Verified Boot in Chrome OS
and
how to make it work for you

Simon Glass
Embedded Linux Conference Europe
Edinburgh, October 2013
Agenda

- Introduction
- Chrome OS
  - Verified Boot
- Requirements
- Technology
- U-Boot + Linux Verified Boot
- Demos
- Doing More
- Resources
Introduction

- **Me**
  - ARM technology since 1987
    - ARM in UK and US
    - Bluewater Systems (NZ ARM/Linux Electronics)
  - Google Chrome OS (first ARM laptop)

- **Some professional Interests**
  - Great ARM devices
  - Open Source Software
What is Chrome OS?

Google Chrome OS is an upcoming Linux-based, open source operating system designed by Google to work exclusively with web applications. Announced on July 7, 2009, Chrome OS is set to have a publicly available stable release in the late fall of 2010. 
http://en.wikipedia.org/wiki/Chrome_os

Chromium OS - The Chromium Projects
www.chromium.org/chromium-os
Chromium OS. Chromium OS is an open-source project that aims to build an operating system that provides a fast, simple, and ... What is Google Chrome OS?

James Cook +1'd this
Converging forces

The migration to the cloud

The HTML 5 juggernaut
Chromebook

- Speed
- Simplicity
- Security
Integrated and streamlined

PC

Chromebook
Simplicity

Familiar UI
Same experience everywhere
Zero Maintenance
Forever new
"Rust" Proof
Seamless sharing
Standard PC: Security as an afterthought
Security for the internet age

Current Operating Systems

- Apps have the same privileges and power as you

Chrome OS

- Web apps and offline apps
- The OS doesn't trust any of them
- Keep them isolated and sandboxed
Chrome OS' defense in depth

- Small list of known executables
  - Signed and verified before each use

- Run in secured sandboxes
  - Chroot, Namespaces
  - Toolchain, Stack protection

- File system is locked down
  - Read-only root file system
  - User data encryption

- Automatic updates for the entire OS
  - Nothing is ever perfect.
  - It's not the user's job to keep it secure.
Why Verified Boot?

- Reduced risk of malware
- Keeps users safe
- Permits safe software updates in the field
- Known software on device

- Verified Boot does not mean the user needs to be locked out
  - E.g. See Chrome OS ‘dev mode’
Requirements of Verified Boot

- Root of trust (static in our case)
- Every byte of code/data loaded is verified
  - Can use a sandbox where this is impractical
- Prior state must be fully validated
- Security holes plugged
- Upgradeable software
- Rollback protection
Technology

- Hashing
- Public key cryptography
- Trusted Platform Module (TPM)
- Root of trust
Hashing of binary images

- Reducing an image down to a very small data block (‘digest’)
- Two images can be considered:
  - Identical if their digests are the same
  - Different if their digests differ
- For a good hashing algorithm:
  - Changing just one bit in the image should completely change the digest
  - ‘Collision resistant’ - need to try sqrt(2^n) images
  - Infeasible to modify an image to obtain a certain digest
- Common hashing algorithms are:
  - SHA1 - 24 byte digest
  - SHA256 - 32 byte digest
Public key cryptography

- Create a key pair to sign a hash, and later to verify its signature
  - One key is ‘private’ – used to sign images and kept secret
  - Other key is ‘public’ – widely broadcast without affecting security
- Two keys are mathematically related
  - Data encrypted by one can be decrypted by the other
- With the public key we can verify that a hash was signed by the associated private key
- Common public key algorithms are RSA and ECC
  - RSA 2048 bits is considered strong
**Trusted Platform Module (TPM)**

- **Security chip**
  - Each device has a unique RSA private key
  - Can store keys, roll-back counters
  - Random number and key generation

- **Commonly used on high-end laptops, or with a plug-in PCB**
  - Typically I2C or LPC bus
  - Many ARM devices make use of TrustZone instead of a discrete TPM
  - Requires additional software

- **TPM can check software and configuration at start-up**
  - Hash each new chunk before using it
  - Pass the hash to the TPM for checking
Root of trust

- Simple ‘static root of trust’
  - Initial code is assumed to be trusted
  - Boot ROM, U-Boot
- Can be stored in read-only memory
  - Or signed so that SoC can verify it
- Root stage holds keys for checking later stages
- From there we can load each stage of boot
  - Verify each as we go, using keys provided by the previous stage
Verified boot in Chrome OS

