Verified Boot: Surviving in the Internet of Insecure Things

Randall Spangler
Chrome OS Firmware Lead

Introduction

Who am I?

- Chrome OS firmware engineer since 2009
- Co-architect of the Chrome OS verified boot reference code
- But today, speaking as a Coreboot developer

What do we want? A boot process that is secure, robust, and open.

When do we want it? Seven years ago…

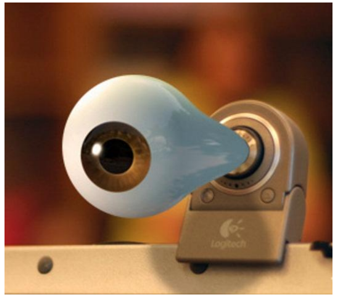
Then and Now

In 2009, focused on boot security for a new open-source laptop

Then and Now

In 2009, focused on boot security for a new open-source laptop

Now, there are a lot more devices.



Talk Outline

- What is verified boot?
- Idealized boot flow
- Verified boot in Coreboot today
- What's coming?
- How you can help

What is Verified Boot?

Verified Boot Means Different Things to People

Normal User: It's running official stuff, which I trust.

Open Source Developer: It's running <u>my</u> stuff, which I trust.

Open Source User: It's running XXX developer's stuff, which I trust.

Content Provider: It's running stuff, which I trust not to leak my content.

Enterprise / Education: It's running official stuff I trust, but I don't trust my users.

What Do We Want To Protect?

User and third-party data

- Stored on device
- Access to the cloud

Owner's control of software running on the device

- Botnets
- Ransomware
- Schools must limit student device usage

Protecting Against Whom?

Remote attackers (most important)

Local attackers with casual access

Local attackers with long-term access but not specialized tools (NEW!)

Local attackers with specialized tools (Ha!)

But not against legitimate device owners!

- You bought it, you can do what you want with it
- Hard tradeoffs to protect the other cases but still allow this

Only As Secure As Where You Start

Mask ROM and eFused keys are <u>really</u> read-only

- Also usually closed-source. (Backdoors, extra keys?)
- Who holds the signing keys?

SPI Flash can be physically read-only (write protect pin)

eMMC / SATA / NVMe storage has its own upgradeable firmware

Only As Secure As the Worst Bug You Can Patch

Read-only boot stage must do as little as possible

- RAM init should be in a later RW firmware stage

Rollback protection for later boot stages

- Need lockable NV storage for this (TPM, or NVMs + discrete logic)
- Keys leak; better make them rollable too

Updates need to be painless and may fail

- Multiple copies of RW firmware

Need a Robust Read-Only Recovery Path

Minimal attack surface

- Recovery path itself is an attack vector for local attackers
- Network recovery is hard

Triggering recovery mode

- Automatically, on verified boot failure
- Manually, by local user

What About Open Source Developers and Users?

What's the point of being open source OS and firmware if you can't use your own?

Developers should have access to all device functionality

- But… not necessarily all pre-installed content keys

Developers should be able to protect their devices too

- "Developer mode" does not mean "insecure mode"

Developer Modes

Developer OS Mode

- Requires physical access to enable
- Warning screen or lights at boot time

Developer Firmware Mode

- No warning screen possible
- Requires long-term physical access

Enterprise owners need to be able to disable developer modes (reversibly!)

Performance

Verified boot shouldn't hurt boot time significantly

Verify only what you need for next stage

Some ARM SoC boot a slow core first

Hardware crypto support on some SoC (but can we trust it?)

