Attacking The Internet of Things (using time)

Paul McMillan

Who am I?

There are many ways
to attack the
Internet of Things

Demo
(start)

What is a timing attack?

```
def authenticate_user(user, pass):
    stored_hash=get_password_hash(user):
    if stored_hash:
        test_hash = sha1(password)
        if test_hash == stored_hash:
            Return True
    Else:
            Return False
```

Many Kinds

- User Enumeration
- Blind SQL Injection
- CPU Cache attacks against crypto
  - Local
  - Cross-VM
- Lucky 13
- Many more...

String Comparison Timing Attacks

memcmp

```
while (len != 0)
    {
      a0 = ((byte *) srcp1)[0];
      b0 = ((byte *) srcp2)[0];
      srcp1 += 1;
      srcp2 += 1;
      res = a0 - b0;
      if (res != 0)
         return res;
      len -= 1;
    }
```

MASSIVE Speedup

c = character set

n = Length of string

Brute Force: c^n

Timing Attack: c * n (* x)

(x is # tries to distinguish)

Why are they interesting?

What are the drawbacks?

Let's talk about time

Internet SF-NY 70ms
Spinning Disk 13ms

Ram Latency 83ns
L1 cache 1ns
1 cpu cycle ~0.33ns

Speed of light
in network cable

1 meter in ~5ns
200 meters ~1µs

So… how long does each byte of that string comparison take?

nanoseconds

(on a modern 3Ghz machine)

What about something a little slower?


PHILIPS
I WANT / JE VEUX
hue
PERSONAL WIRELESS LIGHTING
Available on the App Store
PHILIPS
I WANT / JE VEUX
hue

Network timing precision

Sources of Imprecision

- Graphics drivers
- Background networking
- USB Devices
- Power saving measures
- Audio devices
- Etc.

Software Timestamps are noisy.

Let's use hardware!

(picture of i350 + adapter)

Data Collection

- Generate repeated traffic
- TCPdump the packets
- Analyze the data
- Feed back to traffic gen

Making things work

- Libpcap 1.5.0+
- TCPDump 4.6.0+ (released July 2, 2014)
- Recent-ish Kernel

Compile these from source.

In theory, this might work on OSX?

It works on Ubuntu 14.04 for me.

Nanoseconds. Ugh!

- Scapy doesn't read the pcap files
- Neither do most other packages
- Wireshark does!

- Nanosecond timestamps lose precision if you convert them to a float()
- So we subtract a large offset, and don't work with raw timestamps.
- Use integer Nanoseconds rather than float seconds

What is the Hue API?

- GET /api/<user token>/lights
- Basic RESTful API
- Not very smart - always returns http status 200 even when returning errors.
- User token is the only required auth

  (no username, no sessions)
- Not very fast (can handle ~30req/s)

Hue Base Oddities

- Running FreeRTOS/6.0.5
- Network stack is dumber than a sack of rocks
- SSDP implementation ignores most parameters
- Each HTTP response is split into ~8 pieces
- HTTP stack ignores most incoming headers
- Ethernet Frame Check Sequence sometimes wrong
- Noisy: Broadcasts SSDP, ARPs for OpenDNS

Statistics!

Basic Review

- What is the Null Hypothesis?
- What does it mean to reject the null hypothesis?
- What are we fundamentally trying to do here?
  - We're sorting samples into groups, and trying to identify the outlier

More Stats

- Why can't we use the t-test?
- What is the K-S test, and why does it help us here?
- What other approaches might we use?

[a series of yet to be completed example data graphs]

Data Prep

- Discarding data (2 standard deviations above the mean?)
- How to do that wrong!
- Why?
- [graph of prepped data]

Code

- In the repo now, public after the talk:

  https://github.com/PaulMcMillan/2014_defcon_timing
- 3 separate but related scripts
- Don't forget to save your data for re-analysis
- Starting points for analysis, not full blown attack tools

[brief browse through the code]

[Demo discussion, dissection of working or failure. Draw some graphs]

Tips and Tricks

Keep the socket open (if you can)

Configure your network card parameters properly

- use hardware timestamps
- turn off queues
- use nanosecond timestamps (gah!)
- Turn off some offloads

Make everything Quiet

- reduce extraneous traffic to the device
- Slow loris to exclude other traffic
- don't run extra stuff on your attack machine
- Profile your victim – discard noisy periods

Do a warmup run before starting to collect data!

