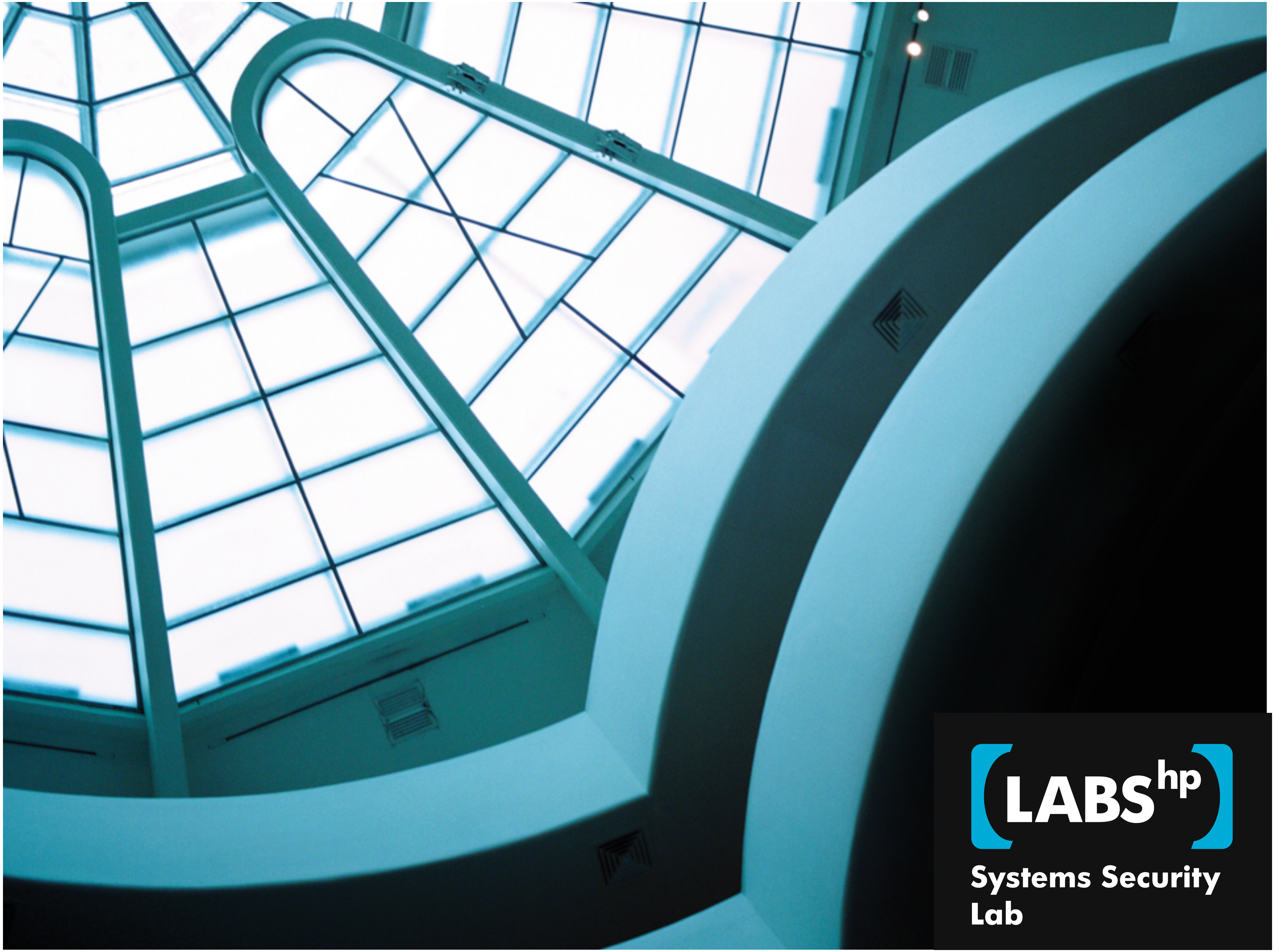


LABS hp

Systems Security Lab

PhlashDance:

Discovering permanent denial of service attacks against embedded systems

EUSecWest 08

Rich Smith, HP Labs

LABS hp Systems Security Lab

Who am I ?

