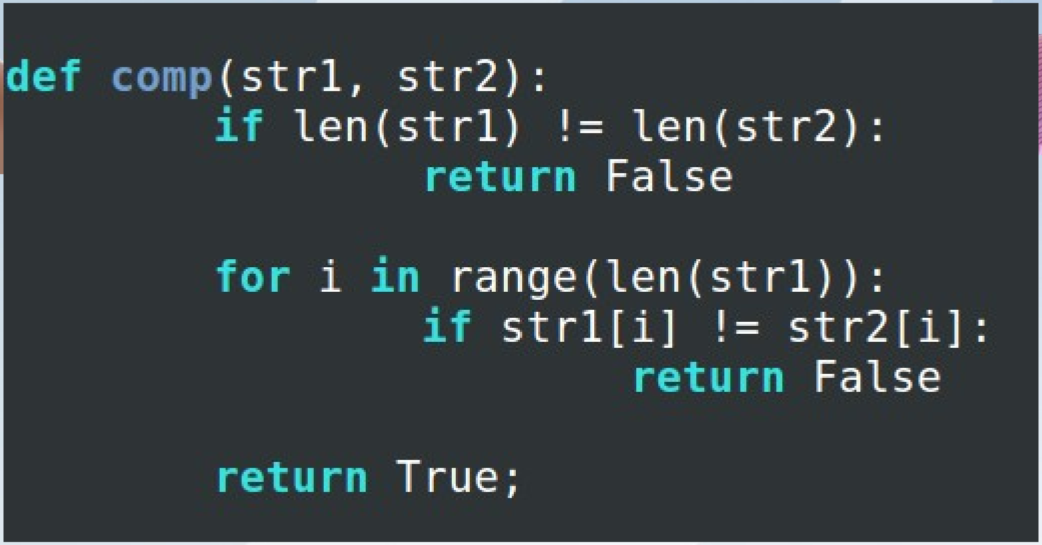
```
const long len = RSTRING_LEN(str1);
const char *ptr1, *ptr2;

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;
```

# Comparison Measurement

## Ruby String Comparison

## “Pseudo Code”



```
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;
```

# Comparison Measurement

String Comparison

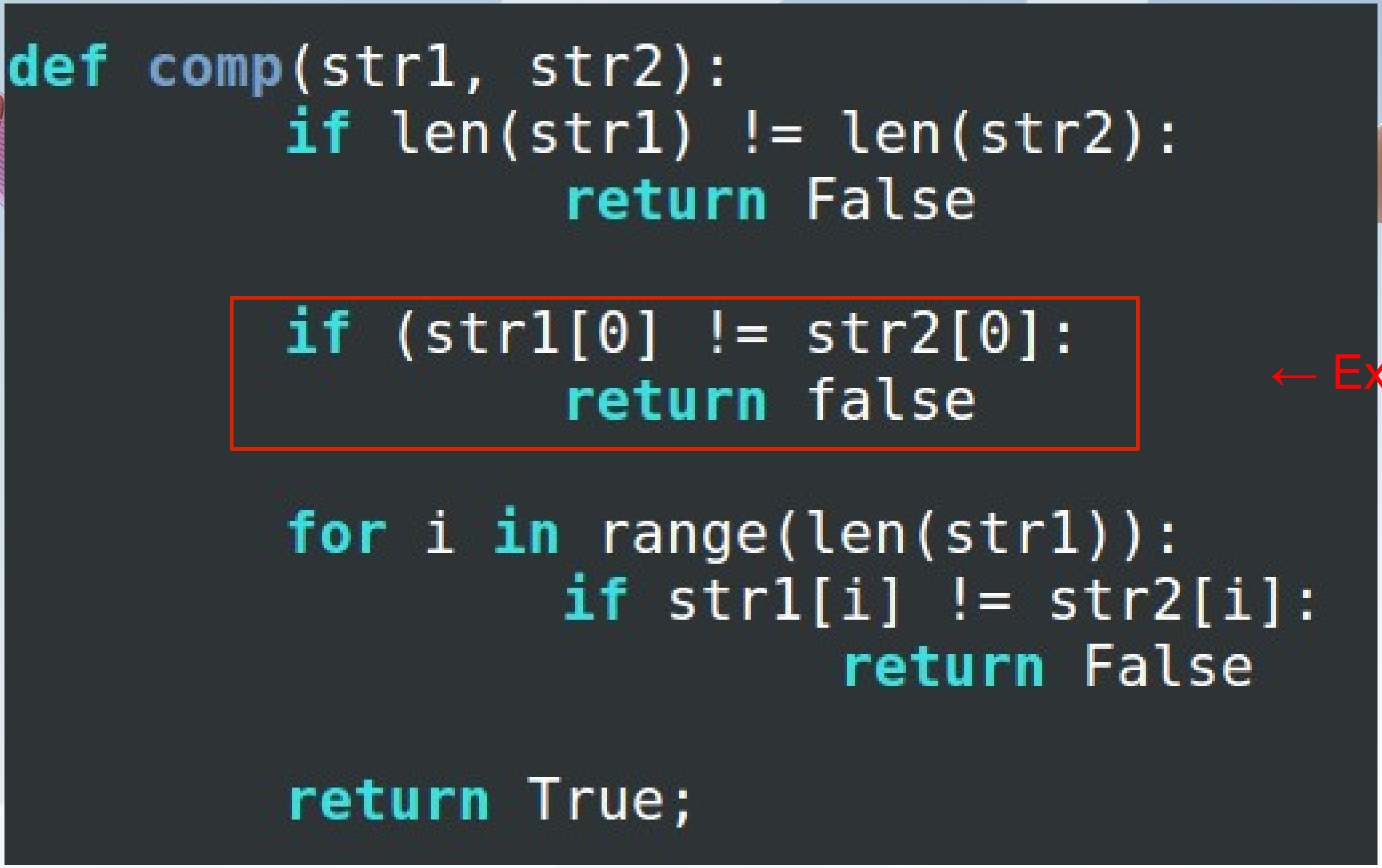
Python

```
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;
}
```

