Qualcomm Linux Security Guide

80-70018-11 AB

April 10, 2025

Contents

1 Security overview

Qualcomm Linux Security is an advanced security framework designed to protect devices running on Qualcomm Linux. It integrates several critical components and methodologies to ensure robust security across various layers of the system.

This guide covers the following areas.

Explore features

→ Security features

→ Security architecture

APIs and tools

→ Security APIs

→ Security tools

Develop

→ Trusted and client applications (For licensed users)

How to

→ Bring up security features

→ Configure security services

→ Customize security services

→ Debug Qualcomm TEE and secure devices

Sample code and applications

→ Security services examples (For licensed users)

Explore SoftSKU

→ Install or upgrade SoftSKU feature packs

1.1 Qualcomm Linux product security

The Qualcomm® Linux® product security provides a comprehensive framework designed to protect devices from both software and hardware threats. The Qualcomm Linux Security foundations ensure a high level of security and robustness, superior performance, and energy efficiency.

The following figure shows the major components of the security foundation framework.

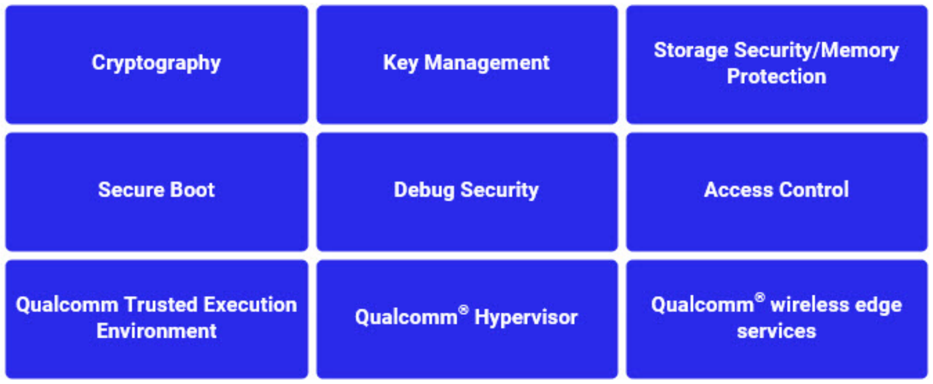

Cryptography
Key Management
Storage Security/Memory Protection
Secure Boot
Debug Security
Access Control
Qualcomm Trusted Execution Environment
Qualcomm® Hypervisor
Qualcomm® wireless edge services

Figure : Qualcomm Linux Security foundations

The following list describes each component within the Qualcomm Linux secure foundations:

- Cryptography: Provides enhanced security and faster encryption/decryption operations through crypto engines and hardware keys.
- Key management: Uses a one-time programmable (OTP) eFuse memory referred as the Qualcomm fuse programmable read-only memory (QFPROM) to store generic information and configurations.
- Storage security: Offers encrypted storage for digital rights management (DRM) keys and certificates. These keys are securely provisioned. The replay protected memory block (RPMB) is a separate partition in the universal flash storage (UFS) device designed for secure data storage.
- Secure boot: Offers image authentication and tamper-resistant root of trust (RoT) in electronic fuses (eFuse).
- Debug security: Allows debugging in a secure environment. It's set by an eFuse and restricts the invasive (INV) and non-invasive (NINV) debug capabilities in commercial products and reverse engineering. For more information, see Learn the architecture - Before debugging on Armv8-A.
- Access control: Prevents unauthorized code execution using the access-controlled memory and peripherals.
- Qualcomm Trusted Execution Environment (TEE): Isolates the secure and non-secure software operations. It's built on the Arm® TrustZone® architecture and has a rigorously

reviewed small code base.

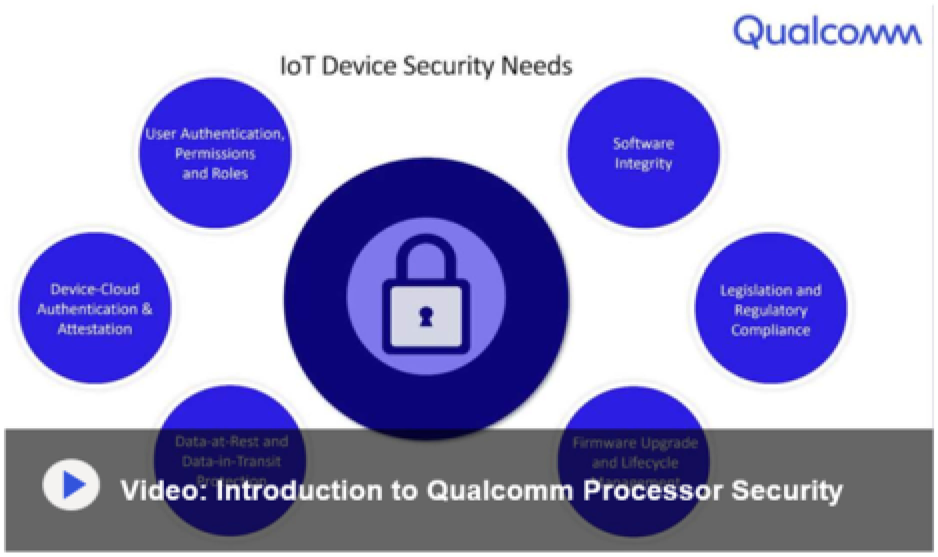
- Qualcomm Hypervisor: Allows multiple operating system environments to run simultaneously and independently.
- Qualcomm wireless edge services (WES): A cloud-based service that ensures trust throughout the device lifecycle using hardware root-of-trust (RoT). Qualcomm WES provides services such as hardware-based attestation, zero-touch device provisioning, and chipset feature management.

Note: See Hardware SoCs that are supported on Qualcomm Linux.

1.2 Watch video on IoT Security

Introduction to Qualcomm Processor Security

Dive deep into the world of IoT security with our comprehensive guide focused on Qualcomm chip products. Learn how to optimize and enhance the safety of your IoT products through advanced security solutions. This video covers everything from understanding potential threats like code modifications and key exposures to implementing foundational security requirements. Whether you are dealing with biometric authentication or cloud connectivity, discover how Qualcomm Linux Security features can safeguard your devices against evolving cyber threats and meet stringent regulatory standards. Join us to fortify your IoT devices now!


Qualcomm
IoT Device Security Needs
User Authentication, Permissions and Roles
Software Integrity
Device-Cloud Authentication & Attestation
Legislation and Regulatory Compliance
Data-at-Rest and Data-in-Transit Protection
Firmware Upgrade and Lifecycle Management
Video: Introduction to Qualcomm Processor Security

1.3 Next steps

- To explore the security features, see Security features.
- To begin the initial setup and assessment and to activate the security features required for developing your software with Qualcomm Linux, see Bring up security features.

2 Security features

Qualcomm Linux incorporates several security features to protect the devices and applications. These security features are essential for protection against vulnerabilities, data integrity and confidentiality, compliance with standards, and system stability.

Qualcomm TEE enhances the security features and their extensions. It offers interfaces that allow the extension of the security feature set through the trusted applications. Certain features are integrated into the hardware-supported TrustZone architecture, providing a system security configuration. These features can be further customized to meet specific requirements.

2.1 Explore security features

Click on each feature to find out more.

Cryptography

Qualcomm Linux Security offering includes support for both hardware and software-based cryptography.

The key capabilities include:

- A register and bus access manager with direct memory-based access.
- Interfaces to the cryptographic hardware.
- The Linux kernel crypto driver (qcrypto) provides access to the hardware cryptography independent of trusted applications.
- The Qualcomm TEE provides the hardware and software crypto application programming interfaces (APIs) to the trusted applications.

Qualcomm TEE supports the following cryptographic algorithms:

Algorithm	**Hardware**	**Software**
Hash	SHA-1/SHA-256	• SHA-1/SHA-224/SHA-256/SHA-384/SHA-512 • SM3
Symmetric cipher	• AES-128/AES-256 CBC, ECB, CTR, CCM, GCM, • Triple-TDES CBC/ECB	• AES-128/AES-192/AES-256 CBC, ECB, CTR, CCM, XTS, CFB, OFB, CTS • Triple-TDES CBC/ECB • PBKDF2 • SM4
MAC	AES-CMAC	Hash-based message authentication (HMAC)
RNG	HRNG	–
HMAC	HMAC-SHA-1/SHA-256	HMAC-SHA-1/SHA-224/SHA-256/SHA-384/SHA-512
Asymmetric cipher	–	• RSA with 1024/2048/3072 modulus • ECDSA with P224, P256, P384, P521 • ECDH • SM2

Inline crypto engine

The inline crypto engine (ICE) is designed for a high throughput cryptographic encryption of the storage data.

ICE supports:

- AES 128/AES 256 ECB/XTS
- Multiple crypto streams to meet high throughput
- Multiple AES cores per crypto stream
- Provision of 32 software configurable keys
- Capability to enable symmetric and asymmetric operations

Next steps

- To learn about the next security feature, see Key management.
- To learn about TrustZone and security framework, see Security architecture.
- To learn about APIs that can be used to interact with Linux and hardware, see Security APIs.

Key management

The Qualcomm Linux Security solution supports the public-key cryptography standards by implementing the PKCS#11 APIs. This feature allows applications to use keys and certificates in a platform-independent manner.

PKCS#11 is implemented as a global platform for running trusted applications within Qualcomm TEE. There is also a corresponding rich execution environment (REE) implementation for these applications.

For more information, see the following documents:

- PKCS #11 Cryptographic Token Interface Base Specification
- PKCS #11 Cryptographic Token Interface Usage Guide

Limitations

The following functionalities aren't supported:

- Random number generator functionality
- P-192 in CKM_ECDSA
- RSA PKCS key generation and signing in CKM_RSA_PKCS mode
- EDDSA key generation and signing

Next steps

- To learn about the next security feature, see Secure boot.
- To learn about TrustZone and security framework, see Security architecture.
- To learn about APIs that can be used to interact with Linux and hardware, see Security APIs.

Secure boot

Secure boot is the boot up sequence that establishes a trusted platform for the entire software stack.

See the workflow to understand the secure boot and UEFI secure boot process.

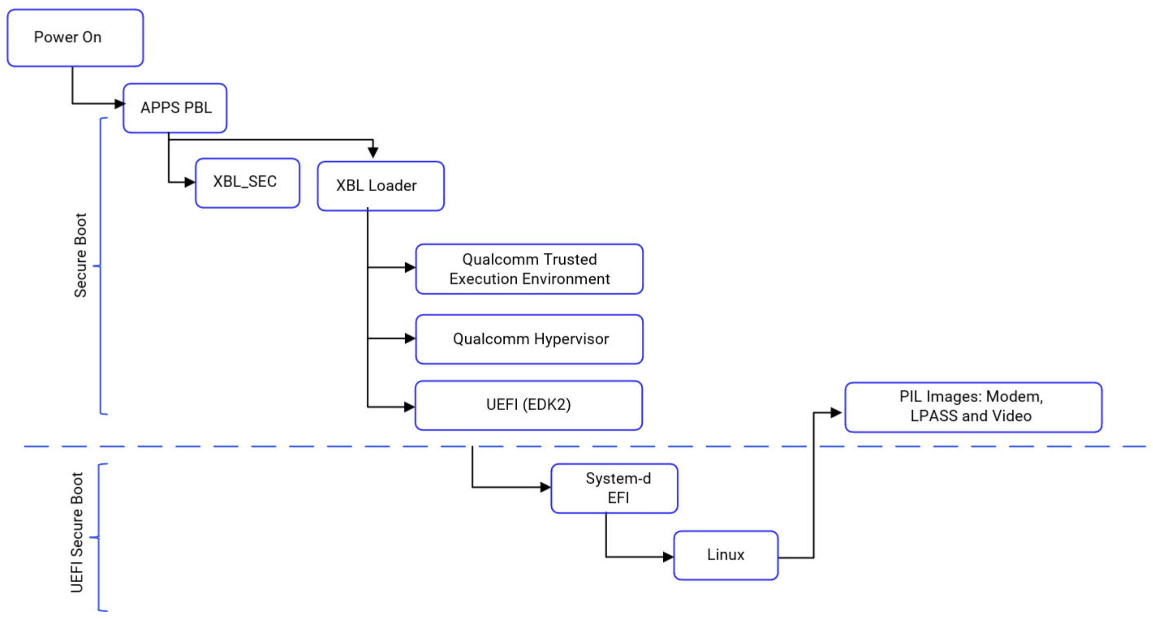


Figure : Secure boot vs UEFI secure boot

This process ensures that only authorized software is executed. It uses cryptographic authentication to initiate an immutable sequence that validates the origin of the code. This process:

- Confirms the authenticity of all software images (non-Linux images) signed by Qualcomm and the users. The device carries out this process.
- Prevents any unauthorized or maliciously modified software from running on the device.

The secure boot feature authenticates non-Linux images while UEFI secure boot authenticates the Linux images.

UEFI secure boot

UEFI secure boot is a feature of the unified extensible firmware interface (UEFI) specification that defines an interface between an operating system and the platform firmware.

For more information, see UEFI specification.

The features of UEFI secure boot include:

- Ensuring that the code executed by the device UEFI firmware is secure and trusted before the operating system begins boot up.
- Defining how the UEFI authenticates images such as the Linux image, the operating system loader (uki.efi), systemd boot (bootaa64.efi), and the device tree blob (DTB) image files.
- Ensuring that the images are loaded only when signed by valid and authorized users. This process also ensures that the Qualcomm Linux security and integrity over systems running on UEFI-based firmware.

The UEFI secure boot allows the Qualcomm Linux users to:

- Verify the integrity and security of a UEFI loaded image, ensuring it's loaded in an approved manner.
- Manage the Qualcomm Linux security policy as defined by the UEFI secure boot authenticated variables, which includes:
  - Platform key (PK)
  - Key exchange keys (KEK)
  - Allowed database (dB)
  - Forbidden database (DBX) (Not supported in this release.)

Next steps

- To learn about next security feature, see Storage security.
- To learn about how to enable secure boot, see Enable secure boot
- To learn about how to enable UEFI secure boot, see Enable UEFI secure boot.

Storage security

The secure file system (SFS) is used to store sensitive data, such as keys and biometric data.

SFS

SFS provides confidentiality, integrity, and anti-rollback support to the trusted applications and securely stores sensitive data. Any file created or stored under SFS is covered by anti-rollback protection. The SFS feature:

- Uses an encryption key for each trusted application to ensure the confidentiality of the files.
- Uses an HMAC key for each trusted application to verify the integrity of the files.

Both the encryption and HMAC keys are derived using a device unique key, which depends on the secure boot state of the device. The SFS anti-rollback protection is enabled by default.

When the devices are secure boot enabled, the SFS uses unique hardware keys for file data encryption and decryption to ensure they're secure from each other.

RPMB

RPMB is a physical partition on the UFS/eMMC flash. This partition is used to store sensitive information and is only accessible from Qualcomm TEE.

To read from and write to the RPMB partition, RPMB key provision is required. This is a one-time process that can't be overwritten or erased when completed.

Every access to the RPMB is authenticated, allowing the host to store data in an authenticated and replay-protected manner.

Next steps

- To learn about the next security feature, see Storage encryption.
- To learn about TrustZone and security framework, see Security architecture.
- To learn about APIs that can be used to interact with Linux and hardware, see Security APIs.
- To learn how to enable or disable the RPMB-based SFS anti-rollback protection, see RPMB-based SFS anti-rollback protection

Storage encryption

Storage encryption enhances security by supporting the transparent encryption of files and directories.

The Qualcomm Linux supports storage encryption with the Inline crypto engine and a hardware-wrapped key. It provides better efficiency and enhanced key protection.

The storage encryption feature allows:

- Provision of standard keys ranging from 32 bytes to 64 bytes for content encryption
- Encryption of filenames and file content with separate keys
- Generation of 32-byte key identifiers

The Qualcomm universal flash storage (UFS) driver is added to support the `fscrypt` API.

Note: The embedded multimedia card (eMMC) support will be provided in a future release.

For more information on the `fscrypt` API, see Filesystem-level encryption (fscrypt) Linux kernel documentation.

For related kernel documentation, see the following files:

- Inline encryption
- I/O control (IOCTL) support for storage encryption

You can invoke the storage encryption functionality using the open-source fscryptctl tool.

Next steps

- To learn about the next security feature, see SMC invoke.
- To learn about TrustZone and security framework, see Security architecture.
- To learn about APIs that can be used to interact with Linux and hardware, see Security APIs.

SMC invoke

The secure monitor call (SMC) invoke is used to expose the services and interfaces implemented in Qualcomm TEE to Linux clients. It provides identification information about the requesting processes to Qualcomm TEE.

The SMC invoke is an object-oriented, capability-based framework that enables communication between Linux clients, TrustZone, and the TrustZone applications, which use mini-kernel (MINK) invoke objects.

During the SMC invoke process, the following objects are involved:

Table : Objects - SMC invoke

Objects	**Owner execution environment**	**Description**
Remote	TrustZone	Linux obtains references to remote objects through the invoked operations. For example, an AppLoader, which is a remote object.
Callback	Linux	Linux sends references of the required callback objects through the invoked operations to the TrustZone, which may invoke these objects as needed.
Memory	Linux	These memory objects are useful for efficiently sharing large buffers between TrustZone and Linux.

The figure shows the SMC invoke architecture:

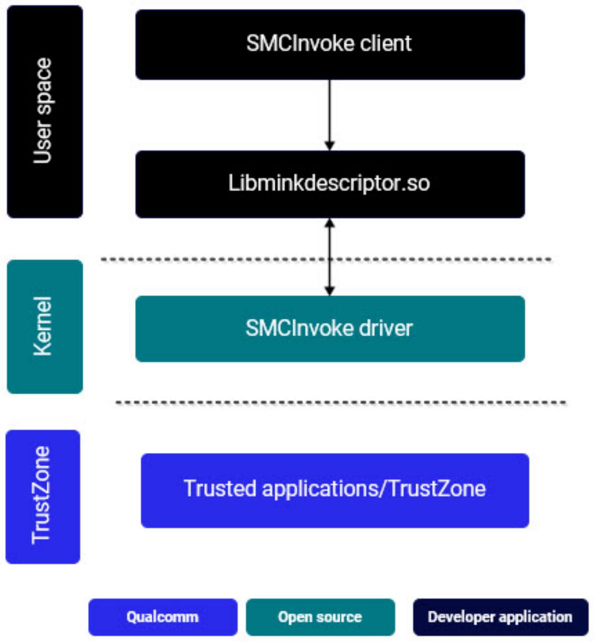


Figure : SMC invoke architecture

The SMC invoke has the following components:

- Linux kernel:

  The SMC invoke driver in the kernel is responsible for transporting user space requests between the TrustZone and the Linux user space clients. It exposes the following input/output controls (IOCTL).

  – `SMCINVOKE_IOCTL_INVOKE_REQ`

  – `SMCINVOKE_IOCTL_ACCEPT_REQ`

  – `SMCINVOKE_IOCTL_SERVER_REQ`

  – `SMCINVOKE_IOCTL_ACK_LOCAL_OBJ`

- Linux user space:

  The SMC invoke in the user space includes `libminkdescriptor.so` and the SSGTZD daemon.

Table : SMC invoke modules

Module	Description
`SMCInvoke client`	This client communicates with the trusted applications in Qualcomm TEE through the SMC invoke driver.
`TA client`	This client communicates with the TrustZone applications through the SMC invoke driver.
`Libminkdescriptor`	• The module creates a bridge between the user space client and kernel. • A client using SMC invoke must use `libminkdescriptor` to communicate with the kernel. • The `libminkdescriptor.so` is available at `/usr/lib/libminkdescriptor.so`
`SMCInvoke driver`	• The driver in the Linux kernel communicates with application clients and the trusted applications in the TrustZone. • The SMC invoke driver source code is available at `/kernel/drivers/soc/qcom/smcinvoke/smcinvoke.c`.

Next steps

- To learn about the next security feature, see Access control.
- To learn about TrustZone and security framework, see Security architecture.
- To learn about APIs that can be used to interact with Linux and hardware, see Security APIs.

Access control

Access control ensures only authorized entities can access specific resources under defined conditions, using policies, technologies, and trust models.

The access control trust model ensures ongoing security by managing access control configurations among various assets, interfaces, SoC (system on chip) components, and images. This model helps maintain the integrity and confidentiality of sensitive information.

- The Qualcomm access control uses an external protection unit (xPU) to control access from the secondary side for registers, fixed address, and dynamic memory regions.
- The system memory management unit (SMMU) controls access from the primary side for content protection and subsystem memory sharing use cases.

Access control domains

The two levels of control are:

1. The TrustZone manages the TrustZone-domains and controls access from the secondary side using xPUs.
    - An xPU is a combination of many security blocks known as protection units. It allows conditional access based on a set of programmable access control registers.
    - If the system denies access, the xPU generates an error output signal and, if necessary, an interrupt request signal.
2. The hypervisor (EL2) manages the non-secure domains, including the content protection zone (CPZ) that's designed to prevent access to premium videos.
    - It configures access control from the primary side through the second-stage page tables through the SMMU.
    - It manages stage-2 mappings to protect assets across all primary domains (Linux, display, GPU, and video). The Linux kernel manages stage-1 mappings to protect the assets in applications that run at the user privilege level.

The figure shows the access control domains.