- Rich Smith
- Lead the Research in Offensive Technologies & Threats (RiOTT) project for HP Labs
- Part of the Systems Security Lab
- Based out of Bristol, UK

Why am I talking about this?

- An industry wide issue, not vendor specific
- We are ahead of in the wild attack
- No point ‘n’ click solutions, requires actions from both developers and users
  - Anything requiring users cant be done behind the scenes
- Proactive is key, pretending the attack focus isn’t changing is naive and utopian

Before we continue!!

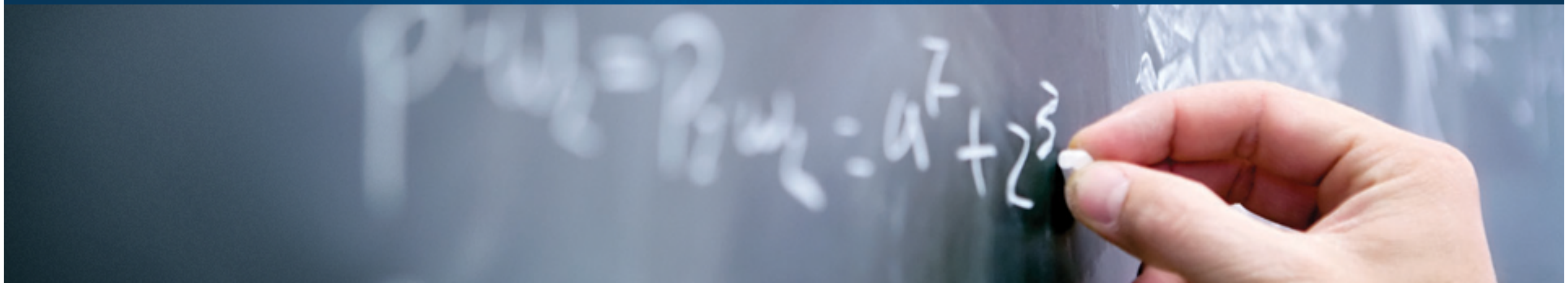
- All examples will be generalised

- No zero day to be given away :p
- Take away the overall message…
- …. Don't get hung up on specific bugs

Outline

- Permanent Denial Of Service – PDOS
- Research motivations
- Phlashing – A method of remote PDOS
- The PhlashDance fuzzing framework
- Conclusions
- Q&A

PDOS



Permanent Denial Of Service - PDOS

- Denial Of Service (DOS):
  - Defn: '*The prevention of authorized access to a system resource or the delaying of system operations and functions*'*
- Service restored upon:
  - Cessation of overwhelming traffic
  - Restarting service
  - Restarting system

- Permanent Denial Of Service (PDOS)
  - Defn: *'DOS attack requiring the introduction of new hardware, or out of band hardware re-initialisation in order to restore service'*
- Service not restored with a restart
- AKA *Bricking*

