CS 1657
Privacy in the Electronic Society

William Garrison
bill@cs.pitt.edu
6311 Sennott Square
http://cs.pitt.edu/~bill/1657

13: Secure boot
Secure boot: Protecting devices from software modifications

What is secure boot?
• Pros/cons?

What implementations exist?
• We’ll look at UEFI in isolation, then Android/Qualcomm secure boot overall

How can we recover from errors?
• Can’t distinguish corruption vs. malicious modification
• Avoid amplifying errors

What about rollback attacks?
• Old versions are correctly signed. Can we forbid them anyway?
What is secure/verified/trusted boot?

This idea has come up several times during the semester

• Supply-chain attacks
• Limits of cryptography
• Protecting disk encryption bootloader

Main idea: Verify a signed hash of system software before running it

• Possibly in layers, e.g., bootloader verifies kernel, kernel verifies system files
• Thus, software (original and updated) must be signed by supplier
  • Who?

What are the potential benefits and drawbacks?
In 1998, Intel began developing a replacement for BIOS

- Too limited for advanced boot features in servers
- Intel Boot Initiative became Extensible Firmware Interface (EFI)
- In 2005, control was turned over to United EFI Forum as Unified EFI (UEFI)
- Cryptographic controls added in v2.1 in 2007

Secure boot requires OS loaders to be signed

- When shipped, “setup” mode allows a platform key (PK) to be stored
- Once in “user” mode, loaders must be signed with PK to boot
- In “custom” mode, additional keys can be added by the user
- What attacks would this protect against?
In Android, there are several layers of verification

- **TrustZone** runs in parallel to verify
- **Primary Bootloader** verifies signature on XBL
- **eXtensible Bootloader** verifies signature on ABL
- **Android Bootloader** implements firmware services, verifies boot image
- **Linux kernel and Android services** take over from here
After the ABL passed control to Android, kernel services continually verify system data

Root partition is read-only, userdata goes in a separate partition

• Like what?
• Thus, system partition can be assumed not to change

Assuming (PBL→XBL→ABL→boot image) verification chain has passed, kernel is trusted

• Applications may not be (i.e., may lie)!
• Kernel hashes entire blocks of system data as they are read
  • SHA256, typically 4 KiB blocks
• Signature over hash included in boot image
• On mismatch, respond with I/O error
dm-verity, the kernel service that verifies blocks of system data, uses a Merkle tree.

Typically, <30 MB for signed hash tree.
Unfortunately, this can amplify benign corruptions

A small error can now cause an entire block to fail to verify
    • Can we make this more robust?

Error-correcting codes (ECC) use redundancy to recover from missing or corrupted data
    • Pioneered by Richard Hamming in 1940–50s
    • QR codes, CDs, DVDs, etc. use Reed-Solomon ECC (1960)
    • RS($n$, $k$) uses $n$ space to represent $k$ data, with $t = n - k$ redundancy
        • Up to $[t/2]$ corruptions or $t$ erasures can be corrected by inferring from redundant bits
        • e.g., RS(255, 223) stores 223 bytes of data in 255 bytes (32 bytes of redundancy), can recover 16 byte error or 32 byte erasure
Unfortunately, corruptions usually happen at the block level

How can we use ECC to mitigate corruptions?
  • Maybe even for many consecutive blocks?

Interleaving allows for RS “blocks” to be distributed widely across the device
  • **Idea:** Map each byte in a block to a separate RS code
    • Each code covers $n$ bytes across $n$ blocks (1 byte in each block)
  • With naive interleaving, device can recover from corruption of up to $\frac{255-k}{2}$ blocks
    • For RS(255, 223), 64 KiB
  • Can we do better?
Widely distributing an ECC’s blocks across the partition

Given $T$ blocks, can recover from corruption of $\left\lfloor \frac{T}{n} \right\rfloor \times \frac{255-k}{2}$ consecutive blocks

$B = \text{block size}$

$D = \text{distance separating bytes in ECC block}$
Can we protect against a rollback attack?

A common exploit feature: **Firmware rollback**

- **Idea:** Find a non-persistent vulnerability in current kernel, rollback to more vulnerable older version to install persistent vulnerability
  - Why does this work? (Consider the signature)

**Rollback protection** prevents this

- In tamper-evident storage, record the most recent version
- Refuse to boot older versions
Modern devices have 2 partitions for updating quickly
Conceptual boot flow with rollback protection

START

Is device locked?

Valid OS found? (Accept verification errors and any key)

Valid OS found? (Accept only embedded verification key)

Cannot boot
Enter repair mode

Update Stored Rollback Indexes

Warn about custom OS.
Dismiss after 10 seconds

Warn about OS not being verified.
Dismiss after 10 seconds

Boot OS
What possible exploits remain?
Wrap-up

Secure boot can protect against attacks that modify system data

Usually, multiple layers needed

Error-correcting codes can avoid amplifying corruptions
  • Interleaving targets common errors in practice

Rollback attacks can be dangerous, but prevented with tamperproof partitions