# Comparison Measurement

## Python String Comparison

## “Pseudo Code”



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

        if (str1[0] != str2[0]:
                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

```
sha1(mcartney) = 038cba2fbdd1cdc8209136e9df8b26fd007e371c
sha1(44706014) = 038cb6cc6a5c2bfaed8ec7c3b1e2c19b2c0a9935
```

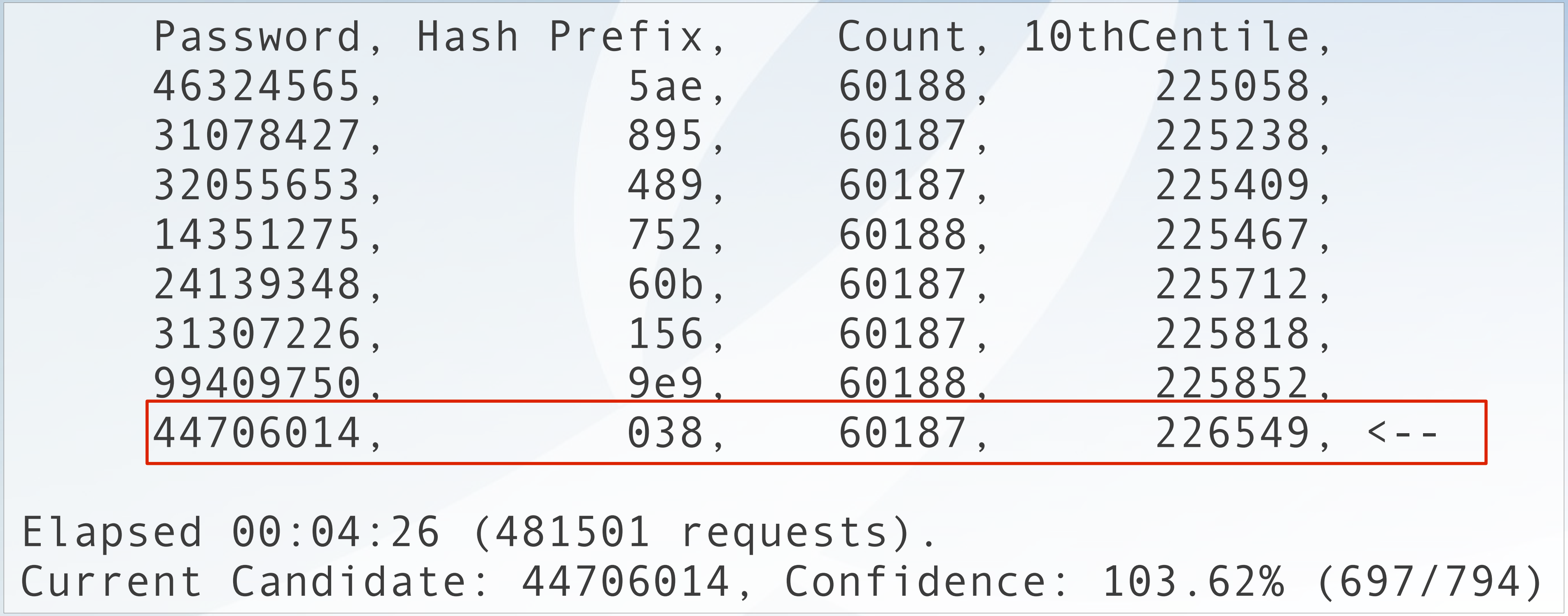
Generate collision for known password
so we don't follow the “correct login” code path

# Hash Attack

Does it work?