Verified Boot Flow
(simplified)

Read-Only (RO) Firmware

Check for recovery request (manual or automatic)

Check rewriteable (RW) firmware

- Two firmware slots (A, B)
- Verify firmware data key using root key (from RO)
- Verify firmware payload using firmware data key
- Rollback protection for both firmware data key and payload

NV storage for firmware versions - roll forward (if needed) then lock (always)

Jump to RW firmware

Rewritable (RW) Firmware

Initialize components/devices

- DRAM, storage (eMMC / SATA)
- Maybe display, keyboard, USB

Check for Developer OS Mode

- Developer flag in lockable NV storage
- Bypasses kernel data key check (below)
- Warning screen with 30-sec timeout, bypass with key combo
- Normal users can disable developer OS mode from screen
- Developers can boot USB or legacy BIOS (SeaBIOS) if enabled

Rewritable (RW) Firmware (continued)

Check OS Kernel

- Multiple kernel partitions in GPT with priority/tries fields
- Verify kernel data key using kernel subkey (from RW)
  ...unless in developer OS mode (then any kernel data key will do)
- Verify kernel payload using kernel data key
- Rollback protection for both kernel data key and payload

NV storage for kernel versions - roll forward (if needed) then lock (always)

Jump to OS kernel

Recovery Firmware (read-only)

Display recovery screen; wait for user to insert recovery media

- Use recovery key (from RO) to verify recovery OS kernel
- No rollback protection
- No developer mode

NV storage for firmware, kernel versions left unlocked

Local user can enable developer OS mode from firmware recovery screen

- Press physical key to confirm
- Store flag in lockable NV storage

Keys and More Keys

Key	Verifies	Stored In	Versioned	Notes
Root Key	Firmware Data Key	RO Firmware	NO	Private key in a locked room guarded by laser sharks; N of M present. RSA4096+
Firmware Data Key	RW Firmware	RW FW Header	YES	Private key on signing server. RSA4096.
Kernel Subkey	Kernel Data Key	RW Firmware	YES (as FW)	Private key only needed to sign new kernel data key. RSA4096.
Kernel Data Key	OS Kernel	OS Kernel Header	YES	Private key on signing server. RSA2048.
Recovery Key	Recovery OS Kernel	RO Firmware	NO	Locked room and laser sharks. RSA4096+. Different than all keys above. Signs recovery installer, not payload.

Verified Boot in Coreboot Today

Vboot Reference

https://chromium.googlesource.com/chromiumos/platform/vboot_reference/+/master/

Verified boot reference library, called by Coreboot and Depthcharge

- firmware/2lib, firmware/lib20 - "vboot 2" RW firmware verification
  - Low memory footprint, designed to run in cache-as-RAM
  - firmware/include/vb2api.h - top level header for APIs
- firmware/lib - "vboot 1" OS kernel verification, firmware screen UI loops
  - Still pretty specific to clamshell laptops and Chromeboxes
  - Clunky; originally designed to be called from to UEFI BIOS
- firmware/lib/cryptolib - Software libraries for SHA, RSA
- firmware/lib/cgptlib - GPT parsing

Coreboot

https://chromium.googlesource.com/chromiumos/third_party/coreboot/+/chromeos-2016.05/src/vendorcode/google/chromeos/

Recovery checking and RW firmware verification, using vboot_reference (2.0) library

- vboot2/vboot_logic.c - verstage_main(), calls into vboot_reference
- vboot2/antirollback.c - uses TPM to implement lockable NV storage
- vboot2/vboot_handoff.c - translation to old vboot1 shared data
- vbnv.c - NV storage (CMOS, SPI flash) for other state (recovery request, etc.)

Depthcharge

https://chromium.googlesource.com/chromiumos/platform/depthcharge/+/master/src/vboot/

OS kernel verification and firmware screens, using vboot_reference (1.0) library

- main.c, stages.c - thin wrapper around VbSelectAndLoadKernel()
- screens.c - boot screen compositing, display (but UI loop inside vboot1)
- callbacks/ - calls back into depthcharge from vboot_reference
  - disk.c - read/write sector-based disk
  - display.c - calls up to screens.c
  - keyboard.c - get keystrokes
  - nvstorage*.c - read/write non-lockable storage for vboot settings
  - ec.c - interface to embedded controller for EC software sync
  - tpm.c - TPM interface for lockable storage (but layer of calls inside vboot1)

What's Coming?