- ‘Verified boot’ is the term used in Chrome OS
- **Firmware**
  - U-Boot and verified boot library (also Coreboot on x86)
- **Kernel**
  - dm-verity
  - A few drivers
- **User space**
  - Firmware interface, update
  - Chrome OS update
- **Other**
  - Signer
  - Other utilities
Verified boot flow - firmware

- Firmware, kernel and root disk all have an A and a B
Verified boot components - firmware

- U-Boot 2013.06
  - Main source base
  - Drivers and subsystems
  - Vboot integration layer in cros/ subdirectory
  - Full source code here [http://goo.gl/N6rhik](http://goo.gl/N6rhik)

- Vboot library
  - Hashing
  - RSA / signature checking
  - Verified boot ‘logic flow’
  - TPM library (only used for roll-back counters)
  - Full source code here [http://goo.gl/dTbkLs](http://goo.gl/dTbkLs)
Verified Boot Components - Kernel

- dm-verity merged to Linux in 2012

- cryptohome (not really verified boot)
Verified Boot Components - User space

- crossystem
  - Allows access to firmware settings
  - Allows signals to be sent to firmware for next boot
- update_engine
  - Update the partition we did not boot
- chromeos_firmwareupdate
  - Update the firmware we did not boot

- Also a few tools
  - Signer
  - cros_bundle_firmware
  - Image utilities
Chromium OS is Open Source

http://git.chromium.org/gitweb/
chromium-review.googlesource.com
DIY Verified Boot

- Can I implement verified boot on my own platform?
  - Yes
- Do I need UEFI?
  - No

- U-Boot
  - Use FIT if you don’t already
  - Imager signer is the trusty mkimage
  - Continue to use bootm
  - Will go through this in some detail

- Linux
  - dm-verity is upstream

- Firmware<->user space layer
  - Roll your own
Introduction to FIT

/ {
    description = "Simple kernel / FDT configuration (.its file)";

    images {
        kernel@1 {
            data = /incbin/("../vmlinuz-3.8.0");
            kernel-version = <1>;
            hash@1 {
                algo = "sha1";
            }
        }
        fdt@1 {
            description = "snow";
            data = /incbin/("exynos5250-snow.dtb");
            type = "flat_dt";
            arch = "arm";
        }
    }

    configurations {
        default = "conf@1";
        conf@1 {
            kernel = "kernel@1";
            fdt = "fdt@1";
        }
    }
};

http://goo.gl/a09ymG
Adding a signature to a FIT

/ {
  description = "Simple kernel / FDT configuration";

  images {
    kernel@1 {
      data = /incbin/("../vmlinuz-3.8.0");
      kernel-version = <1>;
      signature@1 {
        algo = "sha1,rsa2048";
        key-name-hint = "dev";
      }
    }
  }

  fdt@1 {
    description = "snow";
    data = /incbin/("exynos5250-snow.dtb");
    type = "flat_dt";
    arch = "arm";
  }

  configurations {
    default = "conf@1";
    conf@1 {
      kernel = "kernel@1";
      fdt = "fdt@1";
    }
  }
};
Use bootm as normal

## Loading kernel from FIT Image at 00000100 ...

Using 'conf@1' configuration
Trying 'kernel@1' kernel subimage
  Description: unavailable
  Type: Kernel Image (no loading done)
  Compression: uncompressed
  Data Start: 0x000001c8
  Data Size: 5000 Bytes = 4.9 KiB

Verifying Hash Integrity ... sha1,rsa2048:dev+ OK

## Loading fdt from FIT Image at 00000100 ...

Using 'conf@1' configuration
Trying 'fdt@1' fdt subimage
  Description: snow
  Type: Flat Device Tree
  Compression: uncompressed
  Data Start: 0x0000164c
  Data Size: 4245 Bytes = 4.1 KiB
  Architecture: Sandbox

Verifying Hash Integrity ... sha1,rsa2048:dev+ OK
Booting using the fdt blob at 0x00164c
XIP Kernel Image (no loading done) ... OK
Signing images using mkimage

```
mkimage -f test.its -k ../keys -K out/u-boot.dtb -r test.fit
```

- `-k`  Key directory
- `-K`  Output FDT for public keys
- `-r`  Require verification of all keys
How signing works

- image.its
- kernels, FDTs, ramdisks...