Find the simplest possible request

Avoid statistical tests that assume your data is normal

Gather data on all hypothesis concurrently

Randomize the request order

Don't overwhelm the device

Don't forget you can stop and brute force a token

Some code short circuits if strings aren't the same length.

Try both:
Fast and Noisy
Slow and Quiet

Which one works best will vary.

Don't get fooled by your own data!

When in doubt, take more samples.

Questions?
http://github.com/PaulMcMillan/2014_defcon_timing

Repo contains updated slides and a copy of many useful references.

Attacking The Internet of Things (using time)

Paul McMillan

Are my presenter notes showing up properly?

Who am I?

Paul McMillan
Security Engineer at Nebula (I build clouds)
Security team for several prominent open source projects (not as exciting as you think it is)
Developer outreach
Mostly Python
Like building distributed systems
Like taking theoretical vulnerabilities and making them practical

There are many ways to attack the Internet of Things

You generally own the device, so physical attacks all work:

- Take it apart, read the flash memory
- Disassemble the firmware from the manufacturer
- Exploit shitty embedded C
- Fuzzing
- Logic errors
- RF

Most of the standard network security errors are present too:

- Random open ports
- Old and vulnerable OS/application code
- Etc.

We could go on forever. However,

We're here to talk about timing attacks.

Demo
(start)

This is a Philips Hue base station. That's a zigbee connected light. I figured they're a pretty good example from the “internet of things that are put on the internet for no good reason”

What is a timing attack?

What is a timing attack, anyway?

Well, at the most basic level, asking the computer to do any operation takes time. A measurable amount of time. A timing attack exploits this.

An attacker uses timing measurements to test hypotheses:

```
def authenticate_user(user, pass):
    stored_hash=get_password_hash(user):
    if stored_hash:
        test_hash = sha1(password)
        if test_hash == stored_hash:
            Return True
    Else:
            Return False
```

This is a pretty typical example of how user authentication works (yes, I know, it has problems, I left stuff out to keep this simple)

<talk through the code>

You'll notice that if it finds a user, it does some extra work. In this case, that work involves calculating the sha1 of the provided password. That's an expensive operation.

An attacker can exploit this code to figure out which users have accounts in the system. Obviously, this isn't a vulnerability in all systems (some publish that data), but it's an unintended behavior.

Many Kinds

- User Enumeration
- Blind SQL Injection
- CPU Cache attacks against crypto
  - Local
  - Cross-VM
- Lucky 13
- Many more...

The point here is that these all follow a common pattern:

The attacker makes a guess about something

The computer uses that to compute

The attacker observes how long that takes (over many samples) and infers whether their hypothesis was correct

String Comparison Timing Attacks

However, today we're here to talk about string comparison timing attacks.

These are often more difficult to exploit, and usually written off as completely hypothetical vulnerabilities

(Hopefully my demo today will help fix that misconception)

memcmp

```
while (len != 0)
    {
      a0 = ((byte *) srcp1)[0];
      b0 = ((byte *) srcp2)[0];
      srcp1 += 1;
      srcp2 += 1;
      res = a0 - b0;
      if (res != 0)
        return res;
      len -= 1;
    }
```

Snippet from http://www.nongnu.org/avr-libc/

<talk through the snippet>

The key takeaway here is that it stops as soon as it finds 2 non-matching bytes.

Unlike the previous example, we don't have to guess an entire username at once to get our timing difference. Instead, we just have to guess the first character.

MASSIVE Speedup

c = character set
n = Length of string

Brute Force: c^n
Timing Attack: c * n (* x)
(x is # tries to distinguish)

If we're brute forcing an 8 character password with 16 possible characters (I like hex), we have to try a total of 4.2 billion guesses. That's likely to be impractical in the real world.

On the other hand, if we're conducting a timing attack, and let's say it takes us 10000 guesses per character to determine a timing difference, that works out to about 1.2 million.

If you bump the length up to 10, you're looking at a trillion for brute force, and just 1.6 million for the timing attack.

Obviously, this is a worthwhile attack, if you can pull it off.

Why are they interesting?

What are the drawbacks?