POC Python Socket Server

Test Mode



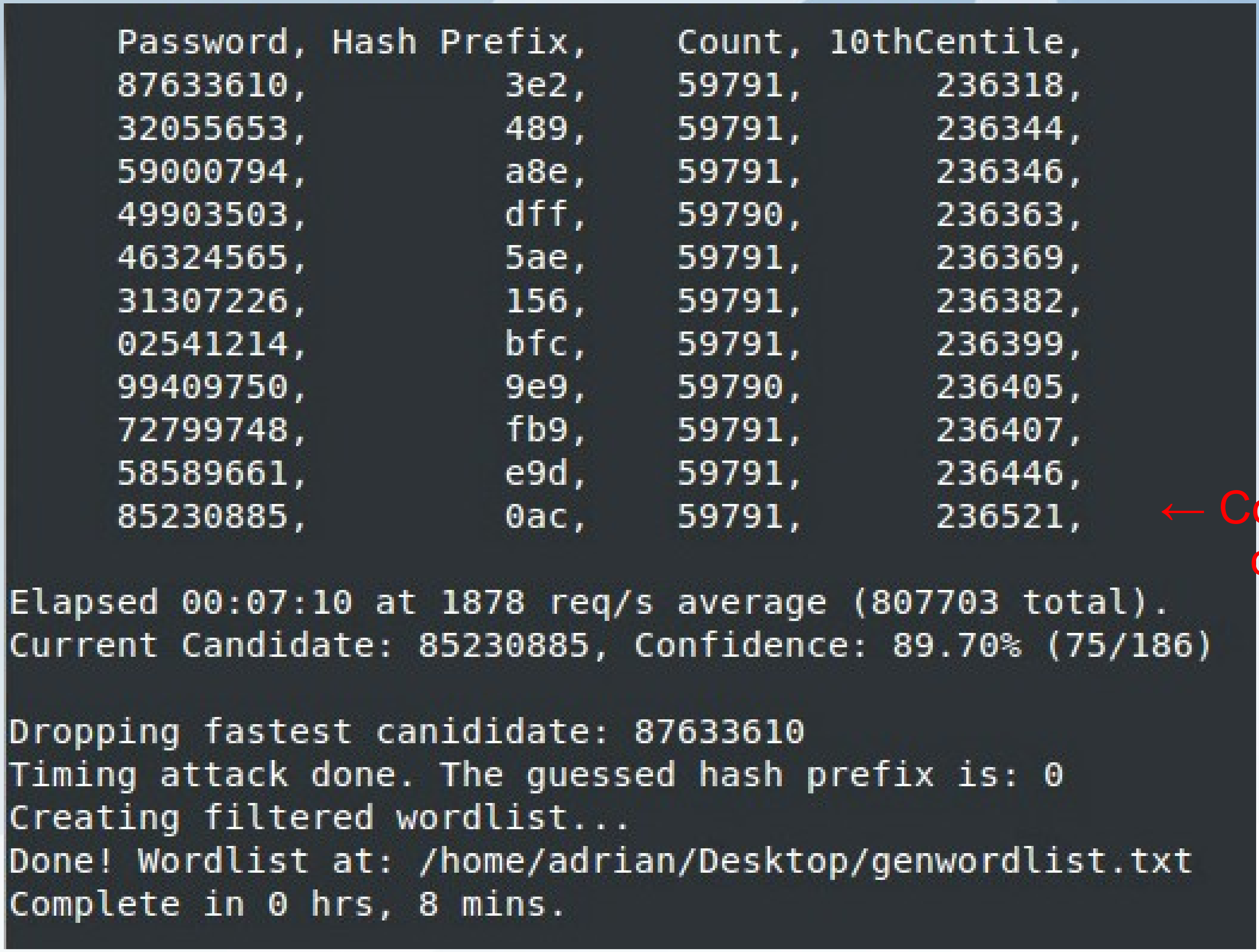
```
     Password, Hash Prefix,    Count, 10thCentile,
     46324565,         5ae,    60188,       225058,
     31078427,         895,    60187,       225238,
     32055653,         489,    60187,       225409,
     14351275,         752,    60188,       225467,
     24139348,         60b,    60187,       225712,
     31307226,         156,    60187,       225818,
     99409750,         9e9,    60188,       225852,
     44706014,         038,    60187,       226549, <--

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

# Hash Attack

## POC Python Socket Server

## Attack Mode



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

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

Dropping fastest canididate: 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

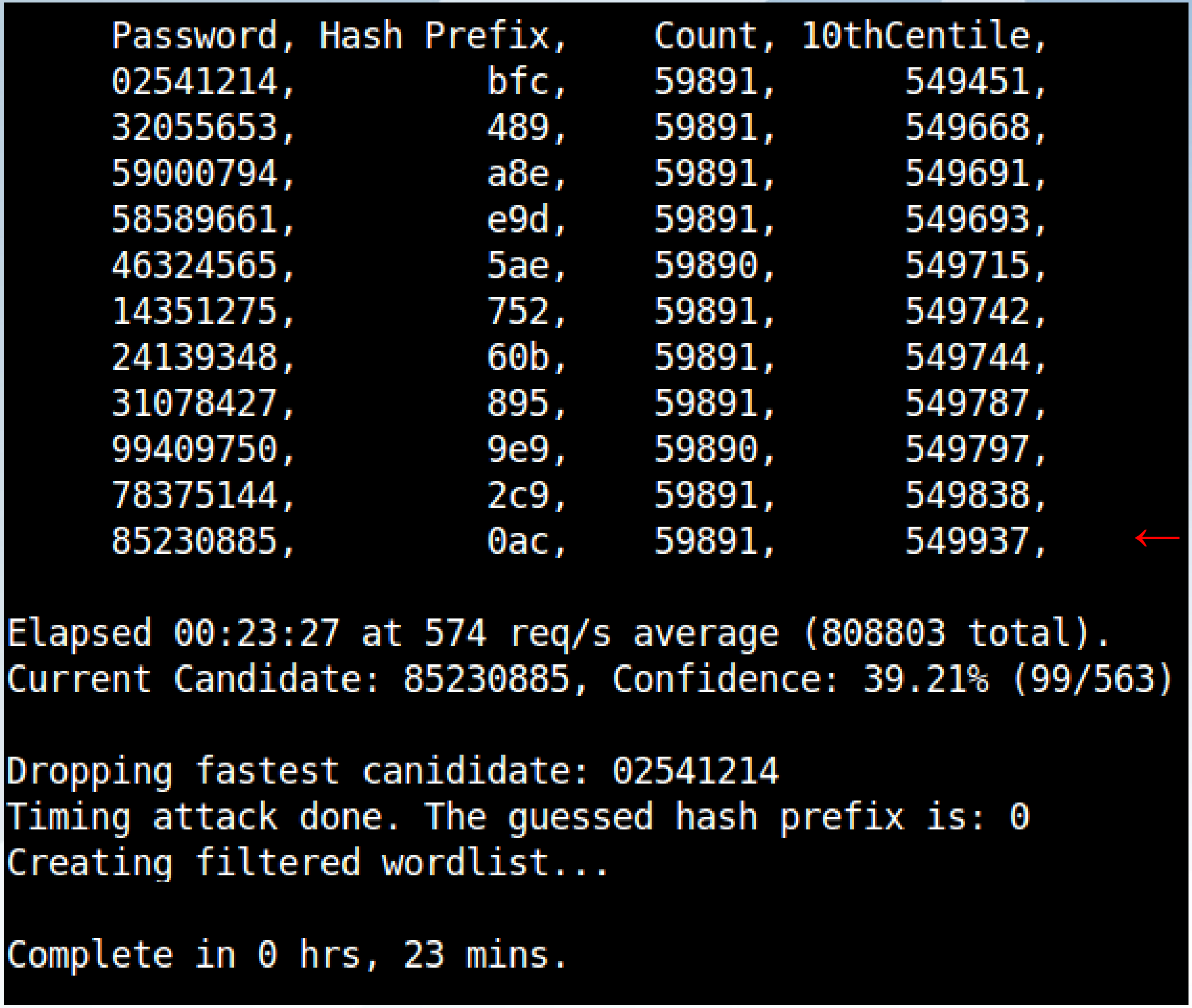
(not vulnerable)

Twisted Web, win!

# Hash Attack

## Hash Attack

## Twisted Web Server



```
     Password, Hash Prefix,    Count, 10thCentile,
     02541214,         bfc,    59891,      549451,
     32055653,         489,    59891,      549668,
     59000794,         a8e,    59891,      549691,
     58589661,         e9d,    59891,      549693,
     46324565,         5ae,    59890,      549715,
     14351275,         752,    59891,      549742,
     24139348,         60b,    59891,      549744,
     31078427,         895,    59891,      549787,
     99409750,         9e9,    59890,      549797,
     78375144,         2c9,    59891,      549838,
     85230885,         0ac,    59891,      549937,

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

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

Complete in 0 hrs, 23 mins.
```

← Correct prefix obtained!

# The Tool

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
- Wont 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

# Conclusion

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