UEFI Secure Boot on U-Boot

Grant Likely

23 Aug 2019

UEFI is a set of standards for firmware

- UEFI defines how firmware should behave and how services are provided to applications
- UEFI is not an implementation
- *Tianocore/EDK2* is the reference UEFI implementation
- U-Boot also implements UEFI

UEFI simplifies development and deployment of embedded Linux systems

- “Every platform is different” -- a common complaint from embedded developers
- Behaviour is defined for common boot scenarios
  - i.e. How to find and boot an OS from a block storage device is part of the spec
  - Distro boot was the first step down this path
  - Adopting UEFI means the embedded and server boot flows are identical
- APIs are defined
  - Pre-boot code and OS loaders are portable
- OS Distros don’t needs separate images for each and every platform
- End user doesn’t need to know device specific details (i.e. Flash partition allocations)

UEFI defines an executable format and API for pre-boot applications

- PE-COFF binaries
- Standard API providing services to UEFI applications
  - Environment Variables
  - Storage and Filesystem services
  - Network services with IP stack and iPXE
- Applications are portable and don’t need any knowledge of hardware
- Applications can run as bare metal applications, or load an OS that takes over the system

UEFI also defines an API for runtime services

- UEFI provides OS with a small library of hooks for access to services at runtime
- Runtime Services remain available after OS calls ExitBootServices()
- Primarily used to access and modify UEFI variables to control boot flow

U-Boot UEFI is in active development and maturing fast

- Both OpenSUSE and Fedora use U-Boot UEFI to boot Arm SBC
  - 32-bit and 64-bit
- EBBR provides specific UEFI requirements that are tailored for embedded
- Can be enabled on most U-Boot targets
- Runtime services implemented, but mostly empty stubs

Demo

- OpenSUSE Tumbleweed generic image
- Mainline QEMU
- U-Boot compiled from mainline yesterday

```
$ qemu-system-aarch64
    -machine virt
    -cpu cortex-a57
    -nographic
    -m 256
    -bios u-boot.bin
    -drive if=none,format=raw,file=openSUSE-Tumbleweed-ARM-JeOS-
efi.aarch64-2019.07.31-Snapshot20190814.raw,id=hd0
    -device virtio-blk-device,drive=hd0
```

UEFI Secure Boot is an extension that verifies application code is signed before execution

- Applications must be signed
- Hierarchal verification model for delegating trust
- Only addresses the firmware→OS boundary
  - Presumes prior boot steps are anchored to a HW root of trust
  - Presumes UEFI application will verify anything it loads

UEFI Secure Boot adds concept of secure variables

- Secure variables are protected from modification, deletion and rollback
- Updates to secure variables must be signed with the appropriate key
- Most important secure variables
  - Platform Key (PK): Used to verify PE
  - Key Exchange Key (KEK): Database of keys used to verify DB/DBX changes
  - db: Database of signatures and keys used to verify applications
  - Dbx: Blacklist database of keys and signatures

UEFI Variable semantics don’t match U-Boot’s

- Currently UEFI variables are stored as U-Boot variables
- U-Boot bulk stores all variables on ‘saveenv’ cmd
- UEFI defines muliple semantics
  - Volatile vs. non-volatile: volatile variables are never stored
  - Secure variables: Updates much be validated against PK/KEK
  - Runtime vs. Boottime: Only runtime variables are exposed after ExitBootServices()
- No clear solution yet
  - Active discussion on mailing list
  - Current patches too invasive
  - Need to extend U-Boot variable system to provide correct behavior

Vast majority of Secure Boot can be implemented in U-Boot proper

- Secure variable updates can be tested against PK/KEK before applying
- No secrets stored on device
- Secure variables can be stored in regular storage
- U-Boot can verify PK/KEK/db/dbx in normal world at bootup