Why are they interesting?
- They're more "pick the lock on the front door" than "find a backdoor"
- They don't require "bad code" (just non-security mindful code)
- Most people don't list them among "practical" exploits
- Commonly overlooked

What are the drawbacks?
- lots of network traffic
- really time consuming
- work best on a slow devices
- Susceptible to network noise / device contention

Let's talk about time

It's hard to get an intuitive grasp on small amounts of time.

Let's look at some examples.

Internet SF-NY 70ms
Spinning Disk 13ms

Ram Latency 83ns
L1 cache 1ns
1 cpu cycle ~0.33ns

Numbers more or less from here:
http://duartes.org/gustavo/blog/post/what-your-computer-does-while-you-wait/

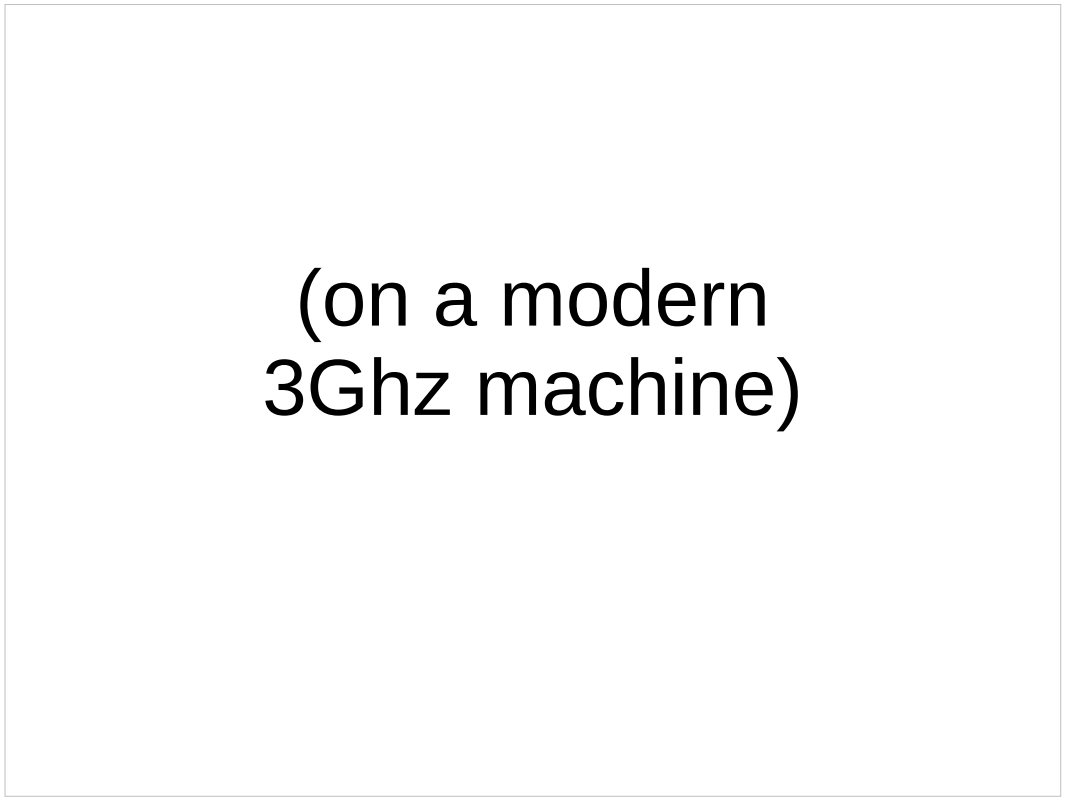
Speed of light
in network cable

1 meter in ~5ns
200 meters ~1µs

Remember that 1 microsecond is the minimum resolution recorded by the default kernel timestamps.

So... how long does each byte of that string comparison take?

nanoseconds



And that's the catch, Ladies and Gentlemen...

What about something a little slower?

120 Mhz STM32 processor
Zigbee to the lights
10 base T network port

The perfect device to demonstrate string comparison timing attacks.

The comparison is still going to take nanoseconds, but it's going to take quite a lot of them.

Image from the Philips press kit

Network timing precision

It turns out, if you want timing measurements good to the millisecond, or so, you're set. The kernel's networking stack works fine.

However, since the effects we're looking for are much smaller than that, we need to work harder.

Sources of Imprecision

- Graphics drivers
- Background networking
- USB Devices
- Power saving measures
- Audio devices
- Etc.