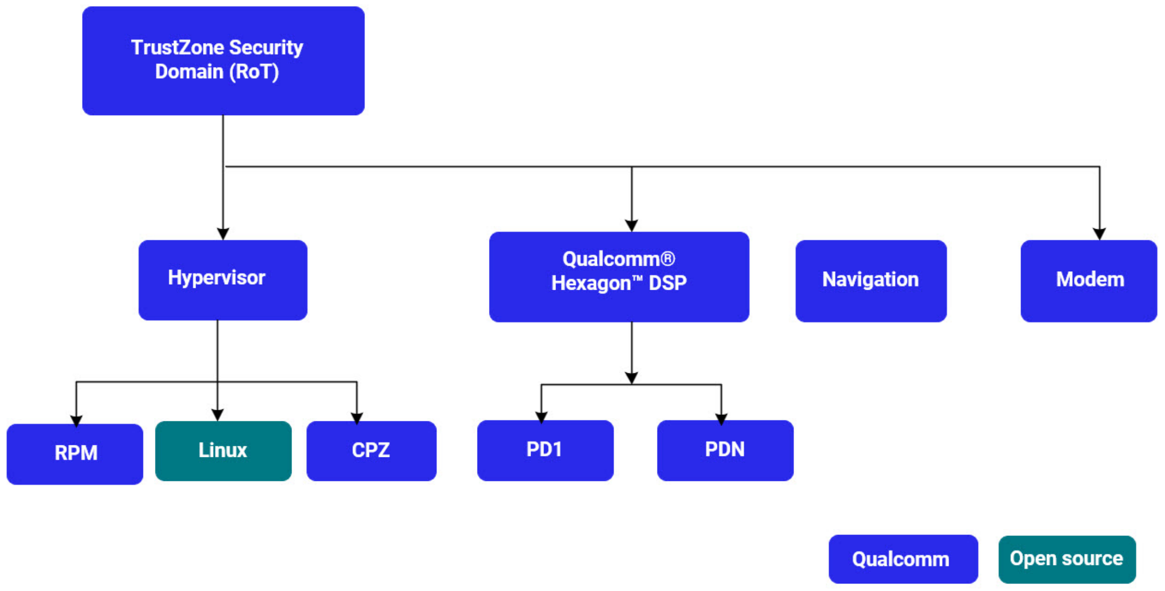


Figure : Access control domains

Next steps

- To learn about the next security feature, see Secure peripheral image loading.
- To learn about TrustZone and security framework, see Security architecture.
- To learn about APIs that can be used to interact with Linux and hardware, see Security APIs.

SPI loading

The secure peripheral image loading (PIL) of a TrustZone authenticates different images and configures the xPUs for all subsystems such as audio, camera, and video.

For more information about xPU, see Access control.

Next steps

- To learn about the next security feature, see SELinux.
- To learn about TrustZone and security framework, see Security architecture.
- To learn about APIs that can be used to interact with Linux and hardware, see Security APIs.

SELinux

SELinux is a security enhancement for Linux, providing greater control over system access. It implements mandatory access control (MAC) in the Linux kernel using the Linux security modules (LSM) framework.

For more information, see What is SELinux?

The LSM framework includes the following controls:

- Discretionary access control (DAC)
  - This is the standard form of Linux access control. The data owner has full control over the resource (such as data and files) and can update or modify the data.
  - Some applications (with root privilege) can override this control. Some resources, like the socket, are unchecked.
  - Access controls are provided with limited granularity, based on user and group identity. For example, File mode `-rwxr-xr-x-`.
- MAC
  - A system-wide security policy determines the Linux access controls for all processes, objects, and operations.
  - The policy can confine flawed and malicious applications or incautious users. It can even prevent users from escalating privilege such as root or UID 0.

  – MAC provides finer granularity to access controls, allowing access to many more resources other than files with more specific controls. For example, unlink, append only, or move a file.

The figure shows the steps in the decision-making chain for DAC and MAC:

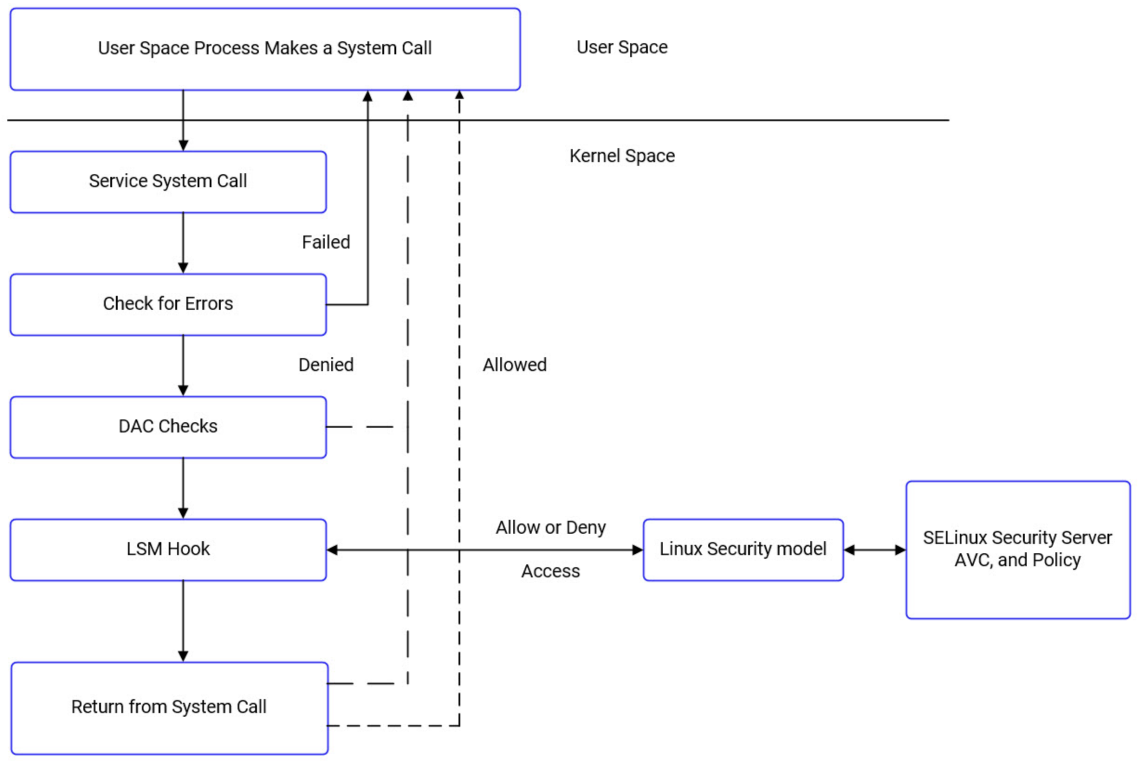


Figure : Linux security modules: DAC and MAC

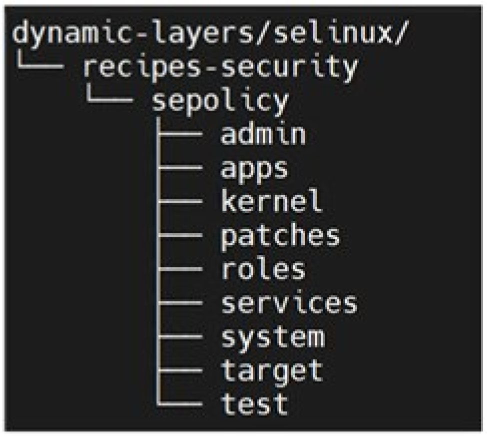
SELinux supports three modes:

- Enforcing mode: In this mode, the SEPolicy is enforced on the system. If the SEPolicy rules aren't met when software is run, access is prevented. The kernel logs an attempted access violation as an access vector cache (AVC) denial message to `dmesg` and `journalctl`.
- Permissive mode: In this mode, the SEPolicy isn't enforced on the system. All denials are logged into `dmesg` and `journalctl` logs, but access of a process or software isn't disallowed.
- Disable mode (Default): In this mode, the SEPolicy isn't enforced or logged.

For information about enabling SELinux, see Enable SELinux. To configure the SELinux modes, see SELinux configuration.

SELinux layers

The SELinux dynamic layer incorporates essential SELinux-specific code modifications that activate and initialize the device when SELinux is enabled with a monolithic design.



```
dynamic-layers/selinux/
└── recipes-security
    └── sepolicy
        ├── admin
        ├── apps
        ├── kernel
        ├── patches
        ├── roles
        ├── services
        ├── system
        ├── target
        └── test
```

Figure : SELinux layers

This layer is built on top of the meta-SELinux layer, which is part of the Yocto upstream. It includes:

- The recipe-kernel, which contains the recipe to enable the required kernel configuration flags for SELinux.
- The recipe-security includes:
  - Policies for booting the device to the shell
  - Policies for Qualcomm services and test applications
  - Policies of upstream services in the form of patches
  - A recipe to handle compilation of Qualcomm policies

SEPolicy

SEPolicy is a directory that contains the core SELinux policy configuration.

The upstream SEPolicy is defined in `sepolicy`, which is downloaded during the build and amended with Qualcomm SEPolicy at: `layers/meta-qcom-hwe/dynamic-layers/selinux/recipes-security/sepolicy/`.

The upstream directory structure is adhered to, with `apps/kernel/system/services` added to the respective directory.

The figure shows the high-level directory structure that's merged and compiled to generate `policy.33`, which is the SEPolicy binary used or loaded during boot.

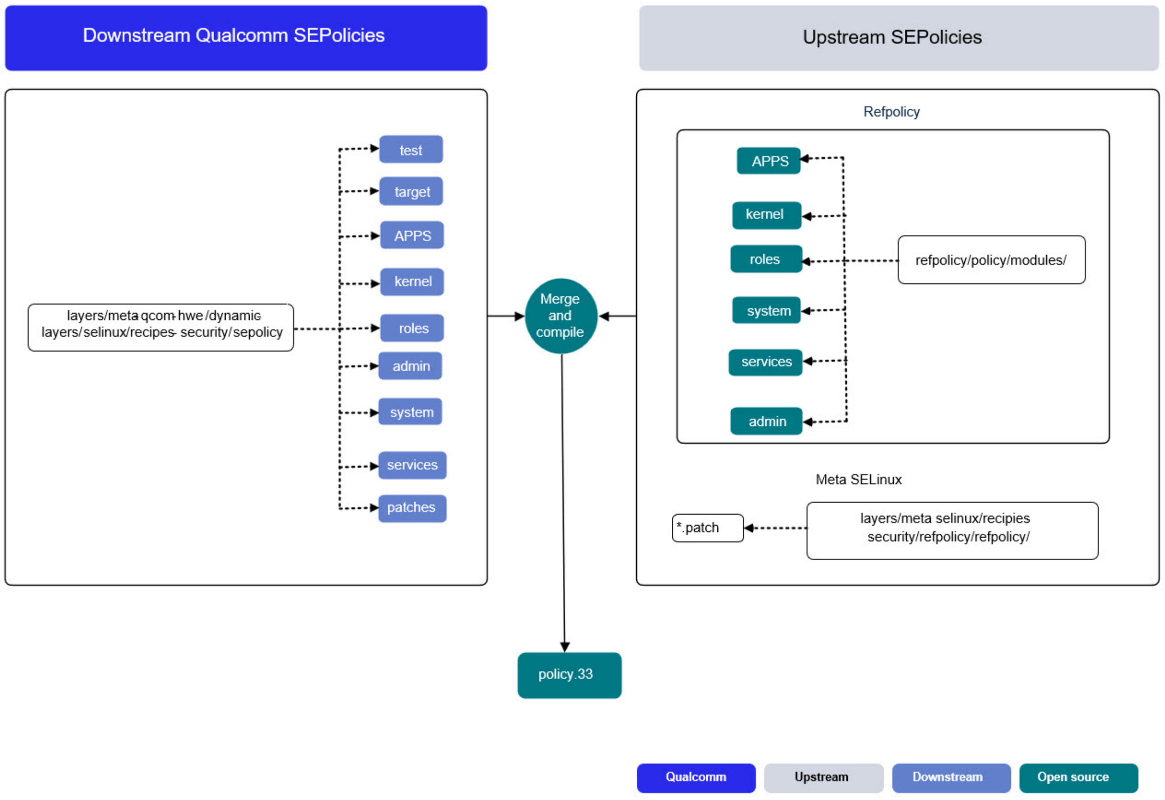


Figure : SEPolicy high-level directory

For more information, see Customize memory and SEPolicy.

Next steps

- To learn about the next security feature, see Qualcomm TEE.
- To learn about TrustZone and security framework, see Security architecture.
- To learn about APIs that can be used to interact with Linux and hardware, see Security APIs.

Qualcomm TEE

Qualcomm TEE is the software that operates within the Arm TrustZone environment on the Qualcomm device.

The TrustZone is a hardware-based security architecture enabled through a Secure mode of the Arm processor. It establishes two execution environments with system-wide hardware-enforced isolation. For more information, see What is TrustZone?.

Qualcomm offers a 64-bit Arm 8.x processor system with hardware virtualization to run TrustZone.

In the TrustZone architecture, there are two security states:

- Secure
- Non-secure

At the EL0, EL1, and EL2 exception levels, the processor can be in either the secure state or the non-secure state while EL3 is always in the secure state.

The operating system runs in non-secure EL1. The transition from Non-secure to Secure mode is facilitated through a Secure Monitor mode.

Qualcomm TEE provides the following features:

- Operation from hardware-protected memory
- Support for power-collapse of security blocks such as the crypto engine, PRNG, inline crypto engine, and external protection units (xPU)
- Support for a secure peripheral image loader (PIL)
- Support for subsystem restart
- Provision of content protection
- Support for running trusted applications
- Support for fuse management

Trusted applications

Trusted applications (TA) offer services within a secure environment for Linux clients that aren't secure. Qualcomm TEE extends the following services to TA:

- Support for trusted applications to operate in the secure world at EL0
- Sand-boxing environment for trusted applications
- Position-independent loading of trusted applications
- Message passing between different trusted applications

TA operates from the memory that's protected by the hardware. However, the applications that require additional memory can use double data rate (DDR) memory for loading and running. By default, an application is set to run from hardware-protected memory.

Next steps

- To learn about the next security feature, see Qualcomm Hypervisor.
- To learn about TrustZone and security framework, see Security architecture.
- To learn about APIs that can be used to interact with Linux and hardware, see Security APIs.

Qualcomm Hypervisor

Qualcomm Hypervisor provides a modern virtualization framework that allows multiple operating systems to run independently and concurrently, delivering high performance.

Qualcomm Type 1 Hypervisor facilitates the hosting of multiple trusted execution environments for secure use cases. The figure shows the Qualcomm Hypervisor software architecture, components, and virtual machines. It also includes an example of one guest virtual machine using the Linux kernel.

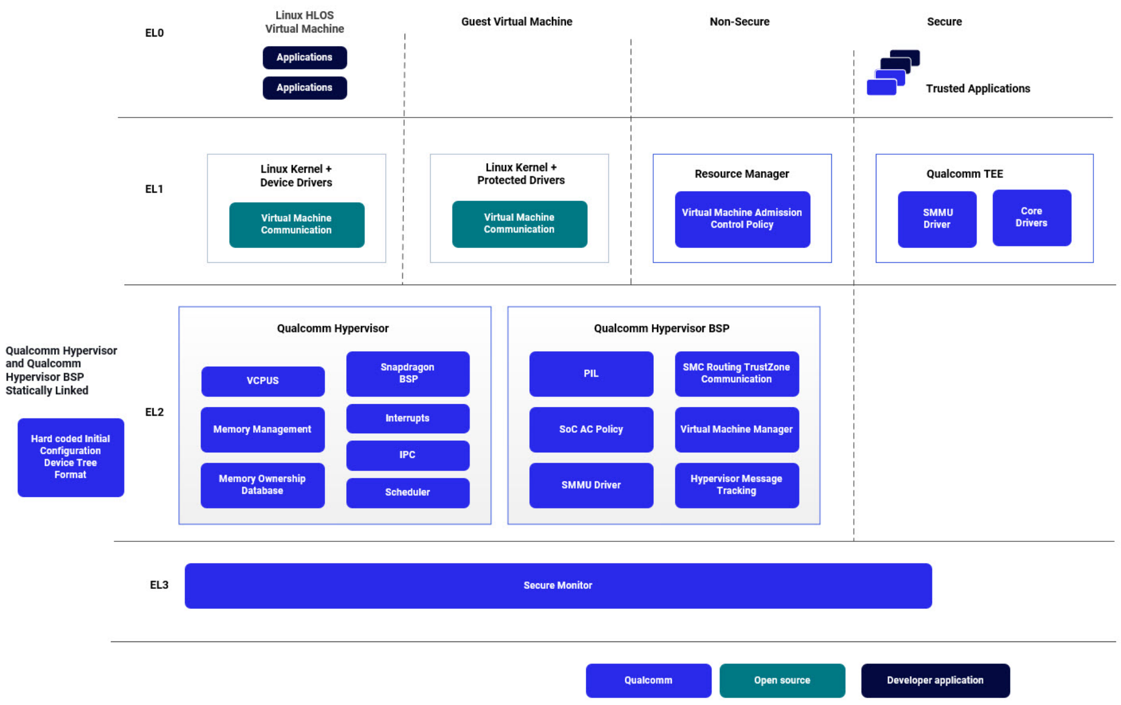

EL0
Linux HLOS Virtual Machine
Applications
Applications
Guest Virtual Machine
Non-Secure
Secure
Trusted Applications
EL1
Linux Kernel + Device Drivers
Virtual Machine Communication
Linux Kernel + Protected Drivers
Virtual Machine Communication
Resource Manager
Virtual Machine Admission Control Policy
Qualcomm TEE
SMMU Driver
Core Drivers
Qualcomm Hypervisor and Qualcomm Hypervisor BSP Statically Linked
Hard coded Initial Configuration Device Tree Format
EL2
Qualcomm Hypervisor
VCPUS
Snapdragon BSP
Memory Management
Interrupts
IPC
Memory Ownership Database
Scheduler
Qualcomm Hypervisor BSP
PIL
SMC Routing TrustZone Communication
SoC AC Policy
Virtual Machine Manager
SMMU Driver
Hypervisor Message Tracking
EL3
Secure Monitor
Qualcomm
Open source
Developer application

Figure : Hypervisor architecture

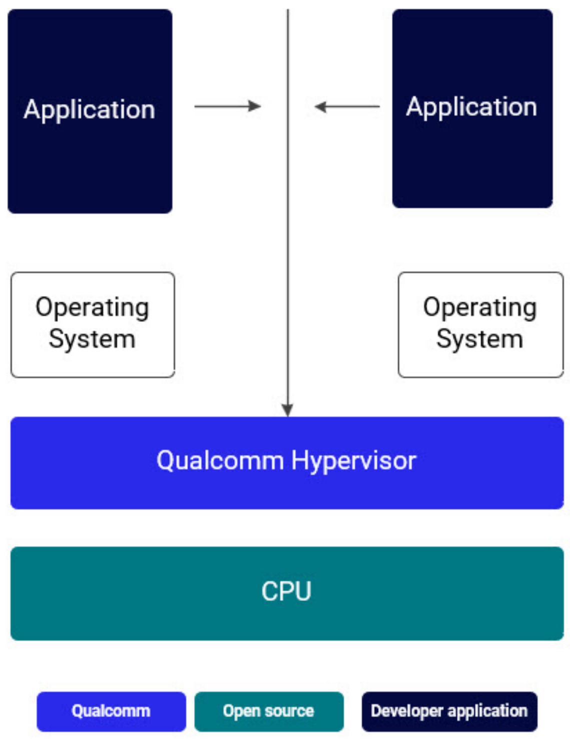
Qualcomm Type 1 Hypervisor, also known as a native Hypervisor, does the following:

- Provides enhanced security compared to Type 2 (hosted) Hypervisors
- Supports multi-core, real-time operations that are independent of Linux
- Operates distinctly, independently, and with higher privileges than Linux
- Has a minimal impact on the performance

The architecture of the Type 1 Hypervisor includes Qualcomm Hypervisor access control, which supports:

- Multiple operating systems
- Smaller attack surface

The following figure shows the Type 1 hypervisor:



Virtualization

Virtual machines and CPUs provide flexibility and abstraction. The Qualcomm Hypervisor offers the following virtualization features:

Feature	Description
Memory management	• Qualcomm Hypervisor manages stage-2 page tables and translates the intermediate physical address (IPA) to the physical address (PA) • It also isolates memory between virtual machines
Interrupt virtualization	It virtualizes and maps interrupts to the correct and independent virtual machines

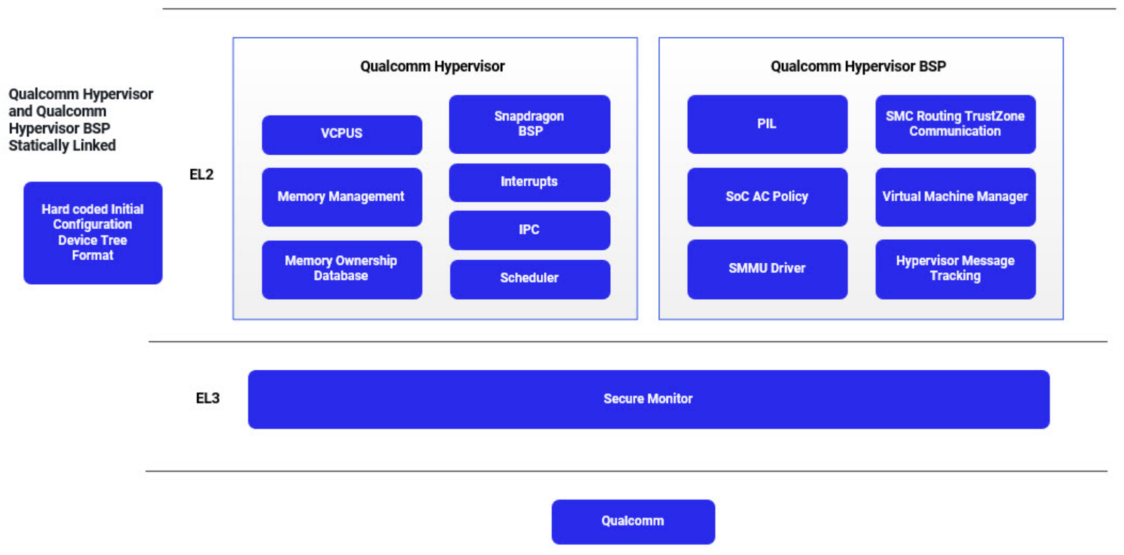
Feature	**Description**
Scheduling	The virtualization CPU (VCPU) scheduler prevents virtual machine starvation during failure
Interprocess communication (IPC)	This includes shared memory, message passing (IPC) APIs, and virtual interrupts
Power management	It supports multiple virtual machines and adopts the lowest common denominator power state

Qualcomm Hypervisor BSP

Qualcomm Hypervisor supports the following features within the board support package (BSP):

- PIL
- Access to SoC AC policy
- System MMU driver
- Pathway support for SMC TrustZone communication

The figure shows all the features of Qualcomm Hypervisor and BSP, which collectively provide a complete hypervisor solution:


Qualcomm Hypervisor and Qualcomm Hypervisor BSP Statically Linked
Hard coded Initial Configuration Device Tree Format
EL2
Qualcomm Hypervisor
VCPUS
Snapdragon BSP
Memory Management
Interrupts
IPC
Memory Ownership Database
Scheduler
Qualcomm Hypervisor BSP
PIL
SMC Routing TrustZone Communication
SoC AC Policy
Virtual Machine Manager
SMMU Driver
Hypervisor Message Tracking
EL3
Secure Monitor
Qualcomm

Resource manager

The exception level 1 (EL1) resource manager performs the following functions:

- Creates and manages virtual machines
- Handles and assigns the virtual machine admission control policy
- Manages the I/O pass-through, which includes:
  - I/O space mapping and isolation
  - Interrupt routing configuration

For more information, see Qualcomm Linux Kernel Guide → Features → Virtualization.