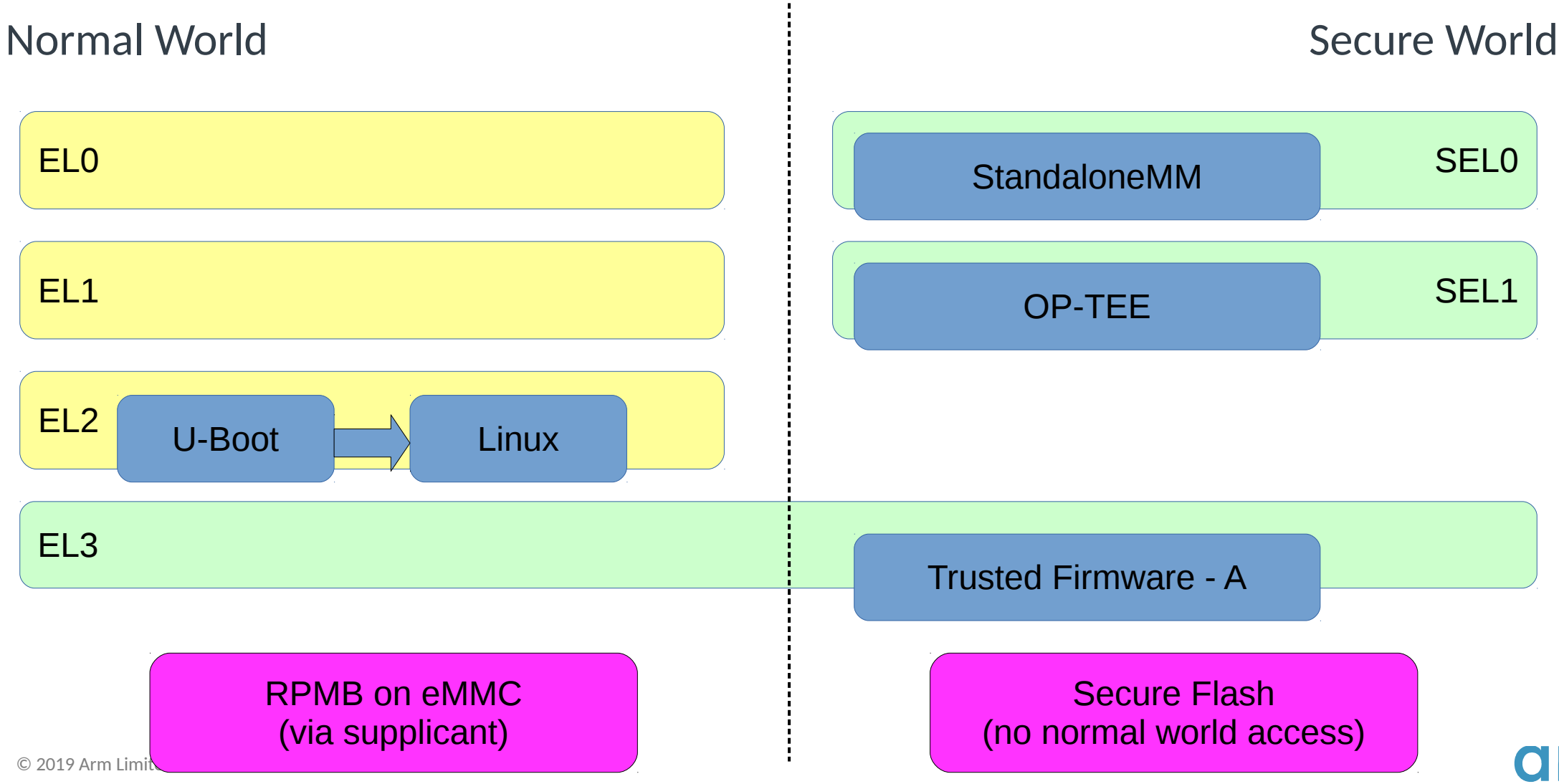
Unless you care about rollback protection

- What if attacker can clear storage device?
- What if attacker can install older, vulnerable, version of variables?
- What if attacker can interfere with storage operations (drop, reorder, or insert transactions)
- Once out of U-Boot, OS can do whatever it wants
  - A compromised OS can be used to attack firmware data

A Trusted Execution Environment can reduce the attack surface against firmware data

- Delegate secure variable storage to a TEE application
- Requires backend changes to U-Boot variable storage
- Trusted Application provides an API for get/set variable at both firmware and OS time
- Internals of variable storage service inaccessible to OS

Proposed AArch64 secure variable architecture using Trusted Firmware and OP-TEE



Work is still required for UEFI Secure Boot to land in mainline

- Regular UEFI in good shape – Use it!
- Takahiro Akashi has prototype secure boot patches
  - Haven't been published publicly recently
  - Still some debate going on over architecture
  - Should image verification happen in the trusted application?
- Variable architecture and RSA implementation has taken precedence
  - Should UEFI variables be integrated in to U-Boot variable service, or be completely separate?
- Linaro LEDGE working on OP-TEE w/ StandaloneMM back end
  - U-Boot driver to communicate with StMM
  - StMM running under OP-TEE

Questions?