Lollipop MR1 Verified Boot

Andrew Boie

Open Source Technology Center

Intel Corporation

Agenda

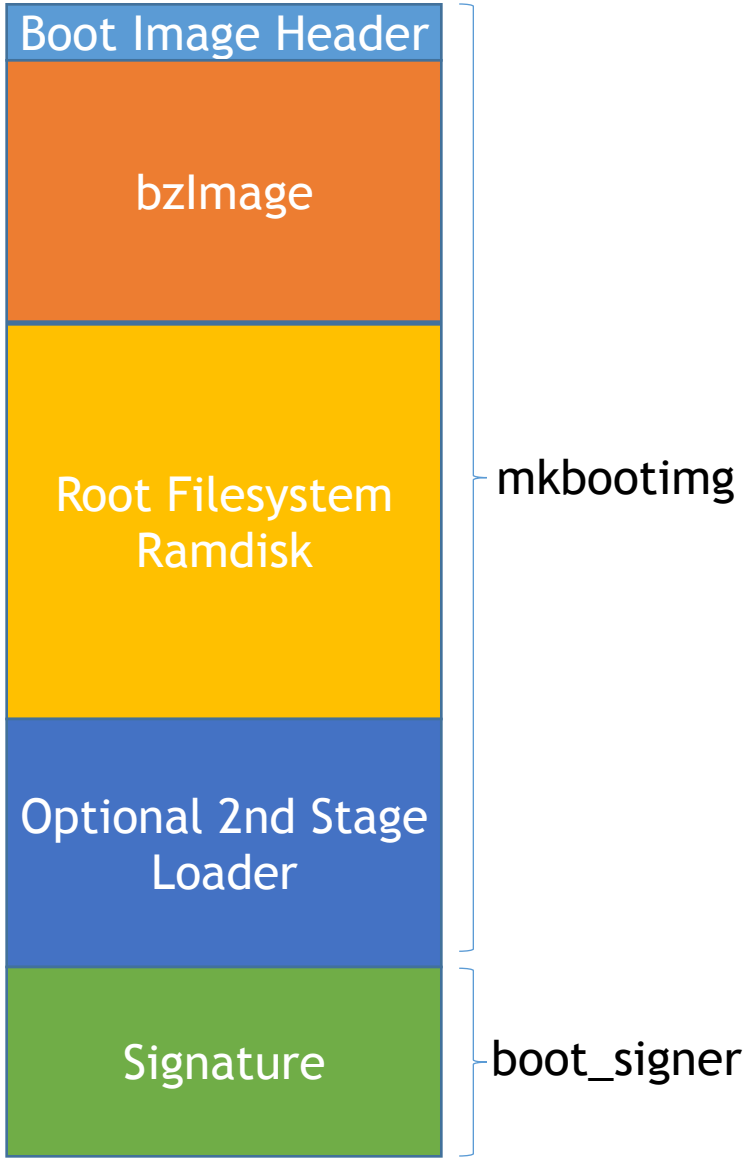
- What is Verified Boot?
- Description of Verified Boot Components
- Q&A

What Is Verified Boot?

- Verified Boot establishes a chain of trust from the bootloader to the system image
- Components verified:
  - Boot / Recovery images
    - Each containing kernel w/command line, ramdisk, optional 2nd-stage bootloader
    - Signature block appended to the end of the boot image
    - Verified by the bootloader using a keystore
      - OEM-signed keystore ships with device
      - User-supplied keystore may be enrolled via Fastboot if device is unlocked first
  - System image (and also Vendor image if present)
    - Protected by Linux dm-verity
    - Signing key stored in boot ramdisk
    - Incremental updates re-implemented to work on a per-block level
- The end user is empowered to unlock the device and flash boot/system/recovery images signed and verified with their own generated key
- Terminology used is sometimes inconsistent, "verified boot" or "verity" in the code can variously apply to verification of boot images, system images, or both
- The integrity of the bootloader itself is out of scope