* Definition from sans.org

LABS hp

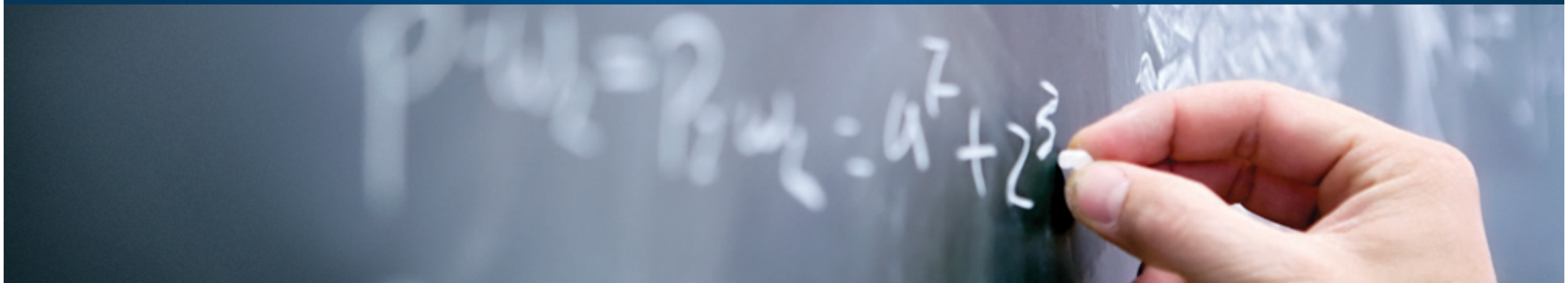
Methods of PDOS

- Both require somewhat ‘local’ access!

Remote PDOS ?

- The research questions raised are:
  - Could PDOS be achieved remotely, without physical access ?
- If so:
  - Can a generic attack strategies be found?
- And (obviously):
  - How could such attacks be mitigated?

Firmware



Why start to look at firmware?

- Major industry efforts to secure the endpoints
- …causing shifts in target focus
- Attack amplification – 1 to many devices
- Firmware generally behind software in terms of secure development & deployment
- In the past is an area that has been over looked, though that is starting to change…..

(in)secure development

- Often lots of legacy code
- Code foundations not designed for current use
- Secure development not as established as in software
- Security mechanisms that are in place are often basic
- New features == new security models
  - Difficult to manage overall device security
  - One password often not enough

(in)secure deployment

- Many devices fall outside of the security perimeter
- Not included in audit
- May not have security policies
- Default security configurations often left
- Firmware not updated – if it works leave it alone!
- Difficult to manage heterogeneous device pool
- No off the shelf products to check for compromise
- Administrators unaware of many features

Focus on firmware update mechanisms

- Almost all network attached embedded devices now have remote firmware update mechanisms
- Part of the reality of product development
  - Post release product bugfix & enhancement
- Part of the customer support model
  - If it stops working rollback to known good firmware
- Reduce administration costs

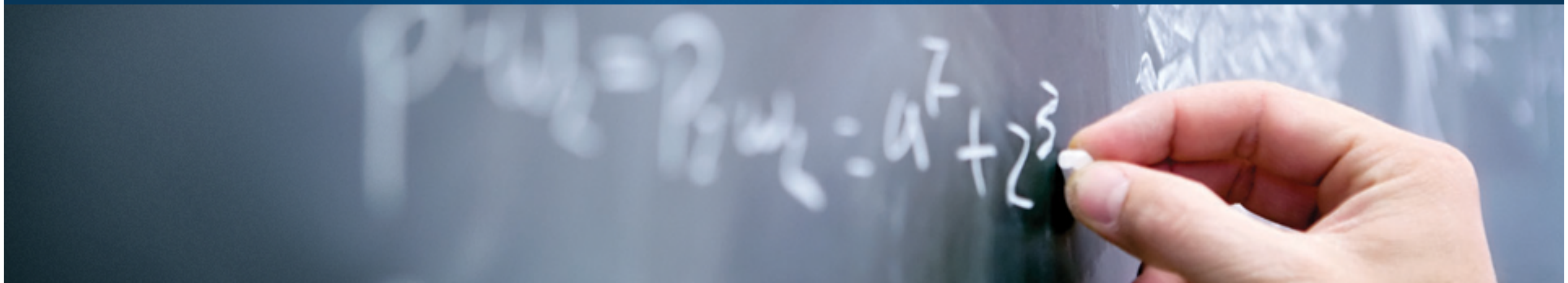
Flash update mechanisms & PDOS

- Good candidates for PDOS attack point as:
  - Turned 'ON' by default
  - Firmware binaries freely available on the net
  - Designed with error detection in mind, not malicious attack
  - The bootblock is not immutable, can be updated
    - Many devices need to boot into full OS to be reflashed
  - Only rudimentary security applied to reflash mechanisms
    - Few systems cryptographically protect firmware – most use CRC's
    - Access control often very weak given the power reflash access gives
    - Some systems bypass access control for recovery purposes!

