YVR18-421: Enabling a Secure Data Center with Arm64 Architecture

Jagadeesh Ujja <jagadeesh.ujja@arm.com>
Thomas Abraham <thomas.abraham@arm.com>
Agenda

● Secure Boot on AArch64 Server Platform
● Arm Trusted Board Boot
● UEFI Secure Boot
● UEFI PI Management Mode (MM) and Secure Storage
● Traditional MM Variable Service driver
● EDK2 StandaloneMmPkg and its Communication Protocol
● Modified MM Variable Service Driver for AArch64 Platforms
● MM Variable Service driver flow
● Secure Variable Access on Arm64 Based Platforms
● UEFI Linux Secure Boot Kernel Signing and Verification Demo on Arm SGI-575
● References
Secure Boot on AArch64 Server Platform

- Secure Boot is a mechanism to build (and maintain) a complete Chain of Trust on all the software layers executed in a system, preventing malicious code to be stored and loaded in place of the authenticated one.

- AArch64 based platforms which use all the TF-A boot stages would have two stages of secure boot:
  - Arm Trusted Board Boot (TBB) Stage authenticates images loaded for BL2 and BL3x.
  - UEFI Secure Boot authenticates the DXE drivers, EFI Applications, OS loader and kernel images.
  - Combined together they enable a full Secure Boot establishing a complete Chain of Trust (despite different PKI) from the very first firmware executed up to the OS.
Arm Trusted Board Boot

- Based on Arm TBSA/TBRR documents

- TBBR-Client specification (DEN0006C) reference for Arm Trusted Firmware implementation

- Arm TBB: a reference example on how to build a CoT from the very first ROM firmware executed (BL1) up to the first normal world firmware (BL33)

- JUNO and FVP Platforms are TBB example running and are available on GitHub!
UEFI Secure Boot

A platform ownership model for establishing a trust relationship among

- Platform Owner (ODM/OEM/EndUser) –PO
- Platform Firmware (EDK2 / U-Boot / 3rdparty BIOS) –PF
- OS / 3rd party software vendors –OSV/ISV ->SV

- Uses standard PKI, X.509 certificates and PE images digital signature, based on PE digest/hash calculation described in Microsoft Authenticode PE Signature Format
- Signature database (white/black list) update mechanism from trusted sources

A generic framework, based on the above model, to allow:

- The firmware to authenticate UEFI executable images before allowing their execution, preventing pre-boot malwares to be run
- The Platform Owner and/or SV to securely update the signature databases into PF with new/known allowed/forbidden image signatures
UEFI Secure Boot ... (continued)

- Two asymmetric key pairs:

  1. Platform Key (PK): Trust relationship between PO & PF
     - PKpriv owned by the PO
     - PKpub enrolled into PF
  2. Key Exchange Key (KEK): Trust relationship between SV & PF
     - Different KEKpriv for each SV
     - Each SV enrolls KEKpub into PF

- Platform firmware on NV variables (on tamper proof storage) to hold
  - PKpub/ KEKpub list
  - Signatures DBs: signatures white/black lists (db/dbx)
UEFI Secure Boot … (continued)

- These keys and databases are accessed as secure EFI variables. Stored on non-volatile storage to maintain the chain of trust.

- These UEFI variables must be kept writeable, at some point the system is going to need to write to them.

- Physical attacks that replace SPI NOR or use tools to modify the UEFI variables can roll-back or add errant entries.

- Similarly, software attacking where OS non-UEFI code can directly modify the flash region containing the authenticated variables poses another concern.

- We need an isolated execution (IsoW) to write to flash and store authenticated variables and verify (IsoV) signature of the authenticated variable when updating these variables.

- UEFI implementations rely on MM mode to protect them from malicious code.
UEFI PI Management Mode (MM) and Secure Storage

- Management Mode (MM) is a secure execution environment
  - For x86 systems, entered when the CPU detects a SMI, implemented with System Management Mode (SMM)
  - For ARM systems, implemented with SPM and S-EL0.