Next steps

- To learn about the next security feature, see Security hardening.
- To learn about TrustZone and security framework, see Security architecture.
- To learn about APIs that can be used to interact with Linux and hardware, see Security APIs.

Security hardening

Security hardening is a process that minimizes the risk of system attacks by making it more challenging for attackers to exploit the system vulnerabilities.

Kernel security hardening aligns with upstream kernel guidelines. Key kernel flags like KASLR, hardened user copy, stack protector, and permissions (RWX) are enabled.

User space hardening

The security_flags.inc file, a part of the Yocto Project is used to enable security compiler and linker flags for a build.

To extend this feature to the Qualcomm modules, add the following command to `qcom-security_flags.inc` (file path: qcom-security_flags.inc):

```
require conf/distro/include/security_flags.inc
```

Adding these flags may result in warnings or errors that can disrupt a build. However, Yocto provides a way to disable certain compiler flags for problematic packages. Modern compilers such as GCC and Clang offer a wide range of compiler flags that can make it more difficult for an attacker to exploit certain types of vulnerabilities.

The following are the example flags with GCC:

- The `Wformat` flag adds compile-time checks to detect issues related to the format of string arguments in common library functions such as `printf`, `scanf`, and `strftime`.

- The `D_FORTIFY_SOURCE` flag adds compile and runtime checks to detect buffer overflows in memory and string functions.
- The `Fstack-protector` flag adds runtime checks to detect buffer overflows and stack smashing.
- The `Fpie` flag enables position-independent code, which allows for loading the binary at randomized locations, thus making certain types of attacks (like return-oriented programming) more difficult.
- The `Wl,-z,relro,-z,now` flag makes it harder to abuse a binary global offset table.

If there are warnings and errors, customizing these flags for some modules can break a build. The binaries in a file system can be verified if the compiler exploit mitigation features are applied using the Checksec tool.

For information about making images more secure, see The Yocto Project Documentation.

Next steps

- To learn about the next security feature, see Qualcomm WES.
- To learn about TrustZone and security framework, see Security architecture.
- To learn about APIs that can be used to interact with Linux and hardware, see Security APIs.

Qualcomm WES

Qualcomm WES is a suite of trusted services, rooted in hardware, which securely connects and manages devices.

It offers the following services:

- Feature licensing enables device features identified by a feature ID and an alias feature name. These features are enabled by installing the corresponding feature licenses or certificates.
- Device attestation provides cryptographically signed and encrypted data items that describe the security state of the device and its software. These data items are useful for risk engines and other similar applications.
- Secure provisioning enables the generation and use of cryptographic keys that are secured using unique device keys. These keys are used to:
  - Securely provision data to the device post-sale and over-the-air (OTA).
  - Sign data on the device.

Next steps

- To learn about TrustZone and security framework, see Security architecture.
- To install or upgrade QCS5430 SoftSKU feature packs, see: Install or upgrade SoftSKU feature packs.
- To develop applications that offer hardware-based attestation, zero-touch device provisioning, and chipset feature management, see: Qualcomm Linux Wireless Edge Services Guide. This feature is available to licensed users with authorized access.

2.2 Watch videos on Qualcomm Processor Security

Qualcomm Processor Security: Foundation

Unlock the full potential of Secure Boot technology on Qualcomm devices in this comprehensive tutorial. From generating cryptographic keys to programming hardware fuses and managing secure boot status, this video covers every step in detail. Ideal for users aiming to enhance device security through authenticated boot processes. Learn how to use Qualcomm tools effectively to ensure your device boots securely every time.

Qualcomm Processor Security: TEE and Chipset Services

Learn how to use Qualcomm Type 1 Hypervisor and Trusted Execution Environment in this comprehensive tutorial. Dive deep into platform virtualization, secure communication, and the extensive security features offered by Qualcomm. Learn how to develop secure applications using Qualcomm tools and APIs, and understand the critical security use cases and compliance standards that Qualcomm supports. Perfect for users aiming to enhance device security and functionality.

Qualcomm®
Processor Security:
TEE and
Chipset Services
Video: Qualcomm Processor Security - TEE and Chipset Services

2.3 Next steps

- To learn more about cryptography, see Cryptography.
- To learn about TrustZone and security framework, see Security architecture.
- To learn about APIs that can be used to interact with Linux and hardware, see Security APIs.

3 Security architecture

Armv8 has enhanced the TrustZone technology to incorporate security into its architecture. This technology offers a security framework that allows a device to handle security threats at both software and hardware levels.

Note: The architectural information is provided for comprehension purposes only. You don’t need to perform any configuration or customization.

The hardware solution from Arm® allows the design and implementation of software, applications, or services that run in a secure environment. This environment is a separate execution unit that ensures hardware isolation from non-secure execution environments. For more information, see Qualcomm TEE.

Qualcomm Linux operates on a 64-bit Arm 8.x architecture. In this setup, Arm cores have two execution modes:

- Non-secure mode: Linux operates in this mode of the Arm core.
- Secure mode: Qualcomm TEE and trusted applications operate in this Secure mode of the Arm core.

The secure mode of the Arm core forms the essence of the TrustZone technology. It provides a hardware-based security environment for a secure OS and separates the secure world from the non-secure world.

The Qualcomm Linux security software architecture is depicted in the following figure. It shows the distribution of components or modules across both the rich (user space and kernel space) and trusted execution environments.

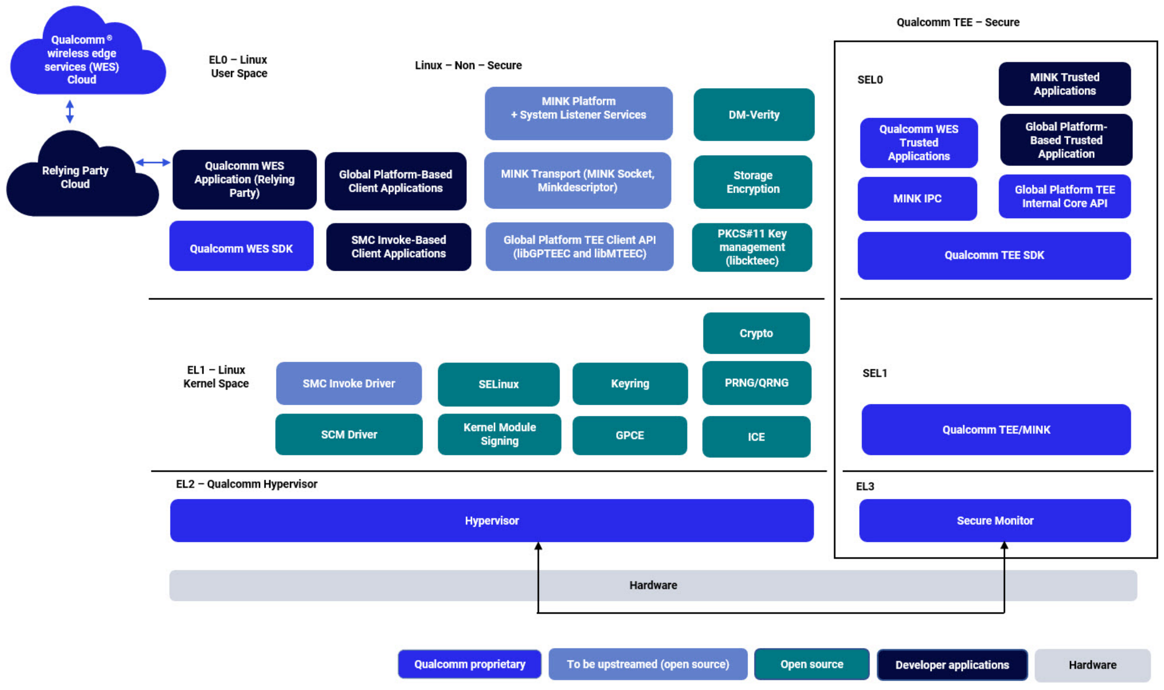

Qualcomm® wireless edge services (WES) Cloud
Relying Party Cloud
EL0 – Linux User Space
Linux – Non – Secure
MINK Platform + System Listener Services
DM-Verity
Qualcomm WES Application (Relying Party)
Global Platform-Based Client Applications
MINK Transport (MINK Socket, Minkdescriptor)
Storage Encryption
Qualcomm WES SDK
SMC Invoke-Based Client Applications
Global Platform TEE Client API (libGPTEEC and libMTEEC)
PKCS#11 Key management (libckteec)
Qualcomm TEE – Secure
SEL0
MINK Trusted Applications
Qualcomm WES Trusted Applications
Global Platform-Based Trusted Application
MINK IPC
Global Platform TEE Internal Core API
Qualcomm TEE SDK
EL1 – Linux Kernel Space
Crypto
SMC Invoke Driver
SELinux
Keyring
PRNG/QRNG
SCM Driver
Kernel Module Signing
GPCE
ICE
SEL1
Qualcomm TEE/MINK
EL2 – Qualcomm Hypervisor
Hypervisor
EL3
Secure Monitor
Hardware
Qualcomm proprietary
To be upstreamed (open source)
Open source
Developer applications
Hardware

Figure : Qualcomm Linux security software components

The Qualcomm Linux security software has the following key components, each operating in different execution environments:

Exception levels	Software components
Non-secure EL0	Linux user space - MINK, key management, client applications,and Qualcomm WES
Non-secure EL1	Linux kernel space - SMC invoke and crypto/PRNG driver
Non-secure EL2	Qualcomm Hypervisor
Secure EL0	Qualcomm TEE SDK and trusted application
Secure EL1	Qualcomm TEE MINK
Secure EL3	Secure monitor

3.1 Linux user space (Non-secure EL0)

This user space includes the primary security modules/interfaces present in the Linux user space, which operates in non-secure EL0 according to the Arm exception levels.

- MINK - Linux (MINK platform/system listener services/transport mechanism)
- Provides the services and transport mechanism to use the Qualcomm TEE capabilities

through trusted applications.

- Forms the system listener services, which are designed to extend the Qualcomm TEE functionalities. The MINK interface is also commonly known as SMC invoke.
- Implements the global platform TEE client APIs for executing global platform-based trusted applications through `libGPTEEC` and `libMTEEClibraries`. For more information, see TEE Client API Specification.
- Implements client applications in a rich execution environment (Linux user space). It interacts with the trusted application that operates in Qualcomm TEE. The various client applications include:
  - SMC invoke-based/MINK client applications
  - Client application that's developed using MINK/SMC invoke APIs
  - Global platform-based client application
  - Client application that's developed using the global platform-based TEE client API specification
- Key management
  - The crypto client library (`libckteec`) interfaces with the PKCS#11 interface of TEE. User space applications running on Linux can use this library for direct access to the PKCS#11 interface.
  - The `libckteec` library defines the interface between an application and a cryptographic device, enabling applications to treat cryptographic devices as tokens and perform cryptographic functions as implemented by these tokens.
  - The PKCS#11 interface provides a range of cryptographic services for encryption, decryption, signature generation, signature verification, and permanent key storage.

The figure shows key management in the kernel and user space.

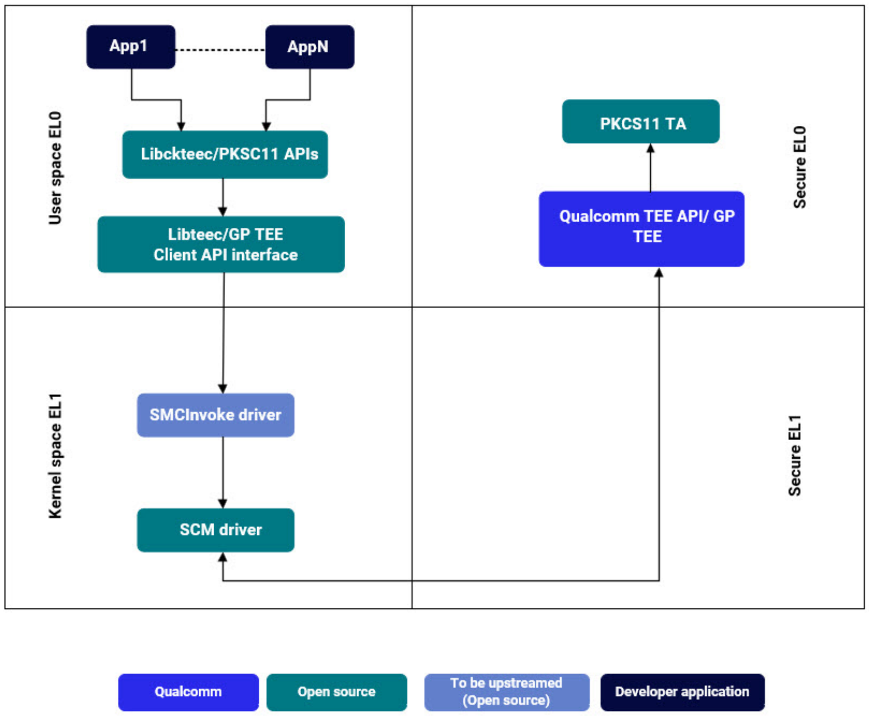


Figure : Key management in kernel and user space

- For more information about key management, see Qualcomm WES and Storage encryption.
- For more information about the various security software components, see Features.

3.2 Linux - Kernel space (Non-secure EL1)

The following are the security-specific kernel drivers:

- SMC invoke driver:
- Provides a Linux abstraction for Qualcomm TEE objects by encapsulating them in file descriptors.
- Marshalls invocations into the Qualcomm TEE objects in buffers that are shared with Qualcomm TEE.
- Tracks and dispatches callback invocations to Linux processes.
- Tracks all the shared memory objects, providing information about them to Qualcomm TEE.
- For the CM and crypto/PRNG drivers, see the respective kernel documentation.

3.3 Qualcomm Hypervisor (Non-secure EL2)

For more information, see Qualcomm Hypervisor.

3.4 Qualcomm TEE SDK (Secure EL0)

Qualcomm offers a software development kit (SDK) for developing and building trusted applications. This SDK includes the necessary build system, header files, and library dependencies to compile trusted applications.

- Trusted applications. See Qualcomm TEE → Trusted applications.
- To develop the SMC invoke trusted applications, global platform trusted applications, and IDL/Object-based Qualcomm TEE, service APIs are available to licensed developers with authorized access. For more information, see Qualcomm Linux Security Guide - Addendum.

3.5 Qualcomm TEE MINK (Secure EL1)

In a Qualcomm TEE kernel-based system, software operates in one or more communicating domains, such as the kernel domain or user domain.

These domains vary in their ability to access memory and other system resources.

- One of these domains, referred to as the kernel domain, executes at secure EL1 and has complete control over the system resources.
- The other domains, known as user domains or processes, run at secure EL0 and have restricted access to system resources. MINK allows for precise control over the access granted to a process.

An IPC primitive is an object capability-based synchronous message passing facility, which allows code execution in one domain to invoke objects running in other domains. The IPC MINK-primitives enable communication between the domains.

3.6 Secure monitor (Secure EL3)

The secure monitor, a component of TrustZone, is responsible for managing the transition between the secure and non-secure worlds.

It operates in Monitor mode, which is activated by running the secure monitor call (SMC) instruction from a privileged Arm mode.

The secure monitor ensures correct context saving and restoration when switching between the non-secure and secure worlds. Additionally, it handles the initial processing of secure interrupts.

3.7 Next steps

- To learn about APIs that can be used to interact with Linux and hardware, see Security APIs.
- To learn how to develop and execute trusted applications and client applications, see Develop trusted and client applications.

4 Security APIs

The security APIs offer the ability to interface with the Linux kernel and the device hardware. They also facilitate various software services that can be run in a trusted execution environment.

4.1 User space APIs

The user space APIs are the functions that the Linux OS accesses to interact with the kernel.

This feature is available to licensed developers with authorized access. If you have access, see User space APIs.

4.2 Interfaces exposed for PKCS#11

See Cryptographic Token Interface Usage Guide and Cryptographic Token Interface Base Specification.

4.3 Kernel APIs

The kernel APIs are the functions that allow the Qualcomm Linux software to interact with the device hardware.

Crypto APIs

The `qcrypto.ko` driver is on the device at `/lib/modules/<version>/kernel/drivers/crypto/qce`. Both kernel-level and user-level APIs can access the crypto engine. For the APIs, see the kernel crypto documentation at Index of crypto documentation.

The following cryptographic algorithms are supported:

- RFC 4309 (CCCM (AES))
- CCM (AES)
- Authenc (HMAC (SHA-256), CBC (AES))
- Authenc (HMAC (SHA-256), CBC (DES3_EDE))

- Authenc (HMAC (SHA-256), CBC (DES))
- Authenc (HMAC (SHA-1), CBC (DES3_EDE))
- Authenc (HMAC (SHA-1), CBC (DES))
- HMAC (SHA-256)
- HMAC (SHA-1)
- SHA-256
- SHA-1
- CBC (DES3_EDE)
- ECB (DES3_EDE)
- CBC (DES)
- ECB (DES)
- XTS (AES)
- CTR (AES)
- CBC (AES)
- ECB (AES)

For more information about the Qualcomm crypto core, see Cryptography.

For more information, see Kernel.org $\rightarrow CryptoAPI$.

Hardware random generator APIs

Qualcomm Linux supports a true random number generator using the `qcom-rng` Linux driver. The random number generated from `qcom-rng` uses kernel crypto for the random number generator API.

In the user space, a random number can be accessed at `/dev/hwrng`. For more information about the hardware random number generator, see the kernel documentation at `https://www.kernel.org/doc/Documentation/hw_random.txt`.

For PRNG APIs, see Pseudo-random number generator APIs. This feature is available to licensed developers with authorized access.

4.4 Qualcomm TEE APIs

Qualcomm TEE provides a collection of APIs that offer services to secure applications. These services include heap management, logging, secure file system access, listener interactions, and cryptography and hashing functions.

This feature is available to licensed developers with authorized access. If you have access, see Qualcomm Linux Security Guide - Addendum → Qualcomm TEE APIs.

4.5 Next steps

- To learn how to develop and execute trusted applications and client applications, see Develop trusted and client applications.
- For sample code and example, see Security services examples.

5 Develop trusted and client applications

You can develop and run trusted and client applications using default files in the global platform interfaces. Trusted applications run in a secure Trusted Execution Environment (TEE) to maintain code and data integrity, while client applications operate in the normal OS, using TEE client APIs to perform secure services.

This feature is available to licensed users with authorized access to develop and execute trusted applications and client applications. If you have access, see Qualcomm Linux Security Guide - Addendum → Develop.

For developing applications that offer hardware-based attestation, zero-touch device provisioning, and chipset feature management, see Qualcomm Linux Wireless Edge Services Guide. This feature is available to licensed users with authorized access.

5.1 Next steps

- To use the security tools needed to sign the images, generate the fuse blower image for secure boot, and generate secure debug policies, see Security tools.
- To initialize and configure the hardware for running securely on Linux, see Bring up security features.

6 Security tools

To develop the TrustZone components and trusted applications on Qualcomm TEE, use the Qualcomm® Snapdragon™ Mobile PC Platform LLVM compiler for Arm® Technology.

You can use SecTools v2 for signing the images, generating the fuse blower image for secure boot, and generating secure debug policies.

6.1 Build system

For more information about the toolchain, build process, and compilation, see Qualcomm Linux Build Guide $\rightarrow GitHubworkflow(firmwareandextras)$.

6.2 SecTools v2 for secure boot

The table lists the documentation for SecTools v2.

Document	Description
SecTools V2: Metabuild Secure Image User Guide	Perform the secure-image operations on metabuild software images.
SecTools V2: Fuse Blower User Guide	Create and sign the fuse blower images. When a device uses a fuse blower image, it causes the specified fuse to be blown.
SecTools V2: ELF Tool User Guide	Generate, add segments, and combine the ELF software images.
SecTools V2: MBN Tool User Guide	Add the modem configuration binary (MBN) headers to binary images.
SecTools V2: ELF Consolidator User Guide	Create the consolidated ELF software images. A consolidated ELF contains the contents of multiple subsystem images.
SecTools V2: Secure Image User Guide	Sign, encrypt, and inspect Qualcomm software images.

Document	Description
SecTools V2: Secure Debug User Guide	Generate and sign the debug policy images to enable device debugging and authentication.

Note: The *SecTools* guides are available to licensed users with authorized access.

6.3 Next steps

- To initialize and configure the hardware for running securely on Linux, see Bring up security features.
- To configure Qualcomm TEE for securing devices that handle sensitive data and run trusted applications, see Configure security services.

7 Bring up security features

Security bring-up is the process of initializing and configuring the hardware to run securely on Linux. This section will help you verify the status of various security features.

The security verification process involves checking and validating the security features of hardware and software components. This process ensures that security features meet the requirements to make the system robust, secure, and ready for further development.