Firmware update mechanisms

- Two generalised update methods:
  - PUSH: The FW binary is just sent to the device.
    (Typically via FTP, SMB or raw TCP)
  - PULL: The FW update is signaled to the device.
    (Typically via SNMP)
    The device then connects back to fetch the binary .
    (Typically via TFTP)
- Client side software utilities simplify the process, maybe also do additional validation

‘Phlashing’



Phlashing – because everything needs *'ph'ing* !

- One method of remotely achieving PDOS
- (mis)using flash update mechanisms to corrupt flash memory in a way which renders the device:
  - Unbootable (corrupt the boot block/loader)
  - Non-reflashable (through normal 'inband' methods)

Phlashing – Attacking flash mechanisms

- Blackbox research
- To attempt remote PDOS, a devices flash update mechanisms were attacked, manipulation of:
  - Binary firmware file format
  - Flashing application level protocol
  - Flash update logic bugs & flaws

LABS hp

Phlashing – Why bother?

- Why not malware or rootkit the firmware??*
  - Both have their place, its not really one or the other
- Different attack focus
  - Extortion & reputation damage – stealth not required
- Easier to accomplish, achievable with:
  - Hex editor
  - Protocol analyser
- Fits into existing criminal *business models* – easily adopted
- So likely to see sooner

*See Sebastian's talk later

Phlashing – Why bother?

- Highly effective brand attack tool
  - Against both vendor or owner
- Higher costs of recovery for victim & vendor
  - Require new hardware & field installation
  - Longer diagnosis & downtime
- Lower cost of realisation for attacker
  - Fire and forget – unlike ddos
  - Can be conducted via internal trojaned boxes (email)
  - Few ongoing costs – No ‘rent-a-botnet’ required

Phlashing game plan

- Diff firmware files
- Understand file construction & headers
- Find CRC'c & algorithms
- Look at flash application traffic (use mibdepot!)
- Generate test traffic to flash good image
- Find ranges that CRC's cover
  - Wrote a little utility called *legwarmer* to try and work out CRC algorithm and byte range used
- Now fuzzing can begin……

Binary file format or firmware updates

- Start to reverse engineer the binary file:
  - Most firmwares split into sections
  - Headers for each section + files headers contain:
    - Sizes & offsets
    - Section ID's, types & orderings
    - Memory addresses of entry points / decompression points
    - 'Magic bytes' for delimination & image ID
    - Version & device model numbers
    - Padding
    - CRC's
    - ….

Example binary file points of interest

- Identify memory addresses & alter values
  - Often entry points etc
  - Both ASCII ‘0xAABBCCDD’ & integer AABBCCDD
- Section duplication/deletion/reordeing
- Fuzz on areas identified as:
  - Integers
  - Strings
  - Padding
  - Magic Bytes

CRC's & Checksums

- Most (though not all!) firmwares use some form of checksum
  - Designed to pick up accidental 'errors on the wire'
  - NOT intentional manipulation
  - Many are not cryptographic so can be regenerated
  - Surprisingly even though present sometimes not used
  - Often multiple checksums per file
    - Sometimes distinct sometimes overlapping/cascading
  - Almost always 32 bits in length
    - CRC32, XOR accumulation, homebrew crazyness

CRC's & Checksums

- Even if they are cryptographic (or you just can't work out the algorithm) attacks may still be possible:
  - Multi-section binaries may not have overall checksum
    - Often due to device memory limitations and flash devices not being designed with security in mind
  - Headers may not be covered by CRC's
  - Occasionally the device does NO crypto checking, all done in client software and simple CRC on device

Flash application protocol

- As devices gain functionality the number of ways in which a device can receive firmware updates have increased:
  - TFTP, FTP, HTTP, SMB, RAW TCP, Netware etc
  - Different protocols often use different code paths….
  - ….which have been added to the codebase overtime
- Initiate multiple flashes in parallel – race condition
- Restart flash many times – memory exhaustion
- Call remote reboot function/bug during flash