For a first try, it makes sense to just try the basic, default kernel packet timestamping. It turns out, it's really noisy.

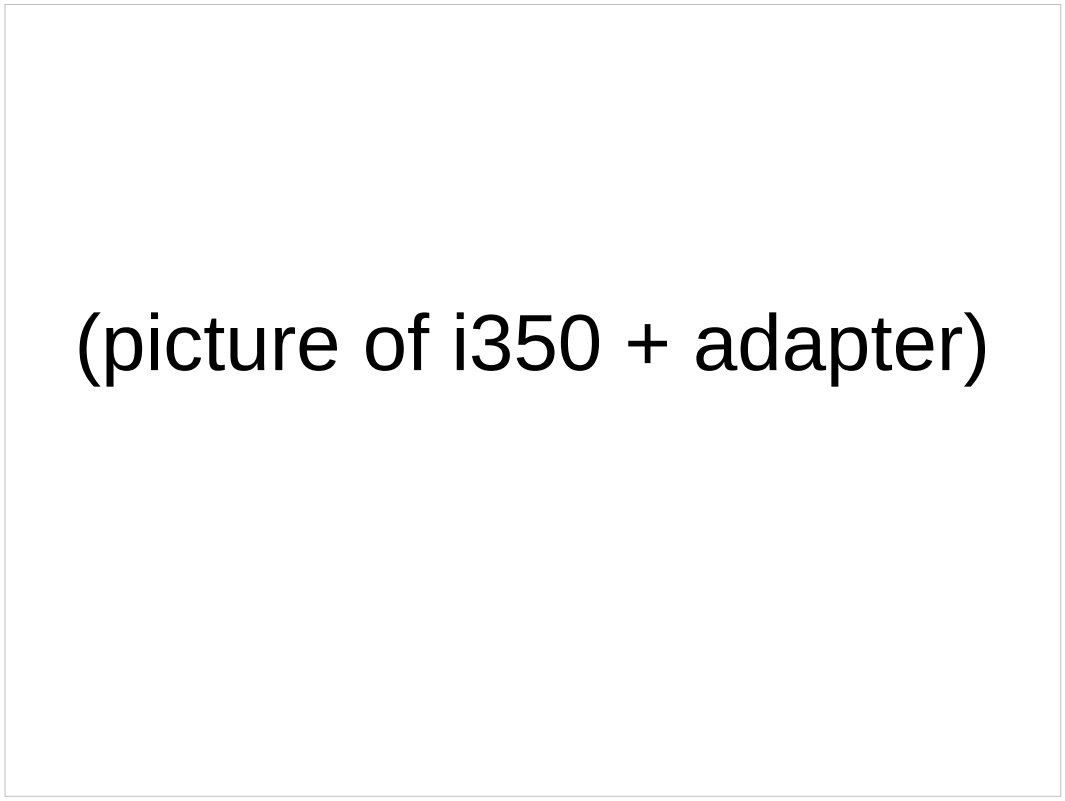
Software Timestamps are noisy.

Let's use hardware!

It turns out this is easier now than when I started working on this a year ago. Linux support for hardware timestamps works out of the box in recent distros.

Most of the results we're going to discuss in the rest of the talk CAN be obtained without hardware timestamp support. You just need many many more samples.

Try and find a NIC with hardware support – they're pretty common now. Hardware with support for PTP (IEEE 1588) usually works.



Since my laptop hardware doesn't seem to support hardware timestamps, I went looking for a high-quality source. It's always better to start with great hardware and work your way down, than to start with bad hardware and realize you need better.

This is the Intel i350 NIC, which supports hardware timestamping with an 8ns resolution (the datasheet claims). It's connected through an expresscard to PCIe adapter, allowing me to run it from my laptop.

It is directly connected to the Hue Base Station, since we don't want extra hardware introducing unpredictable latency, especially for a live demo.

This is what we're doing the demo with right now.

Data Collection

- Generate repeated traffic
- TCPdump the packets
- Analyze the data
- Feed back to traffic gen

So we have our target, we have the hardware which will allow us to carefully measure it, and we have our chosen attack technique. What's next?

Three fairly simple scripts. The first just generates login attempts over and over again. It takes an existing guess at the token, and appends a random character, then fires off the network request.

While it's doing this, a second script makes sure that TCPDump is capturing all relevant traffic, using hardware timstamps, at nanosecond level detail.