Important:

- Before you begin, set up your infrastructure as described in the Qualcomm Linux Build Guide. This guide also provides information about the common build workflows. After your infrastructure is set up, you can get started by verifying the various security features.
- When you have access to the development kit and have built and loaded the software on the hardware, your infrastructure setup is ready.
- You must enable the secure shell (SSH) in the Permissive mode to securely access your host device. The enforcing mode will be supported in the future. For instructions on how to enable SSH and connect to the device, see Qualcomm Linux Build Guide $\rightarrow Howto \rightarrow SigninusingSSH$.

7.1 Verify TrustZone/Device configuration/Hypervisor image loading

By verifying the TrustZone (secure) environment, you ensure that the device's security architecture is robust and reliable, providing a foundation for secure operations.

The following XBL logs contain information about the TrustZone/Device configuration/Hypervisor image loading.

For example:

- Device configuration (DEVCFG)

```
B - 1206031 - QSEE Dev Config - Image Load, Start
```

```
D - 763 - Auth Metadata
```

```
D - 549 - Segments hash check

D - 13054 - QSEE Dev Config - Image Loaded, Delta -
(53248 Bytes)
```

- TrustZone

```
B - 1228113 - QSEE - Image Load, Start

D - 26382 - Auth Metadata

D - 22234 - Segments hash check

D - 88999 - QSEE - Image Loaded, Delta - (4027792 Bytes)
```

- Hypervisor

```
B - 1402237 - QHEE - Image Load, Start

D - 26383 - Auth Metadata

D - 7045 - Segments hash check

D - 35258 - QHEE - Image Loaded, Delta - (1491024 Bytes)
```

- APDP

```
B - 978348 - APDP - Image Load, Start

D - 42212 - Auth Metadata

D - 458 - Segments hash check

D - 48434 - APDP - Image Loaded, Delta - (17332 Bytes)
```

7.2 Verify secure boot and UEFI secure boot

- Secure boot feature is designed to ensure that a device boots using only software that's trusted by the manufacturer.
- UEFI secure boot is an extension of secure boot that operates within the unified extensible firmware interface (UEFI) environment.
- Enabling secure boot for the hardware is crucial for achieving hardware security and protecting intellectual property. For instructions, see Enable secure boot and Enable UEFI secure boot.
- The XBL logs contain the secure boot status of the device. The logs include information about the boot interface, secure boot status, boot configuration, JTAG_ID, OEM_ID, and serial number.

```
S - Format: Log Type - Time(microsec) - Message -
Optional Info
```

```
S - Log Type: B - Since Boot(Power On Reset),  D - Delta,
 S - Statistic
S - QC_IMAGE_VERSION_STRING=BOOT.MXF.1.0.c1-00037-
KODIAKLA-1
S - IMAGE_VARIANT_STRING=SocKodiakLAA
S - OEM_IMAGE_VERSION_STRING=hu-apasunur-hyd
S - Boot Interface: USB **
S - Secure Boot: On **
S - Boot Config @ 0x00786070 = 0x000000c1
S - JTAG ID @ 0x00786130 = 0x001970e1
S - OEM ID @ 0x00786138 = 0x00000000
S - Serial Number @ 0x00786134 = 0x4172f1dd
```

7.3 Verify SELinux status and customize SELinux policies

- Security-Enhanced Linux (SELinux) is a security architecture integrated into the Linux kernel that provides a mechanism for supporting access control security policies.
- Verifying the SELinux status for the device software prevents unauthorized access to critical software modules or drivers. For instructions, see Enable SELinux.
- SEPolicy is the SELinux policy that defines rules for process and user interactions with system resources, enforcing mandatory access controls (MAC) in Linux.
- Default SEPolicy may not address all the specific requirements of your applications. By creating custom policies, you can define precise rules that align with your application's requirements. For instructions, see Customize SEPolicy.
- To verify the SELinux status, do the following:

  1. Verify the kernel configuration.

     ```
     CONFIG_SECURITY_SELINUX=y
     ```

  2. Connect to the device using SSH.
  3. View the SELinux enable status and other details, by running the `seinfo` commands.

     ```
     Statistics for policy file: /sys/fs/selinux/policy
     Policy Version:             33 (MLS enabled)
     Target Policy:              selinux
     Handle unknown classes:     allow
     Classes:                    131       Permissions:
           423
     Sensitivities:              16        Categories:
     ```

```
        1024
Types:                          4376      Attributes:
        319
```

4. Verify the SELinux enforce status from the console or connect to the device using SSH.

```
getenforce
enforcing - if selinux is set to enable
```

7.4 Verify PIL image loading - Sample logs

The following sample logs show the initialization and status of various remote processors and hardware components, which are part of the boot process managed by the primary image loader (PIL). The PIL is responsible for loading and verifying these components to ensure they're secure and functioning correctly.

WLAN (remoteproc1)	`730, 0x00000000000459B4 | 8.700828: remoteproc remoteproc1: Remote processor 8a00000. Remoteproc is now up`
cDSP (remoteproc3)	`735, 0x00000000000465A3 | 8.794052: remoteproc remoteproc3: Remote processor a300000. Remoteproc is now up`
aDSP (remoteproc2)	`741, 0x00000000000469FC | 8.828033: remoteproc remoteproc2: Remote processor 3000000.remoteproc is now up`
MODEM (remoteproc0)	`1035, 0x000000000006B78B | 13.433955: remoteproc remoteproc0: Remote processor 4080000. Remoteproc is now up`

A660_zap	`sh-5.1# dmesg | grep gfx`
	`[7.822629] kgsl-iommu SoC@0: QCOM, kgsl-iommu@3da0000: gfx3d_user: Adding to iommu group 15`
	`[7.870446] kgsl-3d 3d00000. qcom, kgsl-3d0: bound SoC@0: QCOM, kgsl-iommu@3da0000: gfx3d_user (ops kgsl_mmu_cb_ component_ops [msm_kgsl])`

Video (Vpu20_1v.mbn)	`sh-5.1# dmesg | grep video`
	`[7.094229] videodev: Linux video capture interface: v2. 00`
	`[7.847469] qcom-iris aa00000.video-codec: Adding to iommu group 17`
	`[7.856647] qcom-iris aa00000.video-codec: no reset clocks found`
	`[10.131119] [drm] [msm-dsi-warn]: [nt36672e LCD video mode DSI novatek panel with DSC] fall back to default te-pin-select`
	`[10.188079] [drm:dsi_ display_bind [msm_drm]] [msm-dsi-info]: Successfully bind display panel 'QCOM, mdss_dsi_nt36672e_fhd_plus_ 120_video '`

7.5 Verify Qualcomm TEE mink/global platform TEE API availability

Qualcomm TEE uses various APIs to manage secure applications and interactions with secure peripherals. GlobalPlatform provides standardized APIs for Qualcomm TEE that ensures interoperability and security across devices. The verification process involves checking that these APIs are correctly implemented and available for use.

1. Verify the availability of the library on the device.

   ```
   /usr/lib/
   libmink*
   libGPMTEE*
   libGPTEE*
   ```

2. Verify if the mink listener service is running.

   ```
   ps -ef | grep qtee_supplicant and qtee_supplicant is running
   ps -ef | grep ssgtzd
   ```

7.6 Verify SMC invoke driver status

The secure monitor call (SMC) invoke driver facilitates communication between the TrustZone and the operating system. The SMC invoke driver status includes information about the initialization steps, any errors encountered, and the successful setup of secure communication channels. To verify the SMC invoke driver status, do the following:

1. Connect to the device using SSH.
2. Verify if the `/dev/smcinvoke` node is present:

   ```
   ls -l /dev/smcinvoke
   ```

7.7 Verify RPMB provisioning status

The replay protected memory block (RPMB) is a secure partition in storage devices. The RPMB provisioning ensures that the RPMB is correctly provisioned and ready to securely store and retrieve data. To verify the RPBM provisioning status, do the following:

1. Connect to the device using SSH and run the following command.

   ```
   sh-5.1# rpmbClient smci -p 1
   ```

   The following message is displayed.

   ```

   SMCINVOKE INTERFACE WARNING!!!
   You are about to provision the RPMB key.
   This is a ONE time operation and CANNOT be reversed.
   ______________________________________________________
   ```

```
0 -> Provision Production key
1 -> Provision Test key
2 -> Check RPMB key provision status
________________________________________________________
```

2. Select an option to proceed: 2 RMPB Key status: RPMB_KEY_NOT_PROVISIONED (f)

Caution: Once RPMB is provisioned with test keys on a non-secure device, it becomes irreversible. As a result, secure boot can't be enabled on such devices.

7.8 Verify Qualcomm WES status

- Qualcomm WES helps deploy many edge devices easily and manage them without manual intervention. It includes services like plug-and-play setup, on-demand updates, emergency and routine upgrades, and enabling third-party services throughout the device life-cycle.
- If you need to develop applications that offer hardware-based attestation, zero-touch device provisioning, and chipset feature management, see Develop trusted and client applications.
- To install or upgrade QCS5430 SoftSKU feature packs, see Install or upgrade SoftSKU feature packs.

Note: This feature is available to licensed users with authorized access to verify the Qualcomm WES status. If you have access, see Qualcomm Linux Wireless Edge Services Guide → Bring up Qualcomm WES.

7.9 Verify trusted and client applications

- Trusted applications function within a trusted execution environment (TEE), offering a secure and isolated environment to keep the integrity and confidentiality of code and data.
- Client applications function within the normal world operating system, communicating with trusted applications using TEE client APIs to perform secure services.
- To execute the security-critical applications in the secure world (TrustZone), you must develop your own trusted applications. You must have access to the Qualcomm TEE software development kit (SDK) to develop corresponding client applications that launch the trusted applications. For more information, see Develop trusted and client applications.

Note: This feature is available to licensed users with authorized access to develop and execute trusted applications and client applications. If you have access, see Qualcomm Linux Security Guide - Addendum → Develop.

7.10 Next steps

- To configure Qualcomm TEE for securing devices that handle sensitive data and run trusted applications, see Configure security services.
- To customize memory and SEPolicy, see Customize secuity services.

8 Configure security services

Qualcomm Linux Security provides multiple security configurations for enahancing the device security, maintaining the device software authenticity and integrity and to protect the critical and sensitive developer and user information.

This section guides you through the following configuration workflows.

Secure boot

→ Enable secure boot

→ QFPROM fuses

→ Generate local (insecure) root key and certificate

→ Generate ECDSA root key and certificate

→ Generate RSA client application key pair and certificate

→ Generate SHA-384 hash for RSA and ECDSA

→ Sign images

→ Generate signed sec.elf image

→ Flash images

UEFI secure boot

→ Enable UEFI secure boot

→ Generate key and certificate

→ Sign images and copy (.auth) key/signed files to EFI partition

→ Enable UEFI secure boot from systemd-boot menu

→ Hash unsigned images and update DB for image authentication

Other configurations

8.1 Next steps

- To adjust Qualcomm TEE configurations, see Enable device configuration (Devcfg) from Qualcomm TEE.
- To enable secure boot and to ensure only trusted applications runs on the device, see Enable secure boot.

Enable device configuration (Devcfg) from Qualcomm TEE

Configuring Qualcomm TEE is essential for maintaining the security, compliance, performance, and flexibility of devices that manage sensitive data and run trusted applications. Qualcomm TEE configurations can be adjusted using the device configuration (devcfg) framework, which provides a centralized way to manage and adjust device-specific settings.

Note:

– Remember to run all the SSH commands in the SELinux Permissive mode. The Enforcing mode will be supported in the future.

– For instructions on how to connect to the device, see Qualcomm Linux Build Guide $\rightarrow Howto \rightarrow SigninusingSSH$.

Compile devcfg image from TrustZone

1. Select the configuration options that TrustZone offers through the built in `devcfg.mbn` XML files. For example: `trustzone_images/ssg/securemsm/trustzone/qsee/mink/oem/config/<chipset>/oem_config.xml`.
2. Use the command to compile the devcfg image from TZ.XF.5.29.1.

```
cd trustzone_images/build/ms

python3 build_all.py -b TZ.XF.5.0 CHIPSET=<chipset>
```

```
<devcfg> --cfg=build_config_deploy_<chipset>.xml
```

This step generates the devcfg.mbn image at the following location: `trustzone_images/build/ms/bin/<build_flavor>`.

Note: Use the following devcfg files accordingly:

<devcfg> is

– `devcfg` for QCS6490

– `devcfg_iot` for QCS9100

Important:

The devcfg_iot.mbn file isn't being generated by default. Apply the following changes to build devcfg_iot.mbn.

```
trustzone_images/build/ms/build_config_
deploy_lemans.xml
@@ -60,9 +60,12 @@
<alias build-once="false" disable="false"
internal-test="false" recompile="true" strip=
"false" name="devcfg_auto_sgvm">
<artifact name="devcfg_auto_sgvm"/>
</alias>
+   <alias build-once="false" disable="false"
internal-test="false" recompile="true" strip=
"false" name="devcfg_iot">
+   <artifact name="devcfg_iot"/>
+   </alias>
```

The issue is expected to be resolved in the next release.

For instructions on building and compiling, see Qualcomm Linux Build Guide $\rightarrow GitHubworkflow(firmwareandextras)$.

Customize device using configuration parameters

Use the configuration parameters listed in the following table to customize the device as needed.

Configuration parameters	Description
`OEM_pil_secure_app_load_region_size`	Customizes the TA size.
`OEM_pil_subsys_load_region_start`	Customizes the PIL load start address when there is any change from the default memory map.
`OEM_pil_subsys_load_region_size`	Customizes the PIL size when there is any change from the default memory map.
`OEM_enable_app_fatal_err`	Forces a TrustZone system to fatal error when a specific TA crashes. Use with `OEM_crash_ta_name`.
`OEM_crash_ta_name`	Replaces the entry with the TA name that crashed and the TA on which the secure kernel is expected to crash.
`OEM_sec_wdog_bark_time`	Changes the default configuration of the device for secure watchdog bark time.
`OEM_sec_wdog_bite_time`	Changes the default configuration of the device for secure watchdog bite time.
`OEM_tz_log_level`	Sets the TrustZone log level: – Fatal: 0 – Error: 1 – Debug: 2

Enable RPMB-based SFS anti-rollback protection

To enable or disable the RPMB-based SFS anti-rollback protection, use the following configuration parameter and the XML file.

Configuration parameter

`cmnlib_gppo_rpmb_enablement`, can be set to Enabled or Disabled, where the default value is Enabled and must be changed only when required.

XML file location

```
trustzone_images/ssg/securemsm/trustzone/qsee/mink/oem/
config/common/cmnlib_oem_config.xml
```

Next steps

– To enable secure boot and to ensure only trusted applications runs on the device, see Enable secure boot.

– To enable secure boot, QFPROM fuses must be blown. This is a one-time, irreversible process that permanently sets these values. For more information see QFPROM fuses.

Enable secure boot

Secure boot is enabled by blowing a set of hardware fuses that are part of QFPROM. The hash of the root certificate is blown into the hardware fuse, which serves as the primary RoT.

To understand the off-target preparation and the on-device execution, see the workflow.

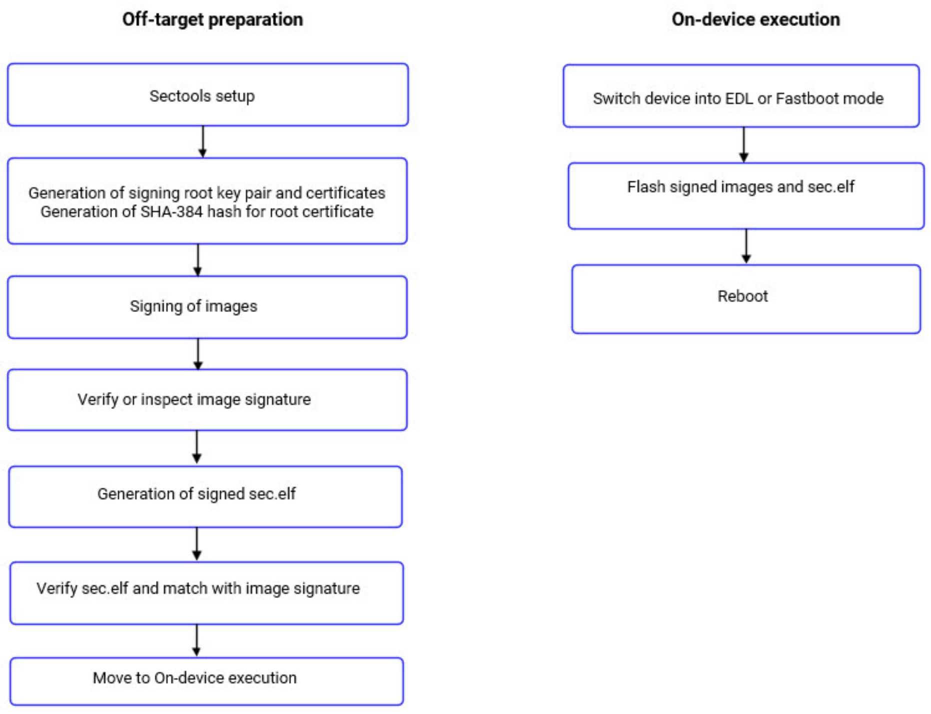

Off-target preparation
Sectools setup
Generation of signing root key pair and certificates
Generation of SHA-384 hash for root certificate
Signing of images
Verify or inspect image signature
Generation of signed sec.elf
Verify sec.elf and match with image signature
Move to On-device execution
On-device execution
Switch device into EDL or Fastboot mode
Flash signed images and sec.elf
Reboot

Figure : Secure boot workflow

Secure boot is guaranteed only after blowing a QFPROM, which is an eFuse. The eFuse configuration required to enable platform secure boot includes:

– Enabling image authentication and anti-rollback protection.

- Disabling debug and JTAG access.
- Blowing *disable read/write permission* fuses for the QFPROM regions.

Enable secure boot by blowing QFPROM fuse

1. Obtain the unique OEM_ID from Qualcomm.

   The ID is required when using the code authorization signing services (CASS) or Qualcomm WES services. Alternatively, you can use 0 as the value for the OEM_ID.
2. Generate and configure signing assets such as keys and certificates.
3. Generate a signed ELF (sec.elf) for blowing fuses.
4. Sign firmware images.
5. Flash sec.elf and signed images to the device.

Images that aren't based on Linux are signed using SecTools v2 and a local signer. This process requires signing certificates and keys to be present on the local machine where the signing takes place. However, these keys aren't secure and can be exposed during and after the signing process on the machine running SecTools.

Note: Qualcomm doesn't provide guidance on how to secure these keys. It's recommended to follow standard security protocols, obtain keys/certificates from a trusted certification authority (CA), and protect these keys using a hardware security module (HSM).

Keys and certificates that are self-generated using the OpenSSL tool don't have a link to any certificate authority.

Enable secure boot using SecTools

1. Set up the environment variable for SecTools v2. For more information, see Security tools.

   The following code snippets are provided as references. Replace `<chipset>` and `<chipset>.LE.X.x` according to the build in use. For example, `<QCM6490.LE.1.0>`.

   Note: The terms metabuild, meta, and meta paths are used interchangeably to indicate the path from the Qualcomm ChipCode™ Portal. The metabuild denotes the complete Qualcomm ChipCode release for a build.

For Linux:

```
setenv SECTOOLS=/<chipset>.LE.X.x/common/
sectoolsv2/ext/linux/sectools
```

```
export SECTOOLS=/<chipset>.LE.X.x/common/
sectoolsv2/ext/linux/sectools
```

2. Use SecTools v2 from the following path:

   `<chipset>.LE.X.x/common/sectoolsv2/ext/<platform>`

   – The `<chipset>_security_profile.xml` file is located in the meta at `<chipset>.LE.X.x/common/sectoolsv2`

   – The minimum SecTools version required is 1.17 or later.

4. It's recommended to use a hardware security module (HSM). However, if a LOCAL (insecure) signer is used, see the LOCAL signing option with -signing-help in SecTools V2: Secure Image User Guide.

5. All the keys and certificate generation commands are run using OpenSSL 1.1.1g (21 Apr 2020). It's a prerequisite to install OpenSSL.

6. Ensure that you enable secure boot on a device that's not RPMB provisioned. See Check RPMB provision status.

   RPMB will be automatically provisioned with production keys after secure boot fuses are blown.

Note: The *SecTools* guides are available to licensed users with authorized access.

Next steps

– To enable secure boot, QFPROM fuses must be blown. This is a one-time, irreversible process that permanently sets these values. For more information see QFPROM fuses.

– To ensure the that the cryptographic keys and certificates are generated and managed in a secure and trusted environment, see Generate local (insecure) root key and certificate.

QFPROM fuses

QFPROM fuses are used to store cryptographic keys that authenticate software images during the secure boot process. This ensures that only authorized software can run on the device.

QFPROM employs a fusing mechanism where registers are programmed by blowing fuses to store permanent data. This is a one-time operation that can’t be undone. The QFPROM fuse values and details are captured in the following table, which enable secure boot once the fuses are blown.

Fuse name	**Start address**	**Bit number**	**Fuse blow value**	**Description**

QCS5430/QCS6490

Fuse name	**Start address**	**Bit number**	**Fuse blow value**	**Description**
Read permissions				
Secondary Key derivation Key Read disable	7801A8	24	1	After provisioning the SKDK, blow this bit to secure the secondary key from being read back. A secure path hardware exists from SKDK to the crypto engine.
Write permissions				
Read permissions write disable	7801B0	6	1	Blow this bit after the region has been provisioned to disable further QFPROM changes to this region.
FEC enables write disable	7801B0	8	1	Blow this bit after the region has been provisioned to disable further QFPROM changes to this region.
OEM configuration write disable	7801B0	9	1	Blow this bit after the region has been provisioned to disable further QFPROM changes to this region.
Public key hash 0 write disable	7801B0	17	1	Blow this bit after the region has been provisioned to disable further QFPROM changes to this region.