New Signing Flow

- Keys
- mkimage
- image.fit
- signed image. fit
- u-boot.dtb with public keys
Signed image.fit

images {
    kernel@1 {
        data = <3.4MB of stuff>;
        signature@1 {
            algo = "sha1,rsa2048";
            key-name-hint = "dev";
            timestamp = <0x50e4b667>;
            signer-version = "2013.01";
            signer-name = "mkimage";
            value = <0x32e48cf4 0xa72b7504 0xe805aeff 0xe1af8b2e8 0x24c5313f
                    0xb4b3d41b 0x3cf03e60 0xe3e103a1c ... 
                    0xc293395e 0x06cfa9e5 0x1cda41e1 0xb0a10e97 0xa92d8d61>;
        }
    }
    fdt@1 {
        description = "snow";
        data = <12KB of stuff>;
        signature@1 {
            algo = "sha1,rsa2048";
            key-name-hint = "dev";
            timestamp = <0x50e4b667>;
            signer-version = "2013.01";
            signer-name = "mkimage";
            value = <0x32e48cf4 0xa72b7504 0xe805aeff 0xe1af8b2e8 0x24c5313f
                    0xb4b3d41b 0x3cf03e60 0xe3e103a1c ... 
                    0xc293395e 0x06cfa9e5 0x1cda41e1 0xb0a10e97 0xa92d8d61>;
        }
    }
};
u-boot.dtb with public keys

/ {
    model = "Google Link";
    compatible = "google,link", "intel,celeron-ivybridge";
    signature {
        key-dev {
            algo = "sha1,rsa2048";
            required;
            rsa,r-squared = <0x0a1ed909 0xf564a4e6 0x539e6791 0x9d9b4a7e 0x2a7788cf
            0x89f9cb7a 0x7cd7a2c3 0xdb02b925 0x97f6cd15 0x76c86fb0 0x16b7b120 0x5825dc2c ...  
            0x0e9e736a 0x852372bd 0x13a08e33>;
            rsa,modulus = <0xc1ad79b6 0x52ef561b 0x2c8b2a54 0x13436fa4 0xcabcce1b9
            0x64c6e1c8 0xbfbeb9a2 0x1e3d974c 0x14a67ada 0x4ecc3648 0xa7fee936 0xb53cc0a8 ...  
            0xabe4f37f 0xdcc15a79 0xfcd530a5>;
            rsa,n0-inverse = <0x75a89dbf>;
            rsa,num-bits = <0x00000800>;
            key-name-hint = "dev";
        }
    }
    ...
In-place signing

- FIT is a very flexible format
- No need to write the signature to a separate place/file
  - Just update the FIT
  - Multiple signatures can be added later without affecting previous signing
- Hashing algorithm supports hashing portions of the FIT
Signing configurations

/ {
  images {
    kernel@1 {
      data = /incbin/("test-kernel.bin");
      type = "kernel_noload";
      hash@1 {
        algo = "sha1";
      }
    };
    fdt@1 {
      description = "snow";
      data = /incbin/("sandbox-kernel.dtb");
      hash@1 {
        algo = "sha1";
      }
    };
  };
  configurations {
    conf@1 {
      kernel = "kernel@1";
      fdt = "fdt@1";
      signature@1 {
        algo = "sha1,rsa2048";
        key-name-hint = "dev";
        sign-images = "fdt", "kernel";
      }
    };
  };
};
Using bootm with configuration signing

## Loading kernel from FIT Image at 00000100 ...
Using 'conf@1' configuration
Verifying Hash Integrity ... sha1,rsa2048:dev+ OK
Trying 'kernel@1' kernel subimage
  Description: unavailable
  Type: Kernel Image (no loading done)
  Compression: uncompressed
  Data Start: 0x000001c8
  Data Size: 5000 Bytes = 4.9 KiB
Verifying Hash Integrity ... sha1+ OK
## Loading fdt from FIT Image at 00000100 ...
Using 'conf@1' configuration
Trying 'fdt@1' fdt subimage
  Description: snow
  Type: Flat Device Tree
  Compression: uncompressed
  Data Start: 0x0000164c
  Data Size: 4245 Bytes = 4.1 KiB
  Architecture: Sandbox
Verifying Hash Integrity ... sha1+ OK
Booting using the fdt blob at 0x00164c
XIP Kernel Image (no loading done) ... OK
U-Boot code size