Signed Boot Images

- Boot images created by mkbootimg in the Android build
  - system/core/mkbootimg
  - Concatenates a header, bzimage, ramdisk, optional 2nd-stage loader image into a single binary blob
  - Small C program
  - Header contains total size of boot image (without signature data), sizes/offsets of sub-components
- New tool in the build system: boot_signer
  - Code is in system/extras/verity/
  - Run by the build system immediately after running mkbootimg
    - Appends signature to the boot image
    - Default key used is “verity” key under build/target/product/security
    - See sign_target_files_apks section for details on production re-signing
  - Implemented in Java using BouncyCastle APIs
- Boot images are written as raw data to dedicated partitions
  - "boot" for main Android Boot Image
  - "recovery" for OTA Recovery Console



Boot Signature Format

- DER Encoded ASN.1 message data appended to the end of the boot image
- No way to tell from the boot image header whether the image is signed or not
  - In our loader, we read 4096 bytes of additional data beyond the size of the boot image as reported by the header
  - Extra data passed to OpenSSL ASN.1 decoding routines
  - Header changes likely due to backward compatibility before signing was introduced -- has implications for incremental OTA updates
- Signature is computed by hashing two components
  - The boot image itself
  - The authenticatedAttributes ASN.1 data (in DER form) inside the AndroidVerifiedBootSignature message
    - target - Boot image type (either "boot" or "recovery")
    - length - Boot image size, should match the header
- algorithmIdentifier block indicates how to hash/verify images
  - boot_signer currently only supports SHA1 or SHA256 with RSA Encryption
- X509 Certificate used to sign the boot image included
  - Included certificate for reference only
  - In production, the public key in the certificate must be contained in the **keystore** managed by the bootloader

```
AndroidVerifiedBootSignature DEFINITIONS ::=
BEGIN
    formatVersion ::= INTEGER
    certificate ::= Certificate
    algorithmIdentifier ::= SEQUENCE {
        algorithm OBJECT IDENTIFIER,
        parameters ANY DEFINED BY algorithm OPTIONAL
    }
    authenticatedAttributes ::= SEQUENCE {
        target CHARACTER STRING,
        length INTEGER
    }
    signature ::= OCTET STRING
END
```

Keystores

- A keystore is a signed collection of RSA key objects, each with an associated AlgorithmIdentifier
- The FormatVersion and KeyBag fields are collectively referred to as the “inner keystore”
- Inner Keystore data signed with an AndroidVerifiedBootSignature
- Given a full DER keystore message, some adjustments must be made to the enclosing SEQUENCE data to create a valid Inner Keystore message

```
AndroidVerifiedBootKeystore DEFINITIONS ::=
BEGIN
    FormatVersion ::= INTEGER                                   Inner Keystore
    KeyBag ::= SEQUENCE {
        Key  ::= SEQUENCE {
            AlgorithmIdentifier  ::=  SEQUENCE {
                algorithm OBJECT IDENTIFIER,
                parameters ANY DEFINED BY algorithm OPTIONAL
            }
            KeyMaterial ::= RSAPublicKey
        }
    }
    Signature ::= AndroidVerifiedBootSignature
END
```

Keystores (Continued)

- Verified boot devices ship with an “OEM Keystore” which is built into the system and signed by a key managed by the OEM
- keystore_signer tool in system/extras/verity creates keystore binaries
  - Implemented with Java BouncyCastle APIs
- On an unlocked device, the end user may enroll their own keystore binary via the “fastboot flash keystore” command
  - Typical scenario: user unlocks device, enrolls new keystore, flashes custom boot/recovery images, sets bootloader to locked or verified state
  - More detail on bootloader states later
- Upon boot, the loader checks if a user keystore is present and will attempt to verify it using the OEM key if the loader isn’t unlocked
  - If the keystore signature doesn’t verify, the user will be may be warned boot before proceeding to use that keystore to verify images
- Regardless of whether the OEM keystore or the user-supplied keystore used, the selected keystore is used to verify the boot or recovery images