Privilege escalation

- Should an admin have the right to damage hardware if he doesn’t have physical access??
- Also acts as a bridge to allow a kind of privilege escalation:
  - Those with only ‘logical’ access privileges (e.g. sys-admins) to have some of the rights that those with ‘physical’ access privileges (e.g. DataCentre admin) should have.
  - Gives a degree of physical touch to those with only logical privileges
  - This can break many associated risk/threat models and assumptions

Mitigations

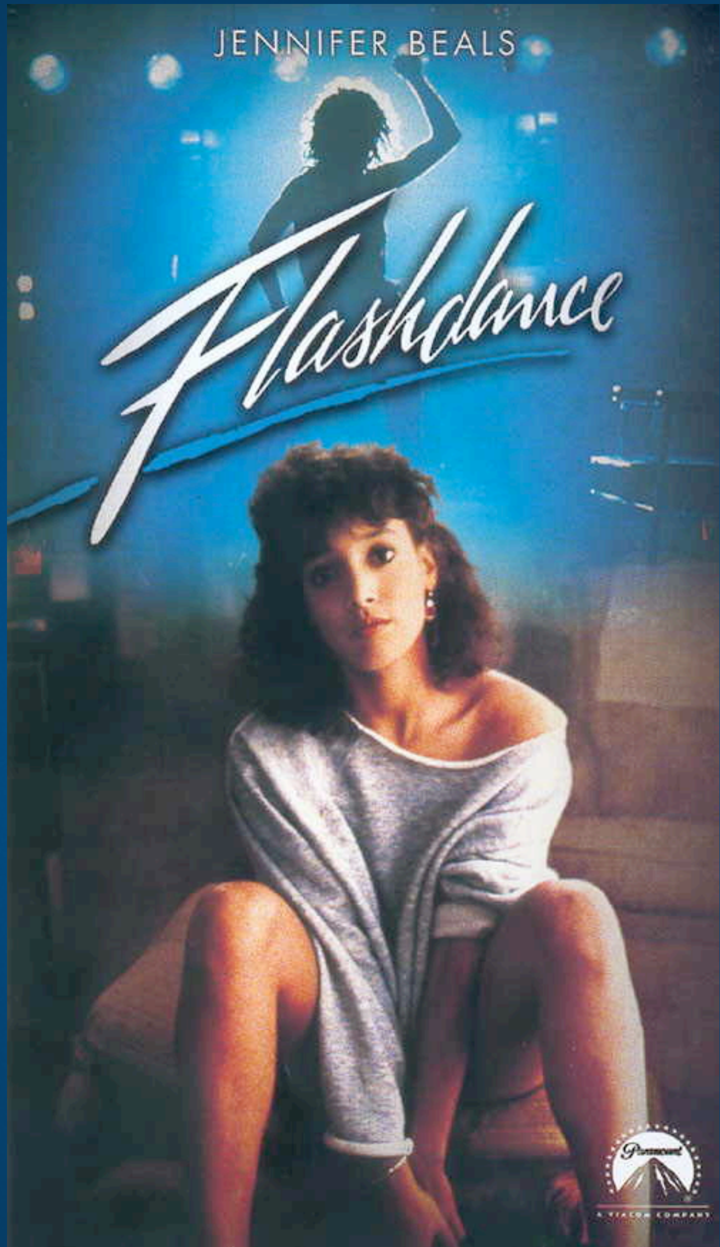
Developers

- Remote updates OFF by default
- Physical presence required to flash
- Crypto signatures on binaries
- Validation in firmware not client application
- Design with attack tolerance not fault tolerance

Mitigations

Users

- Take device security seriously
- Understand the full capabilities of device
- Lock devices down
- Patch your firmware