- OpenSSL is only used in mkimage
  - Produces pre-processed public key parameters for U-Boot run-time
  - Modulus (n), r-squared, n0-inverse and num-bits
- U-Boot simply has to do exponential mod n
- Code size is very efficient
  - RSA verification code is only 2149 bytes (Thumb 2)
- Entire RSA FIT code adds 6.2KB code/data
  - If you don’t already use FIT, then that adds an additional 20KB
  - Both FIT and RSA add only ~12.5KB to gzip-compressed U-Boot size

$ ./tools/buildman/buildman -b talk snow  -Ss
Summary of 3 commits for 1 boards (1 thread, 32 jobs per thread)
01: Merge branch 'master' of git://git.denx.de/u-boot-mmc
  arm: (for 1/1 boards)  all +20437.0  bss +60.0  data +504.0  rodata +1953.0  text +17920.0
02: enable fit
  arm: (for 1/1 boards)  all +6337.0  bss -40.0  data +16.0  rodata +697.0  text +5664.0
03: Enable verified boot
  arm: (for 1/1 boards)  all +6337.0  bss -40.0  data +16.0  rodata +697.0  text +5664.0
U-Boot performance

- Time to check FIT configuration with 2048-bit RSA signature
  - <6ms on Beaglebone (1GHz Cortex-A8)
  - Note: if you care about performance, turn on the cache
    - With cache off it is 290ms
Nice Properties of U-Boot’s verified boot

- Small 6.2KB code on Thumb 2
- Faster - 6ms on 1GHz Cortex-A8
- Uses existing FIT format
  - No need for multiple files - data and signatures are in the FIT
- Can sign and re-sign existing images
  - Signing uses the existing mkimage tool
- No new boot flow - works with existing scripts that use bootm
- Supports multiple stages, sub-keys, etc.
Using bootm

- Verified boot still uses bootm
  - No change in syntax
- Signature verification plumbed into existing image-checking code
- Image check just sits along existing hash/CRC checking
- Configuration check happens before this
  - As soon as the configuration is selected
Demo time
Doing more

- Accelerated hashing
  - U-Boot and Linux have a framework
- Auto-update
- Recovery mode
- Other root of trust options
- Performance
- TPM for roll-back
- Trusted boot using TPM extend
Conclusion

- Verified boot can be enabled in most embedded systems
  - Main new requirement is a verified root of trust
- Available in mainline U-Boot
  - Adds just 6.2KB code and a small run-time penalty
- U-Boot TPM library provides roll-back protection
  - ‘Extend’ functionality also available if desired
- Read-only root filesystem can be protected with dm-verity
  - Chrome OS uses this approach
Thank you

- **U-Boot verified boot**
  - [http://git.denx.de/cgi-bin/gitweb.cgi?p=u-boot.git;a=blob;f=doc/uImage.FIT/verified-boot.txt](http://git.denx.de/cgi-bin/gitweb.cgi?p=u-boot.git;a=blob;f=doc/uImage.FIT/verified-boot.txt)

- **dm-verity**
  - [https://lwn.net/Articles/459420/](https://lwn.net/Articles/459420/)
  - [https://code.google.com/p/cryptsetup/wiki/DMVerity](https://code.google.com/p/cryptsetup/wiki/DMVerity)

- **Chrome OS**
  - [http://www.chromium.org/chromium-os/chromiumos-design-docs](http://www.chromium.org/chromium-os/chromiumos-design-docs)

- **Other ideas:**
  - [https://github.com/theopolis/sboot](https://github.com/theopolis/sboot)

- **Email me** [sjg@chromium.org](mailto:sjg@chromium.org)
  - [cc u-boot@lists.denx.de](mailto:cc@u-boot@lists.denx.de)
Additional slides
U-Boot’s TPM Support

- **TPM library**
  - `tpm_startup()`
  - `tpm_self_test_full()`
  - `tpm_nv_define_space()`
    - `tpm_nv_read_value()`
    - `tpm_nv_write_value()`
  - `tpm_extend()`
  - `tpm_oiap()`...

- **Drivers for common TPMs**
  - Infineon (I2C and LPC), Atmel, STM

- **‘tpm’ command**
  - Provides full access to TPM library for scripts