Fuse name	Start address	Bit number	Fuse blow value	Description
OEM secure boot write disable	7801B0	23	1	Blow this bit after the region has been provisioned to disable further QFPROM changes to this region.
Secondary key derivation key write disable	7801B0	24	1	Blow this bit after the region has been provisioned to disable further QFPROM changes to this region.
FEC enable				
OEM secure boot FEC enable	7801B8	23	1	To enable FEC for OEM secure boot region, blow this bit. Ensure that the complete region is provisioned before FEC is enabled.
Secondary key derivation key FEC enable	7801B8	24	1	To enable FEC for the secondary KDF key, blow this bit. Ensure that the complete region is provisioned before FEC is enabled.
OEM Config				
`WDOG_EN`	7801C0	14	1	Prevents the `WDOG_DISABLE` GPIO from disabling WDOG, freeing up the GPIO and preventing potential abuse by an attacker.

Fuse name	Start address	Bit number	Fuse blow value	Description
SHARED_ QSEE_SPIDEN_ DISABLE	7801C0	30	1	A shared Qualcomm TEE secure invasive debug disable bucket. A corresponding Qualcomm fuse can override this OEM-controlled fuse.
SHARED_QSEE_ SPNIDEN_ DISABLE	7801C0	31	1	A shared Qualcomm TEE secure non-invasive debug disable bucket. A corresponding Qualcomm fuse can override this OEM-controlled fuse.
SHARED_MSS_ DBGEN_DISABLE	7801C4	32	1	A shared MSS invasive debug disable bucket. A corresponding Qualcomm fuse can override this OEM-controlled fuse.
SHARED_MSS_ NIDEN_DISABLE	7801C4	33	1	A shared MSS non-invasive debug disable bucket. A corresponding Qualcomm fuse can override this OEM-controlled fuse.
SHARED_CP_ DBGEN_DISABLE	7801C4	34	1	A shared CP invasive debug disable bucket. A corresponding Qualcomm fuse can override this OEM-controlled fuse.

Fuse name	**Start address**	**Bit number**	**Fuse blow value**	**Description**
`SHARED_CP_NIDEN_DISABLE`	7801C4	35	1	A shared CP non-invasive debug disable bucket. A corresponding Qualcomm fuse can override this OEM-controlled fuse.
`SHARED_NS_DBGEN_DISABLE`	7801C4	36	1	A shared CP non-invasive debug disable bucket. A corresponding Qualcomm fuse can override this OEM-controlled fuse.
`SHARED_NS_NIDEN_DISABLE`	7801C4	37	1	A shared CP non-invasive debug disable bucket. A corresponding Qualcomm fuse can override this OEM-controlled fuse.
`APPS_DBGEN_DISABLE`	7801C4	38	1	Blow this bit for a secure solution. This configuration disables the application processor global invasive debug capabilities (JTAG and monitor mode). The `OVERRIDE` registers can override this configuration.

Fuse name	Start address	Bit number	Fuse blow value	Description
APPS_NIDEN_ DISABLE	7801C4	39	1	Blow this bit for a secure solution. This configuration disables the application processor global non-invasive debug capabilities (trace and performance monitoring). This configuration can be overridden with the OVERRIDE registers.
SHARED_MISC_ DEBUG_DISABLE	7801C4	40	1	A shared miscellaneous debug disable bucket. A corresponding Qualcomm fuse can override this OEM-controlled fuse.
EKU_ ENFORCEMENT_ EN	7801C8	30	1	To enable enforcement of the EKU field in the certificate, blow this device.
OEM_HW_ID[0: 15]	7801CC	[32:47]	0	Represents the OEM hardware ID. Bits 15:0.
OEM_PRODUCT_ ID[0:15]	7801CC	[48:63]	0	Represents the OEM product ID. Bits 15:0.
ANTI_ ROLLBACK_ FEATURE_EN[0]	7801D4	32	1	– Bit 0 - BOOT_ANTI_ ROLLBACK_EN – Bit 1 - TZAPPS_ ANTI_ROLLBACK_ EN – Bit 2 - PILSUBSYS_ ANTI_ROLLBACK_ EN – Bit 3 - MSA_ANTI_ ROLLBACK_EN
ANTI_ ROLLBACK_ FEATURE_EN[1]	7801D4	33	1	
ANTI_ ROLLBACK_ FEATURE_EN[2]	7801D4	34	1	
ANTI_ ROLLBACK_ FEATURE_EN[3]	7801D4	35	1	

Fuse name	Start address	Bit number	Fuse blow value	Description
PK hash				
`PK hash 0[0:383]`	780248	[0:383]	–	The OEM-specific root certificate PK hash value.
OEM secure boot				
`OEM_SECURE_BOOT1_PK_HASH_IN_FUSE`	780728	4	1	When this bit is ‘1’, use the value stored in OEM_PK_HASH for the root certificate hash.
`OEM_SECURE_BOOT1_AUTH_EN`	780728	5	1	To enable secure boot for apps and other peripheral images, blow this bit. When this bit is ‘1’, it enables authentication for any code that references secure boot configuration 1.
`OEM_SECURE_BOOT2_PK_HASH_IN_FUSE`	780728	12	1	For boot configuration 2: If this bit is ‘0’, use the internal ROM hash index and `OEM_SECURE_BOOT1_ROM_PK_HASH_IDX[3:0]` for the root certificate hash. If this bit is ‘1’, use the value stored in `OEM_PK_HASH` for the root certificate hash.

Fuse name	Start address	Bit number	Fuse blow value	Description
OEM_SECURE_BOOT2_AUTH_EN	780728	13	1	To enable the secure boot, blow this bit. When this bit is ‘1’, it enables authentication for any code that references secure boot.
OEM_SECURE_BOOT3_PK_HASH_IN_FUSE	780728	20	1	For boot configuration 3: If this bit is ‘0’, use the internal ROM hash index and OEM_SECURE_BOOT1_ROM_PK_HASH_IDX[3:0] for the root certificate hash. When this bit is ‘1’, use the value stored in OEM_PK_HASH for the root certificate hash.
OEM_SECURE_BOOT3_AUTH_EN	780728	21	1	To enable the secure boot, blow this bit. When this bit is ‘1’, it enables authentication for any code that references secure boot configuration 3.
Sec key derivation key				

Fuse name	**Start address**	**Bit number**	**Fuse blow value**	**Description**
`Sec Key derivation Key[0:255]`	780738	[0:255]		This 256-bit value is used as the secondary key derivation input, which is used to generate the secondary key for the crypto engine. When running in an insecure mode (no secure boot or Debug enabled), the SKDK is fed into the key derivation function to generate a unique non-secure secondary key for use by the crypto engine. When running in a secure mode (secure boot and debug disabled), the SKDK is fed directly to the crypto engine as the secondary key. After the SKDK value has been correctly programmed, the SKDK Read Disable must be blown to permanently protect the SKDK value. The software reads the SKDK value from the QFPROM before this correction is made. The SBL fuse blow API can automatically generate a random number for use as the SKDK, ensuring that the SKDK value is never available outside of the device.

Fuse name	Start address	Bit number	Fuse blow value	Description

QCS9075/QCS9100

Fuse name	Start address	Bit number	Fuse blow value	Description
Read permissions				
Secondary Key derivation Key Read disable	780190	31	1	After provisioning the SKDK, blow this bit to secure the secondary key from being read back. A secure path hardware exists from SKDK to the crypto engine.
Write permissions				
Read permissions write disable	780198	5	1	Blow this bit after the region has been provisioned to disable further QFPROM changes to this region.
FEC enables write disable	780198	7	1	Blow this bit after the region has been provisioned to disable further QFPROM changes to this region.
OEM configuration write disable	780198	8	1	Blow this bit after the region has been provisioned to disable further QFPROM changes to this region.
Public key hash 0 write disable	780198	24	1	Blow this bit after the region has been provisioned to disable further QFPROM changes to this region.

Fuse name	Start address	Bit number	Fuse blow value	Description
OEM secure boot write disable	780198	30	1	Blow this bit after the region has been provisioned to disable further QFPROM changes to this region.
Secondary key derivation key write disable	780198	31	1	Blow this bit after the region has been provisioned to disable further QFPROM changes to this region.
FEC enable				
OEM secure boot FEC enable	7801A0	30	1	To enable FEC for OEM secure boot region, blow this bit. Ensure that the complete region is provisioned before FEC is enabled.
Secondary key derivation key FEC enable	7801A0	31	1	To enable FEC for the secondary KDF key, blow this bit. Ensure that the complete region is provisioned before FEC is enabled.
OEM Config				
`WDOG_EN`	7801A8	14	1	Prevents the `WDOG_DISABLE` GPIO from disabling WDOG, freeing up the GPIO and preventing potential abuse by an attacker.

Fuse name	Start address	Bit number	Fuse blow value	Description
SHARED_ QSEE_SPIDEN_ DISABLE	7801A8	30	1	A shared Qualcomm TEE secure invasive debug disable bucket. A corresponding Qualcomm fuse can override this OEM-controlled fuse.
SHARED_QSEE_ SPNIDEN_ DISABLE	7801A8	31	1	A shared Qualcomm TEE secure non-invasive debug disable bucket. A corresponding Qualcomm fuse can override this OEM-controlled fuse.
SHARED_MSS_ DBGEN_DISABLE	7801AC	32	1	A shared MSS invasive debug disable bucket. A corresponding Qualcomm fuse can override this OEM-controlled fuse.
SHARED_MSS_ NIDEN_DISABLE	7801AC	33	1	A shared MSS non-invasive debug disable bucket. A corresponding Qualcomm fuse can override this OEM-controlled fuse.
SHARED_CP_ DBGEN_DISABLE	7801AC	34	1	A shared CP invasive debug disable bucket. A corresponding Qualcomm fuse can override this OEM-controlled fuse.

Fuse name	**Start address**	**Bit number**	**Fuse blow value**	**Description**
`SHARED_CP_NIDEN_DISABLE`	7801AC	35	1	A shared CP non-invasive debug disable bucket. A corresponding Qualcomm fuse can override this OEM-controlled fuse.
`SHARED_NS_DBGEN_DISABLE`	7801AC	36	1	A shared CP non-invasive debug disable bucket. A corresponding Qualcomm fuse can override this OEM-controlled fuse.
`SHARED_NS_NIDEN_DISABLE`	7801AC	37	1	A shared CP non-invasive debug disable bucket. A corresponding Qualcomm fuse can override this OEM-controlled fuse.
`APPS_DBGEN_DISABLE`	7801AC	38	1	Blow this bit for a secure solution. This configuration disables the application processor global invasive debug capabilities (JTAG and monitor mode). The `OVERRIDE` registers can override this configuration.

Fuse name	Start address	Bit number	Fuse blow value	Description
APPS_NIDEN_DISABLE	7801AC	39	1	Blow this bit for a secure solution. This configuration disables the application processor global non-invasive debug capabilities (trace and performance monitoring). This configuration can be overridden with the OVERRIDE registers.
SHARED_MISC_DEBUG_DISABLE	7801AC	40	1	A shared miscellaneous debug disable bucket. A corresponding Qualcomm fuse can override this OEM-controlled fuse.
EKU_ENFORCEMENT_EN	7801B0	30	1	To enable enforcement of the EKU field in the certificate, blow this device.
OEM_HW_ID[0:15]	7801B4	[32:47]	0	Represents the OEM hardware ID. Bits 15:0.
OEM_PRODUCT_ID[0:15]	7801B4	[48:63]	0	Represents the OEM product ID. Bits 15:0.
ANTI_ROLLBACK_FEATURE_EN[0]	7801BC	32	1	– Bit 0 - BOOT_ANTI_ROLLBACK_EN – Bit 1 - TZAPPS_ANTI_ROLLBACK_EN – Bit 2 - PILSUBSYS_ANTI_ROLLBACK_EN – Bit 3 - MSA_ANTI_ROLLBACK_EN
ANTI_ROLLBACK_FEATURE_EN[1]	7801BC	33	1	
ANTI_ROLLBACK_FEATURE_EN[2]	7801BC	34	1	
ANTI_ROLLBACK_FEATURE_EN[3]	7801BC	35	1	

Fuse name	Start address	Bit number	Fuse blow value	Description
PK hash				
`PK hash 0[0:383]`	7802A0	[0:383]	–	The OEM-specific root certificate PK hash value.
OEM secure boot				
`OEM_SECURE_BOOT1_PK_HASH_IN_FUSE`	780D78	4	1	When this bit is ‘1’, use the value stored in OEM_PK_HASH for the root certificate hash.
`OEM_SECURE_BOOT1_AUTH_EN`	780D78	5	1	To enable secure boot for apps and other peripheral images, blow this bit. When this bit is ‘1’, it enables authentication for any code that references secure boot configuration 1.
`OEM_SECURE_BOOT2_PK_HASH_IN_FUSE`	780D78	12	1	For boot configuration 2: If this bit is ‘0’, use the internal ROM hash index and `OEM_SECURE_BOOT1_ROM_PK_HASH_IDX[3:0]` for the root certificate hash. If this bit is ‘1’, use the value stored in `OEM_PK_HASH` for the root certificate hash.

Fuse name	Start address	Bit number	Fuse blow value	Description
`OEM_SECURE_BOOT2_AUTH_EN`	780D78	13	1	To enable the secure boot, blow this bit. When this bit is ‘1’, it enables authentication for any code that references secure boot.
`OEM_SECURE_BOOT3_PK_HASH_IN_FUSE`	780D78	20	1	For boot configuration 3: If this bit is ‘0’, use the internal ROM hash index and `OEM_SECURE_BOOT1_ROM_PK_HASH_IDX[3:0]` for the root certificate hash. When this bit is ‘1’, use the value stored in `OEM_PK_HASH` for the root certificate hash.
`OEM_SECURE_BOOT3_AUTH_EN`	780D78	21	1	To enable the secure boot, blow this bit. When this bit is ‘1’, it enables authentication for any code that references secure boot configuration 3.
Sec key derivation key				

Fuse name	Start address	Bit number	Fuse blow value	Description
`Sec Key derivation Key[0:255]`	780D88	[0:255]		This 256-bit value is used as the secondary key derivation input, which is used to generate the secondary key for the crypto engine. When running in an insecure mode (no secure boot or Debug enabled), the SKDK is fed into the key derivation function to generate a unique non-secure secondary key for use by the crypto engine. When running in a secure mode (secure boot and debug disabled), the SKDK is fed directly to the crypto engine as the secondary key. After the SKDK value has been correctly programmed, the SKDK Read Disable must be blown to permanently protect the SKDK value. The software reads the SKDK value from the QFPROM before this correction is made. The SBL fuse blow API can automatically generate a random number for use as the SKDK, ensuring that the SKDK value is never available outside of the device.

Fuse name	Start address	Bit number	Fuse blow value	Description

QCS8275

Fuse name	Start address	Bit number	Fuse blow value	Description
Read permissions				
Secondary Key derivation Key Read disable	780190	31	1	After provisioning the SKDK, blow this bit to secure the secondary key from being read back. A secure path hardware exists from SKDK to the crypto engine.
Write permissions				
Read permissions write disable	780198	5	1	Blow this bit after the region has been provisioned to disable further QFPROM changes to this region.
FEC enables write disable	780198	7	1	Blow this bit after the region has been provisioned to disable further QFPROM changes to this region.
OEM configuration write disable	780198	8	1	Blow this bit after the region has been provisioned to disable further QFPROM changes to this region.
Public key hash 0 write disable	780198	24	1	Blow this bit after the region has been provisioned to disable further QFPROM changes to this region.

Fuse name	Start address	Bit number	Fuse blow value	Description
OEM secure boot write disable	780198	30	1	Blow this bit after the region has been provisioned to disable further QFPROM changes to this region.
Secondary key derivation key write disable	780198	31	1	Blow this bit after the region has been provisioned to disable further QFPROM changes to this region.
FEC enable				
OEM secure boot FEC enable	7801A0	30	1	To enable FEC for OEM secure boot region, blow this bit. Ensure that the complete region is provisioned before FEC is enabled.
Secondary key derivation key FEC enable	7801A0	31	1	To enable FEC for the secondary KDF key, blow this bit. Ensure that the complete region is provisioned before FEC is enabled.
OEM Config				
`WDOG_EN`	7801A8	14	1	Prevents the `WDOG_DISABLE` GPIO from disabling WDOG, freeing up the GPIO and preventing potential abuse by an attacker.

Fuse name	Start address	Bit number	Fuse blow value	Description
SHARED_ QSEE_SPIDEN_ DISABLE	7801A8	30	1	A shared Qualcomm TEE secure invasive debug disable bucket. A corresponding Qualcomm fuse can override this OEM-controlled fuse.
SHARED_QSEE_ SPNIDEN_ DISABLE	7801A8	31	1	A shared Qualcomm TEE secure non-invasive debug disable bucket. A corresponding Qualcomm fuse can override this OEM-controlled fuse.
SHARED_MSS_ DBGEN_DISABLE	7801AC	32	1	A shared MSS invasive debug disable bucket. A corresponding Qualcomm fuse can override this OEM-controlled fuse.
SHARED_MSS_ NIDEN_DISABLE	7801AC	33	1	A shared MSS non-invasive debug disable bucket. A corresponding Qualcomm fuse can override this OEM-controlled fuse.
SHARED_CP_ DBGEN_DISABLE	7801AC	34	1	A shared CP invasive debug disable bucket. A corresponding Qualcomm fuse can override this OEM-controlled fuse.

Fuse name	Start address	Bit number	Fuse blow value	Description
SHARED_CP_ NIDEN_DISABLE	7801AC	35	1	A shared CP non-invasive debug disable bucket. A corresponding Qualcomm fuse can override this OEM-controlled fuse.
SHARED_NS_ DBGEN_DISABLE	7801AC	36	1	A shared CP non-invasive debug disable bucket. A corresponding Qualcomm fuse can override this OEM-controlled fuse.
SHARED_NS_ NIDEN_DISABLE	7801AC	37	1	A shared CP non-invasive debug disable bucket. A corresponding Qualcomm fuse can override this OEM-controlled fuse.
APPS_DBGEN_ DISABLE	7801AC	38	1	Blow this bit for a secure solution. This configuration disables the application processor global invasive debug capabilities (JTAG and monitor mode). The `OVERRIDE` registers can override this configuration.

Fuse name	Start address	Bit number	Fuse blow value	Description
APPS_NIDEN_ DISABLE	7801AC	39	1	Blow this bit for a secure solution. This configuration disables the application processor global non-invasive debug capabilities (trace and performance monitoring). This configuration can be overridden with the OVERRIDE registers.
SHARED_MISC_ DEBUG_DISABLE	7801AC	40	1	A shared miscellaneous debug disable bucket. A corresponding Qualcomm fuse can override this OEM-controlled fuse.
EKU_ ENFORCEMENT_ EN	7801B0	30	1	To enable enforcement of the EKU field in the certificate, blow this device.
OEM_HW_ID[0: 15]	7801B4	[32:47]	0	Represents the OEM hardware ID. Bits 15:0.
OEM_PRODUCT_ ID[0:15]	7801B4	[48:63]	0	Represents the OEM product ID. Bits 15:0.
ANTI_ ROLLBACK_ FEATURE_EN[0]	7801BC	32	1	– Bit 0 - BOOT_ANTI_ ROLLBACK_EN – Bit 1 - TZAPPS_ ANTI_ROLLBACK_ EN – Bit 2 - PILSUBSYS_ ANTI_ROLLBACK_ EN – Bit 3 - MSA_ANTI_ ROLLBACK_EN
ANTI_ ROLLBACK_ FEATURE_EN[1]	7801BC	33	1	
ANTI_ ROLLBACK_ FEATURE_EN[2]	77801BC	34	1	
ANTI_ ROLLBACK_ FEATURE_EN[3]	7801BC	35	1	

Fuse name	**Start address**	**Bit number**	**Fuse blow value**	**Description**
PK hash				
`PK hash 0[0: 383]`	7802A0	[0:383]	–	The OEM-specific root certificate PK hash value.
OEM secure boot				
`OEM_SECURE_BOOT1_PK_HASH_IN_FUSE`	780DA8	4	1	When this bit is ‘1’, use the value stored in OEM_PK_HASH for the root certificate hash.
`OEM_SECURE_BOOT1_AUTH_EN`	780DA8	5	1	To enable secure boot for apps and other peripheral images, blow this bit. When this bit is ‘1’, it enables authentication for any code that references secure boot configuration 1.
`OEM_SECURE_BOOT2_PK_HASH_IN_FUSE`	780DA8	12	1	For boot configuration 2: If this bit is ‘0’, use the internal ROM hash index and `OEM_SECURE_BOOT1_ROM_PK_HASH_IDX[3:0]` for the root certificate hash. If this bit is ‘1’, use the value stored in `OEM_PK_HASH` for the root certificate hash.

Fuse name	Start address	Bit number	Fuse blow value	Description
`OEM_SECURE_BOOT2_AUTH_EN`	780DA8	13	1	To enable the secure boot, blow this bit. When this bit is ‘1’, it enables authentication for any code that references secure boot.
`OEM_SECURE_BOOT3_PK_HASH_IN_FUSE`	780DA8	20	1	For boot configuration 3: If this bit is ‘0’, use the internal ROM hash index and `OEM_SECURE_BOOT1_ROM_PK_HASH_IDX[3:0]` for the root certificate hash. When this bit is ‘1’, use the value stored in `OEM_PK_HASH` for the root certificate hash.
`OEM_SECURE_BOOT3_AUTH_EN`	780DA8	21	1	To enable the secure boot, blow this bit. When this bit is ‘1’, it enables authentication for any code that references secure boot configuration 3.
Sec key derivation key				

Fuse name	Start address	Bit number	Fuse blow value	Description
`Sec Key derivation Key[0:255]`	780DB8	[0:255]		This 256-bit value is used as the secondary key derivation input, which is used to generate the secondary key for the crypto engine. When running in an insecure mode (no secure boot or Debug enabled), the SKDK is fed into the key derivation function to generate a unique non-secure secondary key for use by the crypto engine. When running in a secure mode (secure boot and debug disabled), the SKDK is fed directly to the crypto engine as the secondary key. After the SKDK value has been correctly programmed, the SKDK Read Disable must be blown to permanently protect the SKDK value. The software reads the SKDK value from the QFPROM before this correction is made. The SBL fuse blow API can automatically generate a random number for use as the SKDK, ensuring that the SKDK value is never available outside of the device.