JENNIFER BEALS
Flashdance
Paramount
A VIACOM COMPANY

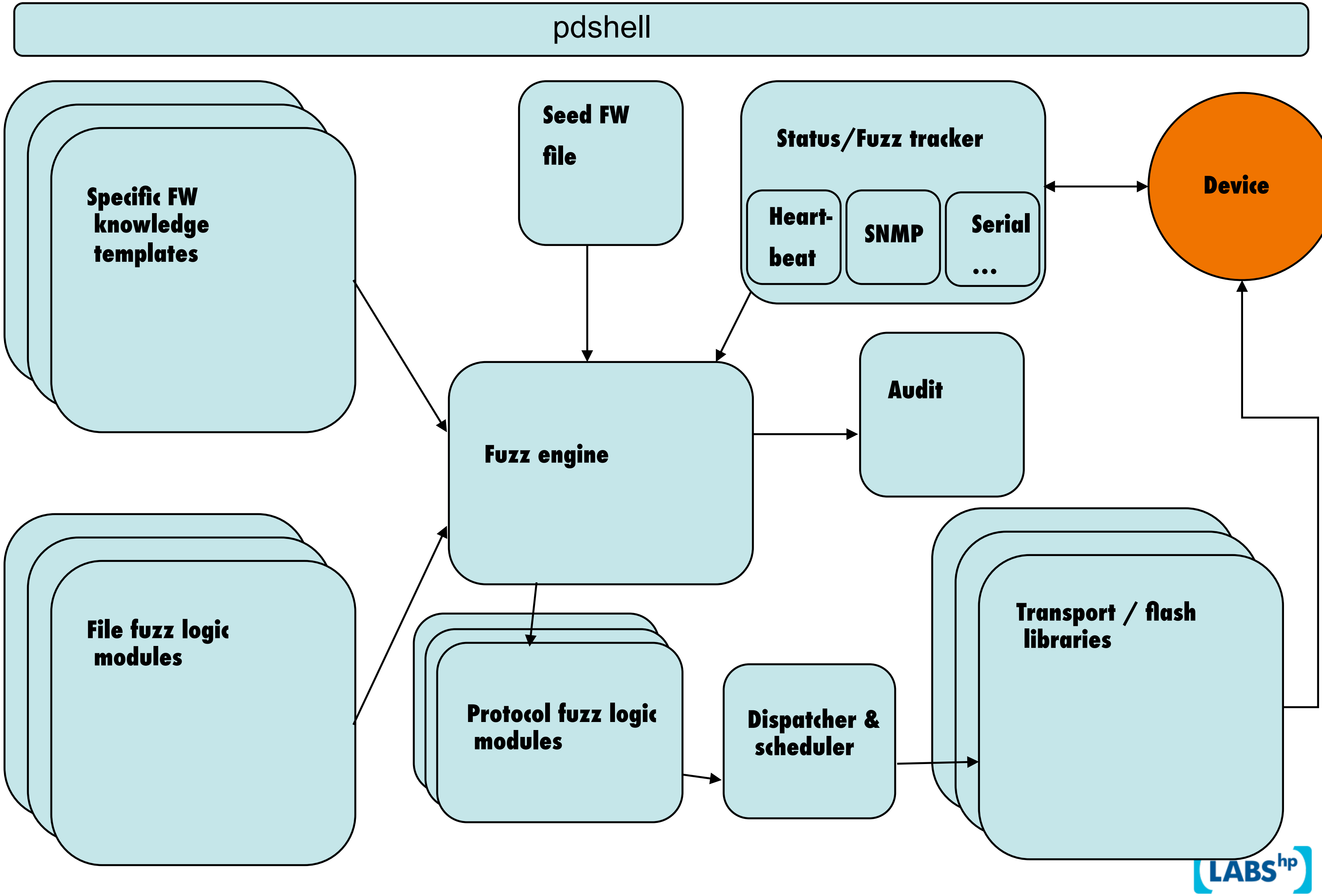
PhlashDance - The need for automation

- Finding such bugs a good task for a fuzzer:
  - Tedious, repetitive, slow, huge number of possibilities
- A combination between file-format fuzzing & protocol fuzzing
- Run against hardware not software
- Decided to write one from scratch for the experience + so it would fit my needs exactly
- Written entirely in python