- Two Types of MM supported by UEFI PI Specification
  - Traditional Mode
    - MM execution environment is setup during DXE phase
  - Standalone Mode
    - MM execution environment can be setup during or prior to SEC phase
    - An MM standalone mode driver can only run in the MM environment. It is not allowed to touch any other resource, such as a UEFI protocol in the DXE phase, or a PPI in PEI phase.

- Standalone MM is ideal for securing EFI variables required for secure boot
  - Allow access to the secure non-volatile storage only via code executing in MM mode
Traditional MM Variable Service Driver

- Existing implementation of secure variable service in upstream implementations of UEFI (example, edk2) is based on traditional management mode
- Traditional Management Mode is setup as part of the UEFI boot process, specifically, during the DXE phase.
  - SMM Initial Program Loader (IPL) loads the SMM Foundation into SMM memory (SMRAM) and the SMM services will start at that time and stay resident until a system shutdown
  - Cannot be run in earlier phases like PEI phases
Protocols that help create the SMM infrastructure for SMM drivers in the DXE phase include:

- **EFI_SMM_ACCESS2_PROTOCOL** – is used to control the visibility of the SMM memory (SMRAM) on the platform.
- **EFI_SMM_CONTROL2_PROTOCOL** - is used to initiate synchronous System Management Interrupt (SMI) activations.
- **EFI_SMM_CONFIGURATION_PROTOCOL** - indicates which areas within SMRAM are reserved for use by the CPU for any purpose, such as SMM entry point.
- SMM IPL is to launch SMM foundation, and **EFI_SMM_BASE2_PROTOCOL** is used to locate the SMST during SMM driver initialization.
- These modules work together to create the SMM infrastructure for SMM drivers in the DXE phase.
- The SMM foundation will load all of the SMM drivers, and the SMM drivers will register SMI handlers to service synchronous or asynchronous SMI activations.
EDK2 StandaloneMmPkg and its Communication Protocol

- StandaloneMmPkg
  - Implements the PI specification v1.5/Volume 4 to provide the Management mode core interface
  - Includes support for both x86 and AArch64 architectures
  - For AArch64 based platforms, the StandaloneMMPkg is setup during the BL32 boot stage
  - Can be used to enforce a trusted execution environment to keep the secure EFI variables tamper proof

- EFI_MM_COMMUNICATION_PROTOCOL
  - PI v1.5 Specification Volume 4 defines Management Mode Core Interface and defines EFI_MM_COMMUNICATION_PROTOCOL
  - Provides a means for communicating between the non-secure DXE drivers and the Standalone MM drivers.
  - Updates to Secure EFI variables protected by the implementation of MM will use this protocol (with appropriate authentication by MM)
Modified MM Variable Service Driver for AArch64 Platforms

- Standalone MM execution environment setup in S-EL0 execution mode (BL32 boot stage)

- Access to Secure Variables is facilitated by MM_COMMUNICATE protocol
VariableSmmRuntime DXE driver and its counterpart VariableSmm of type DXE_SMM_DRIVER, which executes from the MM execution environment, are based on the design principles of traditional MM execution mode.
  ○ For extending it to AArch64 based platforms, these drivers had to be reworked to be usable with Standalone MM execution mode as well.

Reworked the VariableSmmRuntime DXE driver to eliminate dependencies on
  ○ gEfiSmmVariableProtocolGuid
  ○ gSmmVariableWriteGuid

Reworked the VariableSmm driver to convert it from a DXE_SMM_DRIVER to STANDALONE_MM driver
  ○ DXE_SMM_DRIVER does not execute in the SEC phase
  ○ Removed all implicit access to SMRAM and all the communication between VariableSmmRuntime and VariableSMM were modified to use the MM_COMMUNICATE protocol.
### MM Variable Service Driver Flow

- **Normal World**
  - During runtime, OS consumer calls variable services like GetVariable(), SetVariable()
  - UEFI variable driver will copy data to communication buffer (copy the whole data, not the data pointer)
  - Calls MmCommunication
  - Triggers an Smc Call