Fastboot

- Despite its name, simple protocol for communicating with the device over USB
- Implemented in the bootloader on the device
- Client:
  - system/core/fastboot
- Allows issuing commands, flashing images
- Not really any facilities for getting data off the device other than simple text strings

Bootloader Lock States

- A verified boot capable loader has 3 different security states
  - Locked, Verified, Unlocked
- State transitions done via Fastboot commands
- Any state transition should erase all user data
  - Defense against attackers with physical access to the device, so that they cannot flash a hacked boot image and access userdata contents
  - /data partition zeroed out; on next boot, fs_mgr will see this and initiate reboot into Recovery to create a filesystem
- Any state transition should require the user to physically confirm with the device's buttons that the state transition is actually desired
  - Defense against malware which could otherwise surreptitiously issue ADB and Fastboot commands to unlock the device without user's knowledge
- Setting device to "unlocked" state requires option change in Settings app Developer Options
  - Not enabled by default, user with proximate access must get past the lock screen to change this
  - More details later under Persistent Data Block slides
- Specific commands may vary across implementations
  - In Kernelflinger: "fastboot oem {lock|unlock|verified}"

Bootloader States (Continued)

- “Locked” state
  - Devices ship to the end user in this state
  - No images may be flashed or erased with Fastboot
  - Boot/Recovery images verified by the bootloader using enrolled keystore
- “Verified” state
  - A subset of targets/partitions may be flashed or erased with Fastboot
    - bootloader, boot, system, oem, vendor, recovery, cache, userdata
  - Boot/Recovery images verified by the bootloader using enrolled keystore
  - Good state for running user-built Android images or third-party images like Cyanogenmod
    - Device is still secure, may have to deal with a prompt at boot if keystore isn’t signed by OEM
- “Unlocked” state
  - Device may not be unlocked if flag in Persistent Data Block is not set via Settings app
  - All Fastboot commands available
  - User keystore may be enrolled or erased
    - Erasing keystore causes loader to fall back to OEM Keystore for image verification
    - “fastboot flash keystore <path to keystore binary>” or “fastboot erase keystore”
  - Unlocked devices do not verify boot or recovery images
  - User may be warned at boot that the device is unlocked and requires physical interaction to proceed

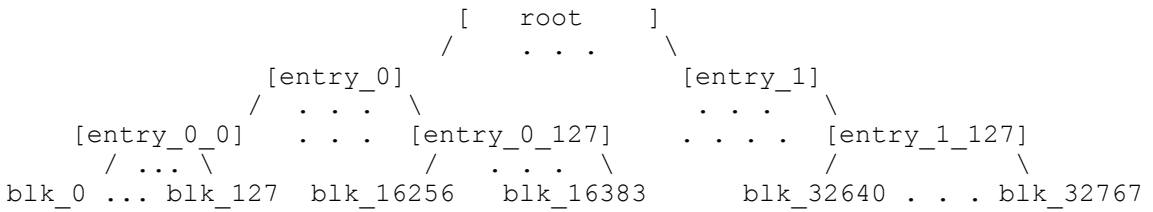
Bootloader Boot States

- Device's security level expressed as colors
  - GREEN - Device is locked or verified, keystore verified by OEM key, selected boot image verified by the keystore
  - YELLOW - Device is locked or verified, keystore NOT verified by OEM key, but selected boot image verified by the keystore
  - ORANGE - Device is unlocked, boot image signature not checked
  - RED - Device is locked or verified, boot image NOT able to be verified, boot cannot continue
- Affects boot policy in Kernelflinger
  - The end user is presented with a warning UI and must acknowledge with a button press for YELLOW or ORANGE state to continue to boot
  - RED state cannot boot the device, only option is to halt or enter Fastboot
- Reported in Fastboot UI and also Android property in Kernelflinger

Persistent Data Block (PDB)