PhlashDance – Design goals

- Fuzz to specifically find phlash bugs
- Integrate tool into secure product development lifecycle
- Usable non-security skilled engineers
- Fuzz engine be generic as possible across devices
- Easily extendable to new devices
- Modular fuzz logic, expand library over time
- Repeatable fuzz runs
- Transport protocols <u>not</u> a fuzz target (FTP etc)
  - Plenty of tools already capable of this

Phlashdance high level architecture



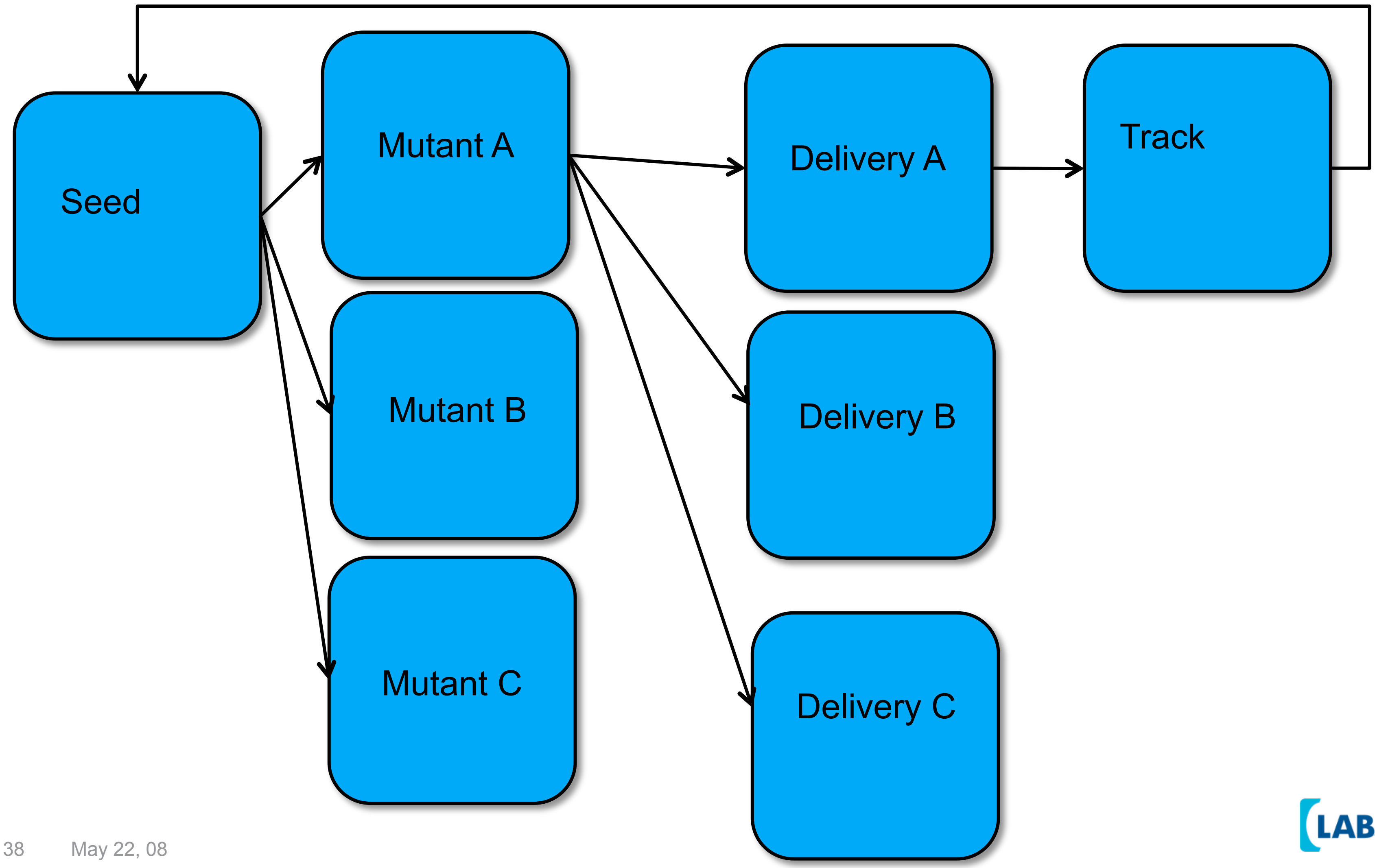
Phlashdance – device independence

- Mutation based fuzzer – using firmware binary seed
- Template based approach per device
  - Including checksum calculations
  - Fuzz tracking specifics
- Common fuzz logic to all devices
- Backend library of common flash transports
- Fuzz tracking via abstraction layer aclling back to template & common libraries

Phlashdance – Workflow

- Workflow:
  - Seed file + template
  - File fuzz logic creates *x* mutants
  - CRC mutants
  - Protocol fuzz *x* mutants to *y* flash runs
  - Send to device
  - Track progress
  - ++

Phlashdance – device independence



Phlashdance firmware knowledge template

- Ideally the knowledge template is the only thing that should need to change for new device….
- Knowledge template consists of:
  - Version number
  - Seed file location
  - Offsets & ranges for data types we have fuzz interest in
  - Flash transports this devices has available
  - Checksum algorithm + checksum offset/ranges
  - Fuzz tracking API calls

Phlashdance example template

- example_template.py

Phlashdance – Fuzz logic

- Fuzz logic is designed to be generic & modular
- Self selecting based upon template variables
- Each module has a UUID
- Can inherit from other logic modules
- File fuzz logic creates 1 or more mutants
- Protocol fuzz logic takes each mutant & for the specified transports applies logic to initiate 1 or more flash processes

Phlashdance – Fuzz logic example

```
from delim_logic import *
from block_logic import *

class partition_prepend(delim_logic):
   uuid="2-0"
   requires=["partition_marker"]

   def __init__(self, vars):
      self.logic_name="Partition prepend"
      delim_logic.__init__(self, vars)

   def logic(self):
      """