Fuse name	Start address	Bit number	Fuse blow value	Description

Next steps

- To ensure the that the cryptographic keys and certificates are generated and managed in a secure and trusted environment, see Generate local (insecure) root key and certificate.
- To enhance device security by providing stronger cryptographic protection and better performance, you must generate the ECDSA root key and certificate. For more information, see Generate ECDSA root key and certificate.

Generate local (insecure) root key and certificate

The version 3 (v3 and v3_attest) extensions define the certificate format and establish the Certificate Authority (CA). This process allows you to create a local CA with specific attributes and constraints set by the v3 extensions, allowing you to issue certificates for testing and development purposes.

Follow these steps to generate a local root key and certificate.

1. To create `opensslroot.cfg` file, see Sample OpenSSL configuration.
2. To create the `v3.ext` and `v3_attest.ext` extensions, use the following:
   - `v3.ext`: This extension can be found at /docs/manmaster/man5/x509v3_config.html (OpenSSL.org), and include the following settings:

     ```
     authorityKeyIdentifier=keyid,issuer
     subjectKeyIdentifier=hash
     basicConstraints=CA:true,pathlen:0
     keyUsage=keyCertSign
     ```

   - `v3_attest.ext`: This extension can be found at /docs/manmaster/man5/x509v3_config.html (OpenSSL.org), and include the following settings:

     ```
     authorityKeyIdentifier=keyid,issuer
     basicConstraints=CA:FALSE,pathlen:0
     keyUsage=digitalSignature
     extendedKeyUsage=codeSigning
     ```

3. Prepare the environment, create a directory named OEM-KEYS to generate

all certificates and keys at one location.

– For Linux, use the following commands:

```
cd /path/to/sectools/$ mkdir ./OEM-KEYS &&
cp /download/opensslroot.cfg ./OEM-KEYS &&
cp /download/v3.ext ./OEM-KEYS &&
cp /download/v3_attest.ext ./OEM-KEYS
```

– For Windows, copy `opensslroot.cfg`, `v3.ext`, and `v3_attest.ext` to the OEM-KEYS directory.

The supported algorithms include:

– Rivest–Shamir–Adleman (RSA) signature algorithm

– Elliptical curve digital signature algorithm (ECDSA)

Next steps

– To enhance device security by providing stronger cryptographic protection and better performance, you must generate the ECDSA root key and certificate. For more information, see Generate ECDSA root key and certificate.

– To allow client applications to authenticate securely and enable encrypted communication, see Generate RSA CA key pair and certificate.

Generate ECDSA root key and certificate

ECDSA offers superior security and performance compared to the RSA signature algorithm. As a result, the default configuration in SecTools supports ECDSA signing.

The following types of keys are created with ECDSA: - The public key, which is accessible to everyone. - The private key, which is only known to the owner of the key pair.

You can modify and run the following ECDSA-specific commands to generate the root key and certificate:

1. Go to the `OEM-KEYS` directory and generate the ECDSA root key and certificate:

```
cd ./OEM-KEYS

openssl ecparam -genkey -name secp384r1 -outform PEM -
out qpsa_rootca.key
```

```
openssl req -new -key qpsa_rootca.key -sha384 -out
rootca_pem.crt -subj '/C=US/CN=Generated OEM Root CA/
OU=CDMA Technologies/OU=General Use OEM Key (OEM
should update all fields)/L=San Diego/O=SecTools/
ST=California' -config opensslroot.cfg -x509 -days
7300 -set_serial 1
```

```
openssl x509 -in rootca_pem.crt -inform PEM -out qpsa_
rootca.cer -outform DER
```

2. Generate the intermediate Certificate Authority (CA) key pair and certificate:

```
openssl ecparam -genkey -name secp384r1 -outform PEM -
out qpsa_attestca.key
```

```
openssl req -new -key qpsa_attestca.key -out ca.CSR -
subj '/C=US/ST=California/CN=Generated OEM Attestation
CA/O=SecTools/L=San Diego' -config opensslroot.cfg -
sha384
```

```
openssl x509 -req -in ca.CSR -CA rootca_pem.crt -CAkey
qpsa_rootca.key -out ca_pem.crt -set_serial 1 -days
7300 -extfile v3.ext -sha384 -CAcreateserial
```

```
openssl x509 -inform PEM -in ca_pem.crt -outform DER -
out qpsa_attestca.cer
```

Next steps

– To allow client applications to authenticate securely and enable encrypted communication, see Generate RSA CA key pair and certificate.

– To enhance security, ensure data integrity, and support secure digital signatures, see Generate SHA-384 hash for RSA and ECDSA.

Generate RSA client application key pair and certificate

RSA is an encryption algorithm that uses a pair of keys to encrypt and decrypt data, ensuring secure data transmission.

A private key and a public key are created with RSA:

– The public key is accessible to anyone.

– The private key is only known to the owner of the key pair.

Either the public or private key can encrypt the data, and the other key decrypts it. Follow these steps to generate an RSA client application key pair and certificate.

1. To generate the root client application key pair and certificate, run the following commands:

   The key size used is 2048. However, a key size of 4096 is also supported.

```
openssl genrsa -out qpsa_rootca.key 2048
```

```
openssl req -new -sha256 -key qpsa_rootca.key -x509 -
out rootca_pem.crt -subj /C=US/ST=California/L="San
Diego"/OU="General Use Test Key (for testing 13 only)
"/OU="CDMA Technologies"/O=QUALCOMM/CN="QCT Root CA 1"
-days 7300 -set_serial 1 -config opensslroot.cfg -
sigopt rsa_padding_mode:pss -sigopt rsa_pss_saltlen:-1
-sigopt digest:sha256
```

```
openssl x509 -in rootca_pem.crt -inform PEM -out qpsa_
rootca.cer -outform DER
```

```
openssl x509 -text -inform DER -in qpsa_rootca.cer
```

2. To generate the attestation client application key pair and certificate, run the following commands using RSA with a key size of 2048:

```
openssl genrsa -out qpsa_attestca.key 2048
```

```
openssl req -new -key qpsa_attestca.key -out attestca.
csr -subj /C=US/ST=CA/L="San Diego"/OU="CDMA
Technologies"/O=QUALCOMM/CN="QUALCOMM Attestation CA"
-days 7300 -config opensslroot.cfg
```

```
openssl x509 -req -in attestca.csr -CA rootca_pem.crt
-CAkey qpsa_rootca.key -out attestca_pem.crt -set_
serial 5 -days 7300 -extfile v3.ext -sha256 -sigopt
```

```
rsa_padding_mode:pss -sigopt rsa_pss_saltlen:-1 -
sigopt digest:sha256
```

```
openssl x509 -inform PEM -in attestca_pem.crt -outform
DER -out qpsa_attestca.cer
```

Next steps

- To enhance security, ensure data integrity, and support secure digital signatures, see Generate SHA-384 hash for RSA and ECDSA.
- To verify that the software hasn't been tampered with and is from a trusted source, see Generate signed sec.elf image.

Generate SHA-384 hash for RSA and ECDSA

The SHA-384 hash is crucial in cryptographic applications for several reasons, including enhancing security strength, creating digital signatures, ensuring compliance with standards, and future-proofing.

To generate the SHA-384 hash of the root certificate, run the following command:

```
openssl dgst -sha384 qpsa_rootca.cer >
sha384rootcert.txt
```

Next steps

- To ensure the authenticity and integrity of software images, see Sign images.
- To verify that the software hasn't been tampered with and is from a trusted source, see Generate signed sec.elf image.

Sign images

Image signing is a security process that involves adding a cryptographic signature to a digital image. This signature serves as a unique identifier, verifying the authenticity, integrity, and origin of the image. Without image signing, there is no assurance of an image's integrity or trusted origin, leading to potential security breaches and data loss.

Follow these steps to sign the images.

1. You can sign the images using SecTools V2. Different signing methods and secure image functionality are available. For more information see SecTools V2: Secure Image User Guide.
2. You can generate the keys and certificates using a local signer. For more information, see Generate local (insecure) root key and certificate.
3. To sign a single image, run the following command, where `tz.mbn` is used as an example.

   Note: You can replace the values of oem-id "**0x1**" and oem-product-id "**0xabcd**" according to your requirement.

```
<meta>/common/sectoolsv2/ext/linux/sectools secure-
image --sign /path/to/tz.mbn --image-id=TZ --security-
profile <meta>/common/sectoolsv2/<chipset>_security_
profile.xml --oem-id=0x1 --oem-product-id=0xabcd --
anti-rollback-version=0x0 --signing-mode=LOCAL --root-
certificate=./OEM-KEYS/qpsa_rootca.cer --ca-
certificate=./OEM-KEYS/qpsa_attestca.cer --ca-key=./
OEM-KEYS/qpsa_attestca.key --outfile ./signed_images_
out/tz.mbn
```

   The following is a sample command for QCS9075/QCS9100.

```
<meta>/common/sectoolsv2/ext/linux/sectools
secure-image --sign /path/to/tz.mbn --image-
id=TZ --security-profile <meta>/common/
sectoolsv2/lemans_security_profile.xml --oem-
id=0x1 --oem-product-id=0xabcd --anti-rollback-
version=0x0 --signing-mode=LOCAL --root-
certificate=./OEM-KEYS/qpsa_rootca.cer --ca-
certificate=./OEM-KEYS/qpsa_attestca.cer --ca-
key=./OEM-KEYS/qpsa_attestca.key --outfile ./
signed_images_out/tz.mbn
```

4. For the images that should be split, use the `--pil-split` option.
5. For signing the complete metabuild, use the following commands.

```
./sectools metabuild-secure-image --image-finder
/prj/qct/asw/crmbuilds/snowcone/builds901/PROD/
QCM6490.LE.1.0.r1-00186-STD.INT.SL-1/common/
build/app/image_finder.py --sign --oem-id=0x1 --
oem-product-id=0xabcd --anti-rollback-
version=0x0 --signing-mode LOCAL --root-
certificate=./OEM-KEYS/qpsa_rootca.cer --ca-
certificate=./OEM-KEYS/qpsa_attestca.cer --ca-
key=./OEM-KEYS/qpsa_attestca.key --chipset
KODIAK --outdir meta_signing_output/
```

   For more information, see SecTools V2: Metabuild Secure Image User Guide.

Note: The *SecTools* guides are available to licensed users with authorized access.

Next steps

- To verify that the software hasn't been tampered with and is from a trusted source, see Generate signed sec.elf image
- To write a complete software image to a storage device that ensures that the device is updated, functional, secure, and optimized, see Flash images.

Generate signed sec.elf image

Generating a signed sec.elf image involves creating a secure executable and linkable format (ELF) file with a cryptographic signature. Signing this image ensures its authenticity, integrity, and origin.

A fuse blower binary is used to permanently disable certain functionalities or components of a device for security reasons. Generating a signed sec.elf image along with a fuse blower binary involves a series of steps to ensure both the integrity of the firmware and the security of the device. To generate fuse blower binary, see SecTools V2: Fuse Blower User Guide.

Integrate sample commands using SecTools

This section provides sample commands only. The following are sample commands for SecTools on Windows.

Note:

- You can replace the values of oem-id "**0x1**" and oem-product-id "**0xabcd**" according to your requirement.
- You can replace the value of "**–fuse-pk-hash-0**" with the **SHA384** of "**OEM-KEYS/qpsa_rootca.cer**."

  To calculate the correct PK_HASH value, use the following command:

  ```
  openssl dgst -sha384 qpsa_rootca.cer
  ```

  For more information, see Generate SHA-384 hash for RSA and ECDSA.
- Replace the digest generated here from the user Root cert in the sec.elf generation command below.

- Stage 1: Basic secure boot (image authentication + OEMID + MODEL ID)

  Run the following command:

  QCS5430/QCS6490

  ```
  <meta>/common/sectoolsv2/ext/linux/sectools fuse-
  blower --security-profile <meta>/common/
  sectoolsv2/kodiak_security_profile.xml --fuse-pk-
  hash-0=<sha384 of OEM-KEYS/qpsa_rootca.cer> --
  fuse-oem-secure-boot1-pk-hash-in-fuse --fuse-oem-
  secure-boot1-auth-en --fuse-oem-secure-boot2-pk-
  hash-in-fuse --fuse-oem-secure-boot2-auth-en --
  fuse-oem-secure-boot3-pk-hash-in-fuse --fuse-oem-
  secure-boot3-auth-en --fuse-oem-hw-id=0x0001 --
  fuse-oem-product-id=0xabcd --generate --sign --
  signing-mode=LOCAL --root-certificate=./OEM-KEYS/
  qpsa_rootca.cer --ca-certificate=./OEM-KEYS/qpsa_
  attestca.cer --ca-key=./OEM-KEYS/qpsa_attestca.
  key --oem-id=0x1 --oem-product-id=0xabcd --
  outfile basic_sec.elf
  ```

QCS9075/QCS9100

```
<meta>/common/sectoolsv2/ext/linux/sectools fuse-
blower --security-profile <meta>/common/sectoolsv
2/ext/<platform>/sectools.exe fuse-blower --secur
ity-profile <meta>/common/sectoolsv2/lemans_secur
ity_profile.xml --fuse-pk-hash-0=0xf953644308944b
b811ca0ec2a736a17fe38509941ce7f55860130857813c837
8e93359b70dfd874c270dca08a53bd99f --fuse-oem-secu
re-boot1-pk-hash-in-fuse --fuse-oem-secure-boot1-
auth-en --fuse-oem-secure-boot2-pk-hash-in-fuse -
-fuse-oem-secure-boot2-auth-en --fuse-oem-secure-
boot3-pk-hash-in-fuse --fuse-oem-secure-boot3-aut
h-en --fuse-oem-hw-id=0x0001 --fuse-oem-product-i
d=0xabcd --generate --sign --signing-mode=LOCAL -
-root-certificate=./OEM-KEYS/qpsa_rootca.cer --ca
-certificate=./OEM-KEYS/qpsa_attestca.cer --ca-ke
y=./OEM-KEYS/qpsa_attestca.key --oem-id=0x1 --oem
-product-id=0xabcd --outfile basic_sec.elf
```

– Stage 2: Complete secure boot (basic secure boot + debug disable + anti-rollback + write permission disable):

  Run the following commands.

QCS5430/QCS6490

```
<meta>/common/sectoolsv2/ext/linux/sectools fuse-
blower --security-profile <meta\>/common/sectools
v2/kodiak_security_profile.xml --fuse-pk-hash-0=0
xf953644308944bb811ca0ec2a736a17fe38509941ce7f558
60130857813c8378e93359b70dfd874c270dca08a53bd99f
--fuse-oem-secure-boot1-pk-hash-in-fuse --fuse-oe
m-secure-boot1-auth-en --fuse-oem-secure-boot2-pk
-hash-in-fuse --fuse-oem-secure-boot2-auth-en --f
use-oem-secure-boot3-pk-hash-in-fuse --fuse-oem-s
ecure-boot3-auth-en --fuse-oem-secure-boot-fec-en
able --fuse-wdog-en --fuse-shared-qsee-spiden-dis
able --fuse-shared-qsee-spniden-disable --fuse-sh
ared-mss-dbgen-disable --fuse-shared-mss-niden-di
sable --fuse-shared-cp-dbgen-disable --fuse-share
d-cp-niden-disable --fuse-shared-ns-dbgen-disable
--fuse-shared-ns-niden-disable --fuse-apps-dbgen-
disable --fuse-apps-niden-disable --fuse-shared-m
```

```
isc-debug-disable --fuse-eku-enforcement-en --fus
e-anti-rollback-feature-en=0xF --fuse-sec-key-der
ivation-key=0x00 --fuse-read-permissions-write-di
sable --fuse-oem-configuration-write-disable --fu
se-secondary-key-derivation-key-read-disable
-fuse-write-permission-write-disable
--fuse-write-permissions-write-disable
--fuse-public-key-hash-0-write-disable --fuse-
oem-secure-boot-write-disable --fuse-secondary-
key-derivation-key-write-disable --fuse-
secondary-key-derivation-key-fec-enable --fuse-
fec-enables-write-disable --generate --sign --
fuse-oem-hw-id=0x0001 --fuse-oem-product-
id=0xabcd --signing-mode=LOCAL --root-
certificate=./OEM-KEYS/qpsa_rootca.cer --ca-
certificate=./OEM-KEYS/qpsa_attestca.cer --ca-
key=./OEM-KEYS/qpsa_attestca.key --oem-id=0x1 --
oem-product-id=0xabcd --outfile sec.elf
```

QCS9075/QCS9100

```
<meta>/common/sectoolsv2/ext/Linux/sectools  --fu
se-blower --security-profile <meta>/common/sectoo
lsv2/lemans_security_profile.xml --fuse-pk-hash-0
=0xf953644308944bb811ca0ec2a736a17fe38509941ce7f5
5860130857813c8378e93359b70dfd874c270dca08a53bd99
f --fuse-oem-secure-boot1-pk-hash-in-fuse --fuse-
oem-secure-boot1-auth-en --fuse-oem-secure-boot2-
pk-hash-in-fuse --fuse-oem-secure-boot2-auth-en -
-fuse-oem-secure-boot3-pk-hash-in-fuse --fuse-oem
-secure-boot3-auth-en --fuse-oem-secure-boot-fec-
enable --fuse-wdog-en --fuse-shared-qsee-spiden-d
isable --fuse-shared-qsee-spniden-disable --fuse-
shared-mss-dbgen-disable --fuse-shared-mss-niden-
disable --fuse-shared-cp-dbgen-disable --fuse-sha
red-cp-niden-disable --fuse-shared-ns-dbgen-disab
le --fuse-shared-ns-niden-disable --fuse-apps-dbg
en-disable --fuse-apps-niden-disable --fuse-share
d-misc-debug-disable --fuse-eku-enforcement-en --
fuse-anti-rollback-feature-en=0xF --fuse-sec-key-
derivation-key=0x00 --fuse-read-permissions-write
-disable --fuse-oem-configuration-write-disable -
```

```
-fuse-secondary-key-derivation-key-read-disable -
-fuse-public-key-hash-0-write-disable --fuse-oem-
secure-boot-write-disable --fuse-secondary-key-de
rivation-key-write-disable --fuse-secondary-key-d
erivation-key-fec-enable --fuse-fec-enables-write
-disable --generate --sign --fuse-oem-hw-id=0x000
1 --fuse-oem-product-id=0xabcd --signing-mode=LOC
AL --root-certificate=./OEM-KEYS/qpsa_rootca.cer
--ca-certificate=./OEM-KEYS/qpsa_attestca.cer --c
a-key=./OEM-KEYS/qpsa_attestca.key --oem-id=0x1 -
-oem-product-id=0xabcd --outfile sec.elf
```

Note: The *SecTools* guides are available to licensed users with authorized access.

Next steps

- To write a complete software image to a storage device that ensures that the device is updated, functional, secure, and optimized, see Flash images.
- To enforce strict access controls, see Enable SELinux.

Flash images

Flashing images involves writing an entire image, including partitions, file systems, and data, onto a storage device. This process helps maintain the functionality, security, and performance of the device.

Follow these steps to flash the images:

1. See QFPROM fuses for the list of fuses to configure.
2. Replace all the binaries with the signed binaries including `prog_firehose_ddr.elf`.
3. To flash all the signed binaries to the device, see Qualcomm Linux Build Guide → $BuildwithQSCCLI$.
4. After generating the signed images and `sec.elf`, enable secure boot:
   a. Flash the signed images first without `sec.elf` and ensure that the device boots successfully.
   b. Flash the signed images and `sec.elf` using the flash procedure

   from Qualcomm Linux Build Guide → *Flashimages*.

   c. Verify that the secure boot is enabled using Bring up → Verified secure boot.

5. When the secure boot is enabled, the device expects images to be flashed using a secure programming method called validated image programming (VIP). In this release, you can proceed with flashing the images on the secure device by disabling VIP using the following workaround programmer (`prog_firehose_ddr.elf`) image at: `<>/BOOT.MXF.1.0.c1/boot_images/boot/QcomPkg/Library/DevPrgLib/devprg_transfer.c`

6. Set the `vip->state` to `VIP_DISABLED` irrespective of the secure boot enable check in the following function:

```
int devprg_transfer_init(void)
{
  int secboot, result;
  struct vip_data *vip = &vip_data;
  devprg_init_vip_state();
  secboot = devprg_is_secure_boot_enabled();
//   if (secboot == 0) /*comment this to set vip state
to VIP_DISABLED
    vip->state = VIP_DISABLED;
  result = devprg_transport_init();
  return result;
}
```

7. To rebuild `prog_firehose_ddr.elf`, see Qualcomm Linux Build Guide → *GitHubworkflow(firmwareandextras)*.

8. If any of the PIL signed images aren't flashed using PCAT, follow these steps to push the PIL images manually using SCP:

```
push adsp, cdsp, modem, wlan, ipa pil split binaries
```

   For instructions, see Qualcomm Linux Build Guide → *Howto* → *SigninusingSSH*.

   a. Copy and replace the PIL split bins and the `.mdt` files generated in the signed output to the `<<QCM6490.LE.x.x>/common/build/ufs/bin/QCM6490_fw/lib/firmware/qcom/qcm6490/` directory.

   b. Connect to the device as the root using SSH. For instructions, see Qualcomm Linux Build Guide → *Howto* → *SigninusingSSH*.