- Implemented as a small “persistent” partition in the fstab
  - Raw data, does not contain a filesystem
  - The very last byte in the partition stores whether unlocking is enabled
    - Must contain value 0x01 or unlocking is forbidden
- Not all methods of doing a Master Clear are the same
  - A Master Clear initiated by the Settings app will zero the persistent partition along with user data
    - Considered trusted as user would have to get past lock screen to do this
  - Erasing userdata from Recovery Console or Fastboot in “verified” state does not allow this
- Relevant code
  - frameworks/base/services/core/java/com/android/server/PersistentDataBlockService.java
  - packages/apps/Settings/src/com/android/settings/MasterClearConfirm.java
  - packages/apps/Settings/src/com/android/settings/Utils.java
- Devices with Google Mobile Services store additional user data in the PDB
  - Untrusted resets will require Google account sign-in of an account that has been already used by the device, before the device can be used again
  - Discourages thieves
- All bets are off if the device can be rooted

dm-verity



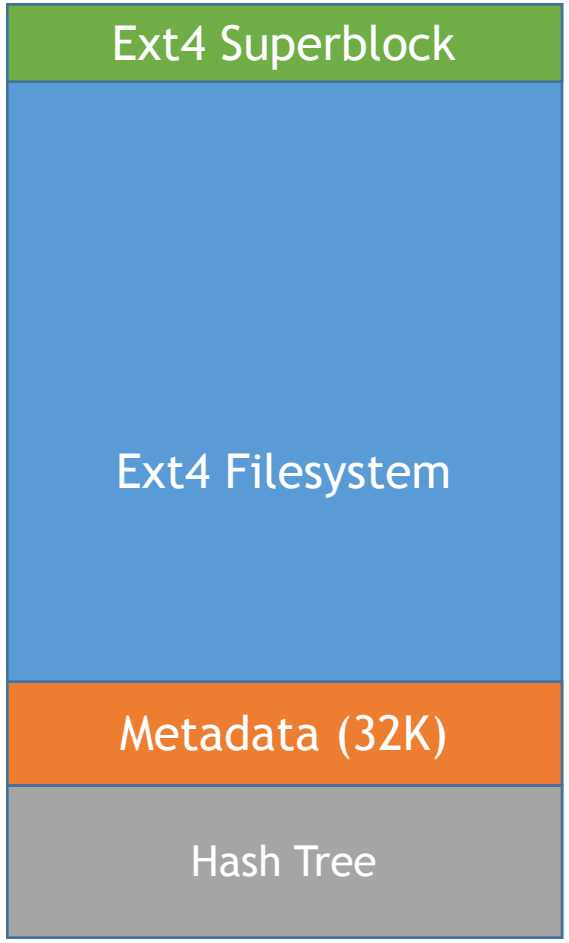
- Linux kernel feature
  - http://lwn.net/Articles/459420/
  - https://code.google.com/p/cryptsetup/wiki/DMVerity
  - https://www.kernel.org/doc/Documentation/device-mapper/verity.txt
- Only supported in Android for ext4 filesystems
- Enforces a specific binary state of the /system and /vendor partitions
  - Uses a cryptographic hash tree
    - Leaf Nodes: every 4K block in the partition has a SHA256 hash of all the data in it
    - Intermediate Nodes: Contains hash of leaf nodes below it
    - At the top there is a root hash node which represents the entire disk
    - On-demand verification of hashes during disk access, verified up to the root node of the tree
  - Root hash is signed with a certificate stored in the boot image ramdisk
    - We trust this certificate since it is verified by the bootloader
  - Done entirely in software, no hardware support needed

dm-verity (continued)