Device Form Factors

Clamshell laptops ⇒ Chromeboxes ⇒ Tablets and routers ⇒ EVERYTHING

- No matrixed keyboard - maybe just a button or two
    - Depthcharge board/rush_ryu/keyboard.c
- No display - maybe just an LED or two
- Extremely cost sensitive

What are we doing?

- Draw screens in depthcharge (vboot/screens.c) instead of compositing from bitmaps inside vboot_reference (dnojiri@chromium.org)
- Moving UI loop from vboot_reference to depthcharge

Refactor Kernel Verification

Currently, vboot1 assumes OS kernel on sector-based storage with GPT

Want to support other storage types

- Raw NAND
- SPI flash

What are we doing?

- Refactor kernel verification API in vboot_reference to be finer-grained
- Move higher-level logic to depthcharge
- Also purge the terrible coding style we used for vboot1

Verified CBFS

Currently, RW firmware inefficiently verified as one big blob

- Want RAM init code in RW
- But before RAM init, not enough RAM to hold the RW firmware
- Don't want to load display assets or component firmware unless needed

What we're doing (pgeorgi@, martinroth@, adurbin@, jwerner@, et al.)

- Vboot only verifies RW CBFS metadata, including file data digests
- CBFS verifies digest of each file's data when loading that data

Better Support for Owner (vs. User) Control

Original model: WP screw. Anyone with the device for 5 minutes is the owner.

Schools don't like this. Students have screwdrivers too.

NEW: Firmware Management Parameters (just checked into vboot_reference)

- Another lockable NV memory space
- Flag which tells RW firmware not to boot in developer OS mode
  - Set/removed by enterprise enrollment
  - Not present on normal individually-purchased devices
- Bonus for OS developers: expected kernel key hash
  - RW firmware checks OS kernel data key in developer OS mode
  - Now safe to allow booting from USB in developer OS mode

Protecting Content Keys

On some SoC, DRM keys left exposed by mask ROM

- Entire RO firmware (Coreboot) verified by boot ROM, eFused keys
- vboot_reference verifies root/recovery key matches expected hash from RO
- If not, clears DRM keys before (always) continuing to RW firmware

Better: A small landing stage before verstage, verified by boot ROM, eFused keys

- Check verstage and root/recovery keys against hash from landing stage
- Clear DRM keys on mismatch as above.
- Does nothing else! Always continues to verstage.
- Easy to inspect landing stage binary and make sure that's all it does.

Other Components Have Upgradeable Firmware

EC, USB-C PD chips use "EC software sync" today. SATA, eMMC, NVMe soon?

- Component has RO, RW firmware
- Depthcharge verifies component RW firmware hash vs. CBFS
- ...and updates component as needed before jumping to OS
- Requires support from component firmware/hardware

USB-C PD dead battery case is interesting

- Don't want PD chip talking to world from RO firmware
- But until it does, can't power up board all the way
- Need to boot SoC in low-power mode, just enough to verify components

The Goal

Coreboot + Depthcharge have full support for verified boot by default

- As simple as setting a build flag and running a script to generate keys
- Flexible enough to customize for a wide range of form factors and use cases
- No excuse for device manufacturers not to have verified boot

Support for open source developers

- Of both OS and firmware
- Transparency matters - need to be able to verify what the firmware does
- Security should not come at the expense of open source development on real devices

How You Can Help

How You Can Help

Look at what we've done

- More eyeballs on verified boot code is a good thing
- Question our assumptions

Contribute needed functionality (see what's coming, above).

Think about security, particularly in the early boot stages (verstage, romstage)

Keep being awesome

- Verified boot without coreboot is just a reference library
- Coreboot everywhere means verified boot can be everywhere too

Thank You!

...Questions?

Backup

Design Docs

http://dev.chromium.org/chromium-os

- Firmware design docs are in this section: http://www.chromium.org/chromium-os/chromiumos-design-docs
- Many docs are out of date, but still helpful to show our thought processes