      Run the following command:

```
mount -o rw,remount  /
scp <QCM6490.LE.x.x>/common/build/ufs/bin/QCM6490_
fw/lib/firmware/qcom/qcm6490/. root@<IP_address>:/
lib/firmware/qcom/qcm6490/
Push gfx (a660_zap) pil split binaries, a660_zap.
mdt and a660_zap.mbn from signed outout
scp <a660_zap signed output folder>/. root@<IP_
address>:/lib/firmware/
Push signed Venus binary:
scp vpu20_1v.mbn root@<IP_address>:/lib/firmware/
qcom/vpu-2.0/
reboot
```

9. To check for PIL loading success, check for the following logs in dmesg:

```
[    7.597009] remoteproc remoteproc0: Booting fw
image qcom/qcs6490/modem.mdt, size 6052
[    8.095883] remoteproc remoteproc0: remote
processor 4080000.remoteproc is now up
[    5.938938] remoteproc remoteproc1: Booting fw
image qcom/qcs6490/wpss.mdt, size 4612
[    6.088524] remoteproc remoteproc1: remote
processor 8a00000.remoteproc is now up
[    5.951047] remoteproc remoteproc2: Booting fw
image qcom/qcs6490/adsp.mdt, size 6852
[    6.107310] remoteproc remoteproc2: remote
processor 3000000.remoteproc is now up
[    5.977966] remoteproc remoteproc3: Booting fw
image qcom/qcs6490/cdsp.mdt, size 5252
[    6.135802] remoteproc remoteproc3: remote
processor a300000.remoteproc is now up
```

Next steps

- To enforce strict access controls, see Enable SELinux.
- To create cryptographic keys and certificates to authenticate and encrypt communications, see Generate key and certificate.

Enable SELinux

When SELinux is enabled, all system objects, including files, directories, processes, sockets, drivers, and more, are labeled with a security context.

A security context consists of a user, role, type identifier, and optional sensitivity, separated by colons.

For example: `user:role:type:sensitivity`

Note: *User* is unrelated to a Linux user, and *Type* is unrelated to the kind of object it is.

- A set of valid users, roles, and types is defined in the policy.
- Different objects are labeled with the same security context.
- The MAC mechanism of SELinux security policies is implemented using:
  - Type enforcement (TE)
  - Role-based access control (RBAC)
  - Multilevel security (MLS)
- Types enable the policy to specify the allowed operations.

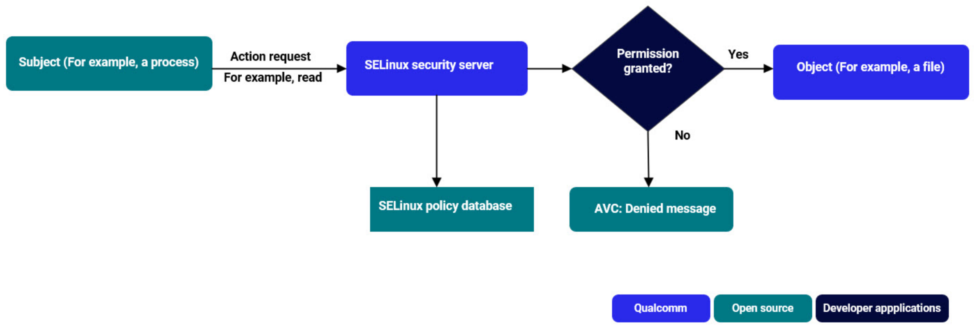


Figure : SELinux process

The following procedures explains how to verify and enable SELinux and modify SELinux modes.

Note: By default, **SELinux is disabled** to simplify the validation process while working with the SoC and SDK. For commercial use, it's recommended to enable the SELinux security feature.

Verify and modify SELinux mode

Caution: If SELinux is enabled, you may not be allowed to update the anti-rollback protection flag.

1. Check the current SELinux configuration of the device (Enforcing or Permissive mode):

   ```
   getenforce
   ```

2. If it's set to the Enforcing mode, run the `setenforce` command to change the mode.

   a. Connect to the device using SSH.

   b. Change the SELinux mode by using the following commands.

   – To switch the device to Enforcing mode:

   ```
   setenforce 1
   ```

   – To switch the device to Permissive mode:

   ```
   setenforce 0
   ```

   – To recheck the current configuration of the device (Enforcing or Permissive mode):

   ```
   getenforce
   ```

Configure SELinux (Enable, disable, and switch modes)

To switch to Enforcing mode (restrictive) or Permissive mode (non-restrictive with logging), follow these steps:

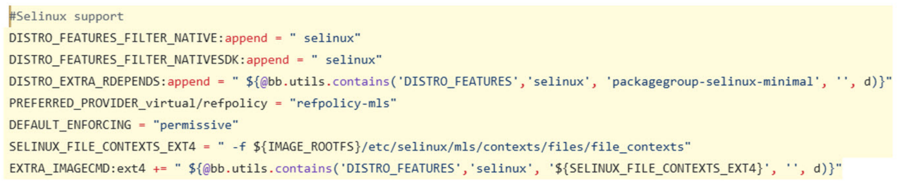
1. To enable or disable SELinux:

   – To disable SELinux for the build, delete the lines of code. By default, these lines are already removed from the distro section, which results in SELinux being disabled.

   – To enable SELinux, add the lines of code as shown in the figure to enable SELinux.

   – Use policy version 33.

   – To add policies for SELinux, see upstream refpolicy. The following figure shows the steps in a SELinux:



```
#Selinux support
DISTRO_FEATURES_FILTER_NATIVE:append = " selinux"
DISTRO_FEATURES_FILTER_NATIVESDK:append = " selinux"
DISTRO_EXTRA_RDEPENDS:append = " ${@bb.utils.contains('DISTRO_FEATURES','selinux', 'packagegroup-selinux-minimal', '', d)}"
PREFERRED_PROVIDER_virtual/refpolicy = "refpolicy-mls"
DEFAULT_ENFORCING = "permissive"
SELINUX_FILE_CONTEXTS_EXT4 = " -f ${IMAGE_ROOTFS}/etc/selinux/mls/contexts/files/file_contexts"
EXTRA_IMAGECMD:ext4 += " ${@bb.utils.contains('DISTRO_FEATURES','selinux', '${SELINUX_FILE_CONTEXTS_EXT4}', '', d)}"
```

2. Check the system status with `getenforce` on target. This command returns one of the three values:
   - Enforcing
   - Permissive
   - Disabled

3. To change the mode, select a mode at runtime by running `setenforce` with a number (this change won't persist after reboot).

Command	**Result**
`setenforce 1`	Switch to Enforcing mode
`setenforce 0`	Switch to Permissive mode

   a. To persist after reboot:

      i. Connect to the device using SSH. For instructions, see Qualcomm Linux Build Guide $\rightarrow Howto \rightarrow SigninusingSSH$.

      ii. Edit SELINUX= to one of the three supported values: `enforcing`, `permissive`, or `disabled` in `/etc/selinux/config`.

      iii. Reboot the device using the following command:

```
reboot
```

   b. To specify the SELinux mode in the build: Change the `DEFAULT_ENFORCING` build flag to one of the three supported values: enforcing, permissive, or disabled.

```
conf/distro/include/qcom-base.inc
-- DEFAULT_ENFORCING = "permissive"
++ DEFAULT_ENFORCING = "enforcing"
```

4. The SELinux Disabled mode leaves behind many code paths that go through the SELinux framework. These code paths aren't useful for KPI testing or verifying bugs in the SELinux framework. It also doesn't allow any more access than Permissive mode.

To disable the feature for testing, remove SELinux from `DISTRO_FEATURES`:

```
conf/distro/include/qcom-base.inc
-- DISTRO_FEATURES:append = " selinux"
```

Next steps

– To ensure that only the verified and trusted applications are loaded during the startup process, see Enable UEFI secure boot.

– To create cryptographic keys and certificates to authenticate and encrypt communications, see Generate key and certificate.

Enable UEFI secure boot

UEFI secure boot enhances the security and reliability of the system by ensuring that only the verified and trusted software loads during startup.

Configure an UEFI secure boot to generate keys and certificates

You can setup an initial UEFI secure boot configuration and convert the keys and certificates into a format that UEFI can understand. See the workflow to understand the off-target preparation and the on-device execution.

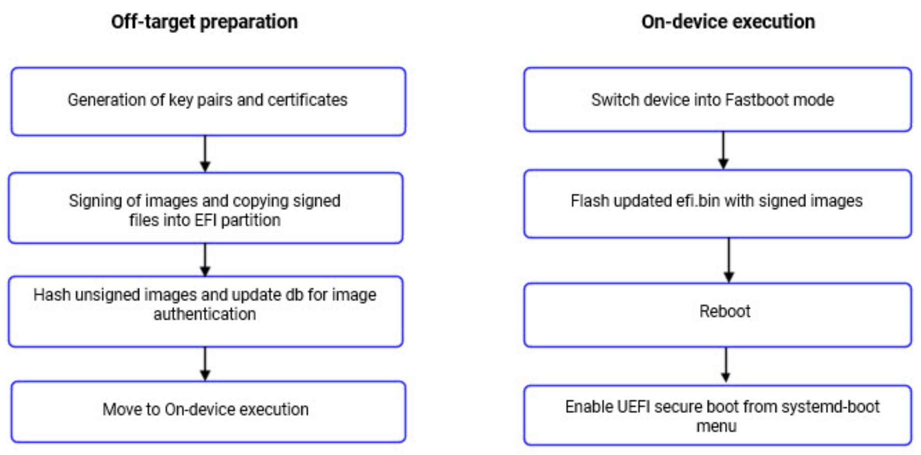

Off-target preparation
Generation of key pairs and certificates
Signing of images and copying signed files into EFI partition
Hash unsigned images and update db for image authentication
Move to On-device execution
On-device execution
Switch device into Fastboot mode
Flash updated efi.bin with signed images
Reboot
Enable UEFI secure boot from systemd-boot menu

Figure : UEFI secure boot workflow

Prerequisites Secure communications and cryptography are facilitated by the OpenSSL toolkit, while keys and signatures for UEFI secure boot are managed by efitool.

Install OpenSSL and efitools

1. Install OpenSSL 0.9.80 June 2010 (or later version) on the Linux host computer.

2.**Install the efitools using the following:**

   – cert-to-efi-sig-list: converts OpenSSL certificates to EFI signature lists

   – sign-efi-sig-list: signs the EFI signature list

   – hash-efi-sig-list: creates a hash signature list entry from a binary

Next steps

– To create cryptographic keys and certificates to authenticate and encrypt communications, see Generate key and certificate.

– For the system to verify and load only trusted software during startup, see Sign images and copy (.auth) key/signed files to EFI partition.

Generate key and certificate

To enable UEFI secure boot, generate a pair of keys and certificates for signing and authentication.

The key generation supports the following algorithms:

– RSA 2048/4096 with SHA-256/SHA384 hash algorithm

– ECDSA secp256r1/secp384r1

The following procedures provide instructions to generate keys and certificates with RSA 2048 and SHA-256 as an example.

Note:

– Create a directory and run the commands in the same location to perform these steps on a Linux machine.

– For ECC, replace `rsa:2048` with `ec:secp384r1` or `ec:secp256r1`. For SHA384, replace `-sha256` with `-sha384` in the following commands.

Generate UID

You can generate a GUID and create three new keys with self-signed certificates in CRT/PEM format and keys in `.key` format:

GUID uses `uuidgen` to generate the signature owner GUID:

```
uuidgen --random > GUID.txt
```

Create PK key

1. Create a PK key pair (RSA-2048) and certificate:

   ```
   openssl req -new -x509 -newkey rsa:2048 -subj "/
   CN=Custom PK/" -keyout PK.key -out PK.crt -days 3650 -
   nodes -sha256
   ```

2. Convert the `.crt` file into the `.cer` file:

   ```
   openssl x509 -outform der -in PK.crt -out PK.cer
   ```

3. Convert the `.crt` file into the `.esl` file:

   ```
   cert-to-efi-sig-list -g "$(< GUID.txt)" PK.crt PK.esl
   ```

4. Sign and generate the `.auth` file with the `.crt`, `.esl`, and `.key` files:

   ```
   sign-efi-sig-list -k PK.key -c PK.crt PK PK.esl PK.
   auth
   ```

Create KEK key

1. Create a KEK key pair (RSA-2048) and certificate:

   ```
   openssl req -new -x509 -newkey rsa:2048 -subj "/
   CN=Custom KEK/" -keyout KEK.key -out KEK.crt -days
   3650 -nodes -sha256
   ```

2. Convert the `.crt` file into the `.cer` file:

   ```
   openssl x509 -outform der -in KEK.crt -out KEK.cer
   ```

3. Convert the `.crt` file into the `.esl` file:

```
cert-to-efi-sig-list -g "$(< GUID.txt)" KEK.crt KEK.
esl
```

4. Sign and generate the `.auth` file with the `.crt`, `.esl`, and `.key` files:

```
sign-efi-sig-list -k PK.key -c PK.crt KEK KEK.esl KEK.
auth
```

Create dB key

1. Create a dB key pair (RSA-2048) and certificate:

```
openssl req -new -x509 -newkey rsa:2048 -subj "/
CN=Custom DB Signing Key 1/" -keyout db.key -out db.
crt -days 3650 -nodes -sha256
```

2. Convert the `.crt` file into the `.cer` file:

```
openssl x509 -outform der -in db.crt -out db.cer
```

3. Convert the `.crt` file into the `.esl` file:

```
cert-to-efi-sig-list -g "$(< GUID.txt)" db.crt db.esl
```

4. Sign and generate the `.auth` file with the `.crt`, `.esl`, and `.key` files:

```
sign-efi-sig-list -k KEK.key -c KEK.crt db db.esl db.
auth
```

Next steps

– For the system to verify and load only trusted software during startup, see Sign images and copy (.auth) key/signed files to EFI partition.

– To ensure that the secure boot settings are correctly applied and maintained through the systemd-boot menu, see Enable UEFI secure boot from systemd-boot menu.

Sign images and copy (.auth) key/signed files to EFI partition

The EFI system partition consists of EFI, loader, and ostree with information relevant to EFI when using systemd-boot. The DTB partition consists of dtb directories.

The EFI system partition holds essential files for booting the system and managing updates, while the DTB partition contains hardware configuration information. This section provides instructions to:

– Sign various images.

– Copy `(.auth)` key and signed files to EFI partition and DTB partition directories.

– Signed and executable images such as the `bootaa64.efi` file (systemd-boot) are placed in the `efimountedbin/EFI/BOOT/` directory and the `vmlinuz.x.x.xx` file (Linux) image is placed in the `efimountedbin/ostree/poky-xxx/vmlinuz-x.x.xx` directory.

The systemd-boot validates the signed images and is also used to enroll the following:

– UEFI secure boot keys are placed in a specific directory in `/keys` for key enrollment. The systemd-boot uses these keys and provisions them in the RPMB or UEFI variable store during UEFI boot time services.

– You can configure the wait time (in seconds) in the systemd-boot loader configuration. Kernel loading is delayed during the wait time, allowing you to review and select available options in the systemd-boot menu.

– Device tree files are stored in the `dtbmountedbin/dtb` directory. These files are used by UEFI during runtime, and the device tree files are initialized. While signing, `.sig` files are created and placed in the same directory as these files are non- PE images. .. container:: nohighlight

Table : EFI system partition (efi.bin)

/EFI	/Loader	/ostree
/Boot/ bootaa64.efi	loader.conf	poky-xxx/ vmlinuz-x.x.xx

	`/keys/authkeys/db.auth` `/keys/authkeys/KEK.auth` `/keys/authkeys/PK.auth`	

Table : DTB partition (dtb.bin)

`combined-dtb.dtb`	`combined-dtb.sig`	`/loader`
		`/keys/authkeys/db.auth` `/keys/authkeys/KEK.auth` `/keys/authkeys/PK.auth`

Place signed images and keys in EFI partition

Follow these steps to place the signed images and keys in an EFI partition on a Linux host machine.

1. Locate the `efi.bin` and `dtb.bin` file paths in the `contents.xml`, file to obtain the `efi.bin` and *dtb.bin*` files from the meta.
2. Mount the `efi.bin` file into the `<workspace>` directory and create an `efimountedbin` directory within the `<workspace>` directory.
3. Mount the `dtb.bin` file into the <workspace> directory and create a `dtbmountedbin` directory within the <workspace> directory.
4. Mount the `efi.bin` file:

```
sudo mount efi.bin efimountedbin
```

```
cd efimountedbin
```

5. Mount the `dtb.bin` file:

   ```
   sudo mount dtb.bin dtbmountedbin
   ```

   ```
   cd dtbmountedbin
   ```

6. Create an authkeys directory within the `<workspace>/efimountedbin/loader/keys` directory to enroll keys.

7. Select and copy the `.auth` files (`PK.auth`, `KEK.auth`, and `db.auth`) to the authkeys directory.

   ```
   sudo cp <selected algo PK/KEK/DB auth files from the
   files location>
   <workspace>/efimountedbin/loader/keys/authkeys/
   ```

8. Create an `authkeys` directory within the `<workspace>/ dtbmountedbin/loader/keys directory` to enroll keys.

9. Select and copy the `.auth files` (PK.auth, KEK.auth, and dB.auth) to the authkeys directory in `dtbmountedbin`.

   ```
   sudo cp <selected algo PK/KEK/DB auth files from the
   files location> <workspace>/dtbmountedbin/loader/keys/
   authkeys/
   ```

10. Sign the `bootaa64.efi, uki.efi and dtb`, `vmlinuz-x.x.xx`, and `combined-dtb.dtb` image files with the keys and copy to the respective directories in the `efimountedbin` directory.

    a. Sign `efi` images:

       The sbsign tool is designed for signing EFI boot images, such as `bootaa64.efior UKI.efi` that follow EFI specifications. This tool, which is used for UEFI secure boot signing is available for download and use on Linux systems. It's important to note that sbsign can only sign PE images with a `.efi` extension.

       i. Copy the `bootaa64.efi` file from the `/efimountedbin` directory `/EFI/BOOT` and the `vmlinuz-x.x.xx` file from the `/ostree/poky-xxx/vmlinuz.x.x.xx ` directory to the :file:`images` directory on your Linux machine.

       ii. Sign the images:

```
cd <workspace>/images
```

```
sudo sbsign --key <workspace>/keys/db.key --
cert <workspace>/keys/db.crt bootaa64.efi --
output <workspace>/bootaa64.efi
```

```
sudo sbsign --key <workspace>/keys/db.key --
cert <workspace>/keys/db.crt vmlinuz.x.x.xx --
output <workspace>/vmlinuz.x.x.xx
```

b. Sign the `dtb` image:

   All images authenticated by UEFI secure boot are regular APIs and typically in the PE format. The signature header and size are appended to the existing PE header, and the signature is appended at the end of the signed file.

   However, when images in non- PE formats require UEFI secure boot authentication, the absence of the PE header and its magic number to recognize the image format fail. As a result, it’s not possible to use standard tools and paths for image verification.

   Currently, among the list of images that UEFI secure boot verifies, only the dtb files are in non- PE format images. As an alternative to the sbsign tool, you can use the `OpenSSL cms` command to generate signature files for signing images in non- PE format.

   Follow these steps for signing non-EFI images:

   i. To sign the dtb file and signature file, run the following command:

```
openssl cms -sign -inkey < .key file > -signer
< .crt file > -binary -in <input dtb file>-out
< Output .dtb.sig file > -outform DER
```

   ii. To sign the image, run the following command:

```
cd <workspace>/images
```

```
sudo openssl cms -sign -inkey <workspace>/
keys/db.key -signer <workspace>/keys/db.crt -
binary -in combined-dtb.dtb --out combined-
dtb.sig -outform DER
```

11. Copy the signed `combined-dtb.sig`, `vmlinuz.x.x.xx`, and `bootaa64.efi` images back to their respective directories

(`dtbmountedbin/`, `efimountedbin/ostree/poky-xxx/`, and `efimountedbin/EFI/BOOT/`).

12. Configure the wait time in systemd-boot:

    a. Open and edit the `loader.conf` file at `/loader/loader.conf` with sudo access:

    ```
    sudo vi loader.conf
    ```

    b. Add the line `timeout 2` to set the boot menu timeout and save the file.

13. To unmount the EFI binary to retrieve the latest `efi.bin` file, run the command:

    ```
    sudo umount efimountedbin
    ```

14. To unmount the DTB binary to retrieve the latest `dtb.bin` file, run the command:

    ```
    sudo umount dtbmountedbin
    ```

15. Securely place the signed images and keys in the EFI partition on target.

    Bring the device into the Fastboot mode and flash the latest `efi.bin` file with the fastboot command:

    ```
    fastboot flash efi <efi binary location>

    fastboot flash dtb_a <dtb binary location>
    ```

For more information, see quic/host-signing-tool.

Next steps

– To ensure that the secure boot settings are correctly applied and maintained through the systemd-boot menu, see Enable UEFI secure boot from systemd-boot menu.

– If no new keys or certificates are added or modified, and if no UEFI secure boot authentication is needed for the existing images, then see Hash unsigned images and update dB for image authentication.

Enable UEFI secure boot from systemd-boot menu

The EFI binary is composed of signed images and secure boot keys, which are generated and then flashed into the system. For more details, see Sign images and copy (.auth) key/signed files to EFI partition.

When the UEFI is loaded and run during the next bootup, the systemd-boot manager displays the **EnrollSecure Boot keys: authkeys** and **Ubuntu1 8.04.6 LTS** menu options on the screen.

Note: These options are displayed when a timeout is configured. For more information, see `loader.conf` settings in Sign images and copy (.auth) key/signed files to EFI partition.

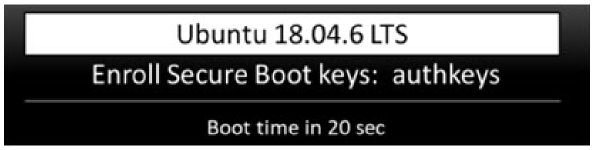
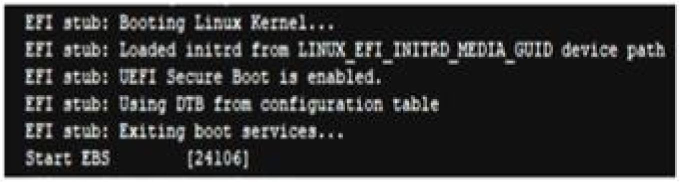


Figure : Systemd-boot menu options

You can use the volume **+/-** buttons to navigate and select the appropriate option to enroll the keys.

When the key is successfully enrolled, UEFI automatically switches from **SetupMode** to **UserMode**. The logs for the UEFI secure boot enablement are listed in the serial logs when the systemd-boot triggers a system reset to apply the changes.