- Creation and signing of hashes handled by Android Build System
  - Defaults to “verity” key in build/target/product/security
  - See section on sign_target_files_apks for details on production re-signing
  - Everything you need is provided by AOSP
- Implications
  - If enabled, dm-verity enforced for user & userdebug builds
  - Significant changes made to the OTA system to support incremental updates
    - Now done at a block level instead of per-file basis
    - Details about this in my other presentation
  - System/Vendor partitions can never be changed or mounted read-write
    - Simply mounting changes the superblock!
  - Userdebug builds support “adb disable-verity” command to allow for system image modification
    - ‘adb sync’, etc
    - Breaks incremental OTA updates from currently installed software, device must be re-flashed or use full image update before they will work again

dm-verity Metadata & Hash Trees

- Metadata
  - Magic number (0xb001b001) (or 0x46464f56 if "adb disable-verity" run)
  - Version (0)
  - Verity Table signature
  - Verity Table length
  - Verity Table passed to DM_TABLE_LOAD ioctl()
    - Contains block device, block sizes, number of data blocks, root hash, salt, device and offset of verity hash tree -- see kernel verity.txt for more information
    - Signature verified by fs_mgr before passing to the kernel using certificate in ramdisk
- Verity Hash Tree
  - Contains all the leaf node and intermediate node hashes
  - Used directly by dm-verity code in the kernel, location passed in via Verity Table
- Relevant code
  - build/tools/releasetools/build_image.py now handles overall creation of dm-verity signed filesystem images
    - Composed of the filesystem itself + metadata blob + verity hash tree
  - system/extras/verity/build_verity_metadata.py creates metadata blob
  - system/extras/verity/build_verity_tree.cpp creates verity hash tree and computes root hash & salt



Production Re-signing Process

- By default, all APKs, OTA packages, boot and filesystem images produced by the build are signed with testing keys
  - CTS test exists to check and fail if these test keys are in use
  - `build/target/product/security`
- OTA updates and factory provisioning images are created using a Target Files Package (TFP)
  - ZIP file containing all elements of the build
- sign_target_files_apks tool re-signs everything in the TFP with production keys supplied by the user
  - Regenerate boot images
  - Regenerate signed filesystem images
  - Replace on-device keys in various locations
    - dm-verity key located in root ramdisk
- Bootloader OEM keystore out of scope of this mechanism

Bootloader Implementation Considerations

- Need to implement confirmation UX with physical key input for various scenarios
  - Improperly signed boot or recovery images
  - Improperly signed User keystore
  - Device in unlocked state
  - Confirm changing device state between locked, unlocked, verified
- Need crypto code which can parse DER ASN.1 messages, DER X.509 certs, SHA256 hashing, RSA verification
  - Don't write your own crypto code
  - For EFI Kernelflinger we used EFI-built OpenSSL library from UEFI Shim Project
- Need nonvolatile place to store Fastboot state information
  - Ideally store Fastboot lock state, user keystore in area not accessible to running OS
  - For EFI devices that can do Fastboot in Boot Services context, we use EFI variables with Boot Services access only
- We relax some security policies in eng/userdebug loaders to make life less annoying for development
  - Persistent Data Block ignored, device always unlockable
  - State transition UX skipped to assist with automation
  - Verity key used to verify boot images is the default AOSP verity key
  - All security turned off in Eng builds, loader always acts like it is unlocked with no UX
  - Some policies needs to be bypassed in a trusted way during initial device provisioning steps and also RMA process

Configuration Prerequisites for Verified Boot

- Write a bootloader!
  - 01.org distributes Kernelflinger which implements Verified Boot for EFI devices
- Product Makefile:
  - $(call inherit-product,build/target/product/verity.mk)
    - Enables additional steps in build system to sign boot images, etc
  - Set PRODUCT_SYSTEM_VERITY_PARTITION (and also PRODUCT_VENDOR_VERITY_PARTITION if used) to the device nodes corresponding to these partitions
    - Needed by build_image.py tool
  - PRODUCT_COPY_FILES += frameworks/native/data/etc/android.software.verified_boot.xml:system/etc/permissions/android.software.verified_boot.xml
    - Tells PackageManager that the system supports Verified Boot, which may be required for some apps to be allowed on the device
- fstab
  - Add “verify” to the options for the /system (and also /vendor if applicable) line(s)

Q&A?