A third script periodically analyzes all captured data up to this point, and determines whether to advance the generator to a new guess.

Making things work

- Libpcap 1.5.0+
- TCPDump 4.6.0+ (released July 2, 2014)
- Recent-ish Kernel

Compile these from source.

In theory, this might work on OSX?
It works on Ubuntu 14.04 for me.

I'm going to take a slight diversion now and explain the practical steps to get nanosecond level hardware timestamps out of tcpdump.

Until very recently, tcpdump only supported microsecond level timestamps. Libpcap added support a while ago, but tcpdump didn't add it until a couple months ago. Hardware timestamp support is also relatively new. The upshot of all this is you need to install tcpdump 4.6.0 and libpcap 1.5.0+.

Nanoseconds. Ugh!

- Scapy doesn't read the pcap files
- Neither do most other packages
- Wireshark does!

- Nanosecond timestamps lose precision if you convert them to a float()
- So we subtract a large offset, and don't work with raw timestamps.
- Use integer Nanoseconds rather than float seconds

Nanosecond timestamps turn out to be inconvenient to work with.

For starters, very few things understand the new file format (this will get fixed soon I hope).

Wireshark does understand pcap files with nansecond captures, and so we use tshark as our disector to analyse our capture file. This isn't very efficient.

If you float() a nanosecond capture, you lose the last couple digits. To avoid imprecision and mistakes here, we prefer to work only in integers, and subtract a large offset to avoid overflows.

What is the Hue API?

- GET /api/<user token>/lights
- Basic RESTful API
- Not very smart - always returns http status 200 even when returning errors.
- User token is the only required auth

  (no username, no sessions)
- Not very fast (can handle ~30req/s)

Hue Base Oddities

- Running FreeRTOS/6.0.5
- Network stack is dumber than a sack of rocks
- SSDP implementation ignores most parameters
- Each HTTP response is split into ~8 pieces
- HTTP stack ignores most incoming headers
- Ethernet Frame Check Sequence sometimes wrong
- Noisy: Broadcasts SSDP, ARPs for OpenDNS

Statistics!

(Groan) but don't worry, we're only going to cover what you need to know

Basic Review

- What is the Null Hypothesis?
- What does it mean to reject the null hypothesis?
- What are we fundamentally trying to do here?
  - We're sorting samples into groups, and trying to identify the outlier

More Stats

- Why can't we use the t-test?
- What is the K-S test, and why does it help us here?
- What other approaches might we use?

[a series of yet to be completed example data graphs]

Data Prep

- Discarding data (2 standard deviations above the mean?)
- How to do that wrong!
- Why?
- [graph of prepped data]

Don't forget that these stats only work if your data points are approximately aligned in time, and you have the SAME NUMBER OF THEM!

This doesn't work at all with lopsided data sets. It's easy to accidentally get there.

Code

- In the repo now, public after the talk:
  https://github.com/PaulMcMillan/2014_defcon_timing
- 3 separate but related scripts
- Don't forget to save your data for re-analysis
- Starting points for analysis, not full blown attack tools

[brief browse through the code]

[Demo discussion, dissection of working or failure. Draw some graphs]

Tips and Tricks

These are going to come fairly rapid-fire, but most of them represent at least an afternoon of learning associated with failure.

Keep the socket open
(if you can)

Configure your network card parameters properly

- use hardware timestamps
- turn off queues
- use nanosecond timestamps (gah!)
- Turn off some offloads

Make everything Quiet

- reduce extraneous traffic to the device
- Slow loris to exclude other traffic
- don't run extra stuff on your attack machine
- Profile your victim – discard noisy periods

Do a warmup run before starting to collect data!

Yes, physical warmup

Find the simplest possible request

Avoid statistical tests that assume your data is normal

Gather data on all hypothesis concurrently

You really can't come back later and mix them up.

Randomize the request order

Don't cycle repeatedly. You're more likely to hit cache and cyclical timing weirdness.

Don't overwhelm the device

Don't forget you can stop and brute force a token

Some code short circuits if strings aren't the same length.

Try both:
Fast and Noisy
Slow and Quiet

Which one works best will vary.

Don't get fooled by your own data!

When in doubt, take more samples.

Questions?
http://github.com/
PaulMcMillan/
2014_defcon_timing

Repo contains updated slides and a copy of many useful references.