- **Secure world**
  - In MMI handler, MM core infrastructure checks the “Header GUID” in MM communication header
  - Dispatch to MM Variable Handler.
  - MMI Variable Handler checks the “Function ID” in Variable Communication Header
  - Dispatch to each variable services, like GetVariable(), SetVariable()
  - MM Variable services can get parameter, like GUID, Name, Data field, from Variable Access Communication Data
  - Finally, SetVariable() API can call the crypto services to do the authentication and update flash
Secure Variable Access on Arm64 Based Platforms

Normal World

- EL0: Applications
- EL1: OS Kernel
- EL2: EDK2 Boot, EDK2 Runtime, VariableSmmRunTimeDxe, MM_COMMUNICATE
- EL3: Arm Trusted Firmware

Secure World

- S-EL 0: Pkpub KEKpub db/dbx
- S-EL 1: UEFI Variables in NOR Flash partition:
  - Keys
  - SignatureDBs

NOR Flash

- NOR Flash Driver
- VariableSmm/FaultTolerant WriteSmm
- BaseCryptLib/AuthVariableLib/PlatformSecureLib
- SPM Shim
- OS Runtime: (Authenticated) Read/Write Keys / SignatureDBs
- Secure Partition Manager

GetVariable(), SetVariable()
UEFI Linux Secure Boot Kernel Signing and Verification Demo on Arm SGI-575

- The Uefi Secure Variable Service patches are under internal review
- Tools used for Key pair creation and signing
  - git://git.kernel.org/pub/scm/linux/kernel/git/jejb/efitools.git
  - git://git.kernel.org/pub/scm/linux/kernel/git/jejb/sbsigntools.git
- Key Pair Creation: PK, KEK, DB and DBX
  - openssl req -new -x509 -newkey rsa:2048 -subj "/CN=PK/" -keyout PK.key -out PK.crt -days 3650 -nodes -sha256
  - openssl req -new -x509 -newkey rsa:2048 -subj "/CN=KEK/" -keyout KEK.key -out KEK.crt -days 3650 -nodes -sha256
  - openssl req -new -x509 -newkey rsa:2048 -subj "/CN=DB_Key/" -keyout DB.key -out DB.crt -days 3650 -nodes -sha256
  - openssl req -new -x509 -newkey rsa:2048 -subj "/CN=DBX_Key/" -keyout DBX.key -out DBX.crt -days 3650 -nodes -sha256
- Convert crt certificate to der format
  - openssl x509 -in PK.crt -outform der -out PK.der
  - openssl x509 -in KEK.crt -outform der -out KEK.der
  - openssl x509 -in DB.crt -outform der -out DB.der
  - openssl x509 -in DBX.crt -outform der -out DBX.der

- Sign the Grub And Linux Images
  - sbsign --key DB.key --cert DB.crt --output bootaa64_signed.efi bootaa64.efi
  - sbsign --key DB2.key --cert DB.crt --output Image_signed Image
UEFI Linux Secure Boot Kernel Signing and Verification Demo on Arm SGI-575

- For the first time boot, enable Secure boot and register the DBX, DB, KEK, PK.
UEFI Linux Secure Boot Kernel Signing and Verification Demo on Arm SGI-575 ...

(continued)
UEFI Linux Secure Boot Kernel Signing and Verification Demo on Arm SGI-575 ... (continued)
References

- HKG18-104 - Securing Arm Servers
  ○ https://connect.linaro.org/resources/hkg18/hkg18-104/
- SFO-201: Secure Boot on Arm systems
  ○ http://connect.linaro.org/resource/sfo17/sfo17-201/
- Arm MM Interface Specification
- Arm Trusted Firmware-A –Secure Partition Manager design document
  ○ https://github.com/ARM-software/arm-trusted-firmware/blob/master/docs/secure-partition-manager-design.rst
- Arm SDEI Specification
- Arm Trusted-Firmware-A –SDEI design document
  ○ https://github.com/ARM-software/arm-trusted-firmware/blob/master/docs/sdei.rst
- Arm Secure-EL2 extension
  ○ https://community.arm.com/processors/b/blog/posts/introducing-2017s-extensions-to-the-arm-architecture
- Arm Fixed Virtual Platforms, including latest System Guidance for Infrastructure
Thank You