      This logic places a number of bytes in front of the partition marker
      which indicates separate parts of the firmware
      """
      ##Long string repeats various chars – BOF ticklers
      self.mutant_images.extend(self.prepend_long_string(delims=self.partition_marker))
      ##Long string repeats format string ticklers
      self.mutant_images.extend(self.prepend_format_string(delims=self.partition_marker))

      ##Long string repeats the partition marker
      self.mutant_images.extend(self.repeat_delim(delims=self.partition_marker))
```

Phlashdance – hardware differences

- Fuzzing software targets allows tracking by attaching debugger
- Hardware makes this difficult
  - Every device has different ways to track progress
  - Different granularities
  - Makes knowing when to start testing for PDOS tough
  - Often no data on what went wrong
- Much slower – flash write latency

Phlashdance limitations

- V slow – need quite a bit of parallel hardware
- Granularity of errors & tracking difficult
- CRC implementation a bit clunky
- More work needed on protocol fuzzing

Phlashdance future

- Emulation
  - Deep fuzz tracking possible – greater fuzz depth
  - Will make more generic across devices
- Auto generate the firmware template from firmware at compilation time
- Improve fuzz tracking & pdos detection (JTAG?)
- Integrate into a slicker firmware security QA process – look at the bigger lifecycle picture

Phlashdance advantage to vendor

- Access to lots hw & fw knowledge
- When emulation support is complete much faster than attackers can be
- Understand fw lineage

Conclusions

- Just because something hasn't happened publically yet doesn't mean we shouldn't evaluate potential risks
- Most problems stem from the low security profile firmware is given
- Risk to firmware need to be understood from the time of architecture & development
- Well designed firmware can be badly deployed
- Meaning the fix is not simple, but multi layered

Conclusions

- Phlashdance a start in a way to bring firmware security wrt PDOS into the development lifecycle
- Vendors in an advantageous position over attackers
- Fuzzing hardware is heaps good fun and there is plenty of ground left for others to explore

Thanks for your time!

Questions ?

LABS hp