By default, UEFI starts in **UserMode**, and the UEFI secure boot is initialized during the next boot cycle. The logs for the UEFI secure boot enablement are listed in the serial logs when thesystemd-boot transfers control to the `KERNEL EFI STUB`.



```
EFI stub: Booting Linux Kernel...
EFI stub: Loaded initrd from LINUX_EFI_INITRD_MEDIA_GUID device path
EFI stub: UEFI Secure Boot is enabled.
EFI stub: Using DTB from configuration table
EFI stub: Exiting boot services...
Start EBS        [24106]
```

Figure : UEFI secure boot enablement information from serial log

The option to enroll with systemd-boot is only available once. This release doesn’t support the reprovisioning and updating of UEFI secure boot keys.

Next steps

- If no new keys or certificates are added or modified, and if no UEFI secure boot authentication is needed for existing images, then see Hash unsigned images and updatedB for image authentication.
- For configuration files used to generate keys and certificate requests, see Sample OpenSSL configuration.

Hash unsigned images and update dB for image authentication

UEFI secure boot allows image authentication. This authentication is achieved through the hash of images stored in the signature database (dB), even if the images aren't signed or the certificates in the images aren't present in the dB.

This process is reserved for content that can't be signed or altered from its vendor-provided state. If the image hash is available in the database deny (dBX) list, the trust of signed binaries can be removed without having to revoke the corresponding certificates or keys. This is useful, for example, when dealing with an earlier signed boot loader that's vulnerable to recent exploits.

It's redundant to apply a signature and create a dB hash for the same binary. Follow these steps if the image composition doesn't require any changes, meaning no new keys and certificates are being added or modified in the image, and no UEFI secure boot authentication is needed for the existing images.

You can calculate the hash of images and generate an allowed signature dB file.

Generate db.auth file for unsigned images

1. Generate a hash of all images to be verified and convert the hash into an `.esl` file:

```
hash-to-efi-sig-list <list of efis to be hashed>
<output file name with .esl extension>
```

2. Sign the `.esl` hash file with the dB key:

```
sign-efi-sig-list -k < .key file location > -c < .crt
file location > <secure variable name> <Above
generated .esl file> <o/p .auth file>
```

3. Copy the generated `db.auth` file into the EFI binary and provision the keys into the device.

For example, on a Linux host machine:

1. Mount the `efi.bin` file to the `<workspace>` directory and create an `efimountedbin` folder in the `<workspace>` directory.
2. Create a `testkeys` folder in the `<workspace>` directory on the Linux machine and copy the pre-existing keys to it.
3. Sign the images:

   ```
   hash-to-efi-sig-list <workspace>/efimountedbin/EFI/
   BOOT/bootaa64.efi <workspace>/efimountedbin/EFI/Linux/
   uki.efi mergedhash.esl
   sign-efi-sig-list -k keys db.key -c db.crt db
   mergedhash.esl db.auth
   ```

4. Copy the `db.auth` file to the `qckeys` folder at `<workspace>/efimountedbin/loader/keys/qckeys`.
5. Follow the dtb signing steps and sign the dtb images to generate a new `efi.bin` file. For more information, see Sign images and copy (.auth) key/signed files to EFI partition.
6. For a Linux host machine on the target:
   a. Erase any existing UEFI secure boot keys and flash the EFI binary with fastboot.
   b. Provision keys with systemd-boot. For more information, see Enable UEFI secure boot from systemd-boot menu.

   Note: All unsigned files are signed with other keys and authenticated with UEFI using this method.

Next steps

– For configuration files used to generate keys and certificate requests, see Sample OpenSSL configuration.

– For chipset feature management and to upgrade the chipset feature packs, see Customize security services.

Sample OpenSSL configuration

OpenSSL is an open-source toolkit for secure sockets layer (SSL) and transport layer security (TLS) protocols, offering cryptographic functions and a command-line tool.

This sample OpenSSL configuration file is used for generating certificate requests and managing a Certificate Authority (CA).

```
#
#  Copyright (c) 2013 Qualcomm Technologies, Inc.
#  All Rights Reserved.
#  Confidential and Proprietary - Qualcomm Technologies,
Inc.
#
# OpenSSL example configuration file.
# This is mostly being used for generation of certificate
requests.
#

# This definition stops the following lines choking if
HOME isn't
# defined.
HOME            = .
RANDFILE        = $ENV::HOME/.rnd

# Extra OBJECT IDENTIFIER info:
#oid_file       = $ENV::HOME/.oid
oid_section     = new_oids

# To use this configuration file with the "-extfile"
option of the
# "openssl x509" utility, name here the section
containing the
# X.509v3 extensions to use:
# extensions        =
# (Alternatively, use a configuration file that has only
# X.509v3 extensions in its main [= default] section.)

[ new_oids ]

# We can add new OIDs in here for use by 'ca' and 'req'.
# Add a simple OID like this:
# testoid1=1.2.3.4
# Or use config file substitution like this:
```

```
# testoid2=${testoid1}.5.6

##########################################################
###########
[ ca ]
default_ca  = CA_default        # The default ca section

##########################################################
###########
[ CA_default ]

dir     = ./demoCA      # Where everything is kept
certs       = $dir/certs        # Where the issued certs
are kept
crl_dir     = $dir/crl      # Where the issued crl are
kept
database    = $dir/index.txt    # database index file.
#unique_subject = no            # Set to 'no' to allow
creation of
                    # several certificates with same
subject.
new_certs_dir   = $dir/newcerts     # default place for
new certs.

certificate = $dir/cacert.pem   # The CA certificate
serial      = $dir/serial       # The current serial
number
crlnumber   = $dir/crlnumber    # the current crl number
                    # must be commented out to leave a V1
CRL
crl     = $dir/crl.pem      # The current CRL
private_key = $dir/private/cakey.pem# The private key
RANDFILE    = $dir/private/.rand    # private random
number file

x509_extensions = usr_cert      # The extensions to add
to the cert

# Comment out the following two lines for the
"traditional"
# (and highly broken) format.
name_opt    = ca_default        # Subject Name options
cert_opt    = ca_default        # Certificate field
options
```

```
# Extension copying option: use with caution.
# copy_extensions = copy

# Extensions to add to a CRL. Note: Netscape communicator
chokes on V2 CRLs
# so this is commented out by default to leave a V1 CRL.
# crlnumber must also be commented out to leave a V1 CRL.
# crl_extensions    = crl_ext

default_days    = 365           # how long to certify for
default_crl_days= 30            # how long before next
CRL
default_md  = sha1          # which md to use.
preserve    = no            # keep passed DN ordering

# A few different ways of specifying how similar the
request should look
# For type CA, the listed attributes must be the same,
and the optional
# and supplied fields are just that :-)
policy      = policy_match

# For the CA policy
[ policy_match ]
countryName     = match
stateOrProvinceName = match
organizationName    = match
organizationalUnitName  = optional
commonName      = supplied
emailAddress        = optional

# For the 'anything' policy
# At this point in time, you must list all acceptable
'object'
# types.
[ policy_anything ]
countryName     = optional
stateOrProvinceName = optional
localityName        = optional
organizationName    = optional
organizationalUnitName  = optional
commonName      = supplied
emailAddress        = optional
```

```
#########################################################
###########
[ req ]
default_bits        = 1024
default_keyfile     = privkey.pem
distinguished_name  = req_distinguished_name
attributes      = req_attributes
x509_extensions = v3_ca # The extensions to add to the
self signed cert

# Passwords for private keys if not present they will be
prompted for
# input_password = secret
# output_password = secret

# This sets a mask for permitted string types. There are
several options.
# default: PrintableString, T61String, BMPString.
# pkix   : PrintableString, BMPString.
# utf8only: only UTF8Strings.
# nombstr : PrintableString, T61String (no BMPStrings or
UTF8Strings).
# MASK:XXXX a literal mask value.
# WARNING: current versions of Netscape crash on
BMPStrings or UTF8Strings
# so use this option with caution!
string_mask = nombstr

req_extensions = v3_req # The extensions to add to a
certificate request

[ req_distinguished_name ]
countryName         = Country Name (2 letter code)
countryName_default     = AU
countryName_min         = 2
countryName_max         = 2

stateOrProvinceName     = State or Province Name (full
name)
stateOrProvinceName_default = Some-State

localityName            = Locality Name (eg, city)
```

```
0.organizationName       = Organization Name (eg, company)
0.organizationName_default  = Internet Widgits Pty Ltd

# we can do this but it is not needed normally :-)
#1.organizationName      = Second Organization Name (eg,
company)
#1.organizationName_default = World Wide Web Pty Ltd

organizationalUnitName       = Organizational Unit Name
(eg, section)
#organizationalUnitName_default =

commonName           = Common Name (eg, YOUR name)
commonName_max           = 64

emailAddress             = Email Address
emailAddress_max         = 64

# SET-ex3            = SET extension number 3

[ req_attributes ]
challengePassword        = A challenge password
challengePassword_min        = 4
challengePassword_max        = 20

unstructuredName         = An optional company name

[ usr_cert ]

# These extensions are added when 'ca' signs a request.

# This goes against PKIX guidelines but some CAs do it
and some software
# requires this to avoid interpreting an end user
certificate as a CA.

basicConstraints=CA:FALSE
keyUsage = nonRepudiation, digitalSignature,
keyEncipherment

# Here are some examples of the usage of nsCertType. If
it is omitted
# the certificate can be used for anything *except*
object signing.
```

```
# This is OK for an SSL server.
# nsCertType            = server

# For an object signing certificate this would be used.
# nsCertType = objsign

# For normal client use this is typical
# nsCertType = client, email

# and for everything including object signing:
# nsCertType = client, email, objsign

# This is typical in keyUsage for a client certificate.
# keyUsage = nonRepudiation, digitalSignature,
keyEncipherment

# This will be displayed in Netscape's comment listbox.
nsComment           = "OpenSSL Generated Certificate"

# PKIX recommendations harmless if included in all
certificates.
subjectKeyIdentifier=hash
authorityKeyIdentifier=keyid,issuer

# This stuff is for subjectAltName and issuerAltname.
# Import the email address.
# subjectAltName=email:copy
# An alternative to produce certificates that aren't
# deprecated according to PKIX.
# subjectAltName=email:move

# Copy subject details
# issuerAltName=issuer:copy

#nsCaRevocationUrl      = http://www.domain.dom/ca-crl.
pem
#nsBaseUrl
#nsRevocationUrl
#nsRenewalUrl
#nsCaPolicyUrl
#nsSslServerName

[ v3_req ]
```

```

# Extensions to add to a certificate request

subjectKeyIdentifier=hash

#authorityKeyIdentifier=keyid:always,issuer:always

basicConstraints = CA:FALSE
keyUsage = nonRepudiation, digitalSignature,
keyEncipherment

[ v3_ca ]


# Extensions for a typical CA


# PKIX recommendation.

subjectKeyIdentifier=hash

#authorityKeyIdentifier=keyid:always,issuer:always

# This is what PKIX recommends but some broken software
chokes on critical
# extensions.
#basicConstraints = critical,CA:true
# So we do this instead.
basicConstraints = CA:true

# Key usage: this is typical for a CA certificate.
However since it will
# prevent it from being used as a test self-signed
certificate, it is best to be
# left out as a default.
keyUsage = cRLSign, keyCertSign

# Some might want this also
# nsCertType = sslCA, emailCA

# Include email address in subject alt name: another PKIX
recommendation
# subjectAltName=email:copy
# Copy issuer details
# issuerAltName=issuer:copy

# DER hex encoding of an extension: beware experts only!
# obj=DER:02:03
# Where 'obj' is a standard or added object
# You can even override a supported extension:
# basicConstraints= critical, DER:30:03:01:01:FF

[ crl_ext ]

# CRL extensions.
# Only issuerAltName and authorityKeyIdentifier make any
sense in a CRL.

# issuerAltName=issuer:copy
authorityKeyIdentifier=keyid:always,issuer:always

[ proxy_cert_ext ]
# These extensions should be added when creating a proxy
certificate

# This goes against PKIX guidelines but some CAs do it
and some software
# requires this to avoid interpreting an end user
certificate as a CA.

basicConstraints=CA:FALSE

# Here are some examples of the usage of nsCertType. If
it is omitted
# the certificate can be used for anything *except*
object signing.

# This is OK for an SSL server.
# nsCertType            = server

# For an object signing certificate this would be used.
# nsCertType = objsign

# For normal client use this is typical
# nsCertType = client, email

# and for everything including object signing:
# nsCertType = client, email, objsign

# This is typical in keyUsage for a client certificate.
# keyUsage = nonRepudiation, digitalSignature,
keyEncipherment

# This will be displayed in Netscape's comment listbox.
nsComment           = "OpenSSL Generated Certificate"

# PKIX recommendations harmless if included in all
certificates.
subjectKeyIdentifier=hash
authorityKeyIdentifier=keyid,issuer:always

# This stuff is for subjectAltName and issuerAltname.
# Import the email address.
# subjectAltName=email:copy
# An alternative to produce certificates that aren't
# deprecated according to PKIX.
# subjectAltName=email:move

# Copy subject details
# issuerAltName=issuer:copy

#nsCaRevocationUrl      = http://www.domain.dom/ca-crl.
pem
#nsBaseUrl
#nsRevocationUrl
#nsRenewalUrl
#nsCaPolicyUrl
#nsSslServerName

# This really needs to be in place for it to be a proxy
certificate.
proxyCertInfo=critical,language:id-ppl-anyLanguage,
pathlen:3,policy:foo
```

Next steps

– For chipset feature management and to upgrade the chipset feature packs, see Customize security services.

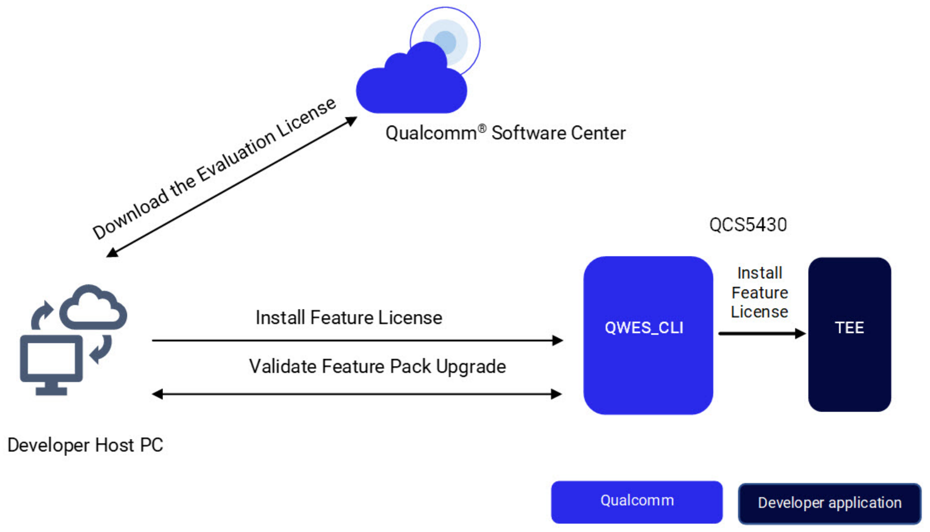
– For common logging and debugging techniques, see Debug Qualcomm TEE and secure devices.

Install or upgrade SoftSKU feature packs

You can upgrade the QCS5430 soft stock keeping unit (SKU) feature packs using the Qualcomm® wireless edge services (WES) license. For more information about the feature packs, see QCS5430: Feature packs.

Important: This feature is applicable to QCS5430.

The figure shows the process of upgrading the QCS5430 SoftSKU feature packs:



Feature packs	Download links to license files

Download evaluation license

Download the evaluation license according to the feature packs listed in the table:

Feature packs	Download links to license files
FP2	QCM5430 FP2.0
FP2.5	QCM5430 FP2.5
FP3	QCM5430 FP3.0

Prepare device

1. Set up the Wi-Fi on the device and get the `device-ip-address`.
2. Connect to the device using SSH:

   `ssh root@device-ip-address`

   For instructions, see Qualcomm Linux Build Guide
   $\rightarrow Howto \rightarrow SigninusingSSH$.
3. Remount the file system with read/write permissions:

   ```
   mount -o rw,remount /
   ```

4. Push the license to the device:

   ```
   scp /path/to/<QCM5430 FP*>.pfm root@device-ip-address:
   /data/
   ```

Install evaluation license

1. Ensure that the following prerequisites are met:
   - RPMB is provisioned.

     To verify if RPMB is provisioned, see Verify RPMB provision status.
   - The `qwes_cli` tool is available in the device.
2. Install the Qualcomm WES command-line interface tools:
   - Help: `qwes_cli -help`.
   - Default tools shipped with the Qualcomm Linux software product.
3. Install the evaluation license:

```
ssh root@device-ip-address
qwes_cli -f /data/<QCM5430 FP*>.pfm install
```

4. Verify license installation:

```
qwes_cli list
```

Command	Output

Check the serial number:
`qwes_cli list`
	`# qwes_cli list`
	`Running`
	`GetAllSerialNumber test`
	`with multithread Disabled`
	`Success. Len = 25`
	`Parsing 2 serial numbers`
	`Serial Number: 15b3`
	`Serial Number:`
	`3e52b512f2f47851a40fa5b30000018c86261aa1`

5. Ensure you restart the device for the license to take effect.

Verify feature pack upgrade

The two ways to verify the feature packs are:

– Verify the feature pack upgrade by using the following commands:

```
ssh root@device-ip-address
cat /proc/device-tree/model
```

The feature pack ID model is displayed in the output:

- `FP2: Qualcomm Technologies, Inc. qcs5430 fp2 addons rb3gen2 platforms`
- `FP2.5: Qualcomm Technologies, Inc. qcs5430 fp2p5 addons rb3gen2 platform`
- `FP3: Qualcomm Technologies, Inc. qcs5430 fp3 addons rb3gen2 platform`

Note: The feature pack and model ID mapping will be `{Feature Pack2: fp2 ; Feature pack2.5: fp2.5; Feature pack 3: fp3}`.

– Verify feature pack ID from the following UEFI logs:

FP1	`SoftSKU ID is disabled. The default setting is 1.`
FP2	`SoftSKU ID is enabled. updated here: 2`
FP2.5	`SoftSKU ID enabled. updated here: 8`
FP3	`SoftSKU ID enabled. updated here: 3`

Manage device licenses

You can perform the following operations to manage your device licenses:

– Upgrade the feature pack or apply a new license.
  - Multiple licenses can be installed.
  - License with the highest feature pack will be enforced by design.
  - You can upgrade or downgrade the feature pack license among the offered.
  - To downgrade, remove the license with the higher feature pack and apply the license for the required feature pack.

– Remove the evaluation license.
  - Get the license serial number:

```
qwes_cli list
Running GetAllSerialNumber test with multithread
Disabled
Success.  Len = 25
Parsing 2 serial numbers
Serial Number: 15b3
Serial Number:
3e52b512f2f47851a40fa5b30000018c86261aa1
```

  - Remove the license:

```
ssh root@device-ip-address
qwes_cli -s
3e52b512f2f47851a40fa5b30000018c86261aa1 remove
Serial number length 20
Remove License is successful.
```

  - Verify if the removed license serial number doesn't appear in the license list:

```
qwes_cli list
Running GetAllSerialNumber test with multithread
Disabled
Success.  Len = 4
Parsing 1 serial numbers
Serial Number: 15b3
```

○ **Ensure you restart the device for the removal of the license to** take effect.

Feature pack license - Persistence over software upgrades

– The license is installed in persistent secure storage, RPMB.

– When the license is installed successfully, its metadata is stored in RPMB, and a copy is maintained in the data partition.

– Even if user data is erased, the license remains preserved in RPMB, ensuring that the device functionality isn't impacted.

Note: This feature is available to licensed developers with authorized access to manage feature pack licenses. If you have access, see Qualcomm Linux Wireless Edge Services Guide.

Next steps

– To customize memory and SEPolicy, see Customize secuity services.

– To learn about Qualcomm WES, see Qualcomm WES.

– To develop applications that offer hardware-based attestation, zero-touch device provisioning, and chipset feature management, see: Qualcomm Linux Wireless Edge Services Guide. This feature is available to licensed developers with authorized access.

9 Customize secuity services

Customization is supported for memory and SEPolicy. For a large-size trusted application, you can customize the memory regions.

9.1 Customize memory

To customize memory, this feature is available to licensed users with authorized access. If you have access, see Qualcomm Linux Security Guide - Addendum → Customize memory for trusted application.

9.2 Customize SEPolicy

Qualcomm SEPolicy depends on the upstream SEPolicy. Therefore, the upstream SEPolicy's make system is used for building and customizing the SEPolicy.

Any customization to upstream code must be stored in the patches directory as a path to the upstream code. You can find the upstream code at https://github.com/SELinuxProject/refpolicy/.

The Qualcomm code is configured to the monolithic SEPolicy mode and SELinux types as targeted. To modify the SEPolicy mode and the SELinux types, do the following:

- To change the SELinux type and mode, you can edit the Qualcomm base file.
- To add the SEPolicy patches, add it to the patches folder and update the respective selinux_type bbappend files (refpolicy-targeted.bbappend or refpolicy-mls.bbappend), which aren't set by default. Add selinux enablement code in distro and then edit PREFERRED_PROVIDER_virtual/refpolicy.

Compile SEPolicy

1. To compile the SEPolicy, run the following commands:

   ```
   export SHELL=/bin/bash
   ```

2. Set up the build environment. For instructions, see Qualcomm Linux Build Guide $\rightarrow GitHubworkflowforunregisteredusers.$

3. Based on the SELinux type, compile only the SEPolicy with `bitbake refpolicy-mls` or

`bitbake refpolicy-targeted`.

```
bitbake <recipe_file_name>
```

Modify and build

You can also modify and build incrementally.

The audit2allow and research tools on Ubuntu 18 or 20 don't support policy version33. The policy version33 is supported from Ubuntu 23. If you aren't using Ubuntu 23, you can use a docker setup or a virtual machine to run audit2allow.

Install docker

The following command is used to install docker on Ubuntu with lower version.

```
sudo docker pull ubuntu:23.04
sudo docker run -ti --rm ubuntu /bin/bash
apt-get update
apt-get -y install policycoreutils
apt-get install -y policycoreutils-python-utils
```

Then run audit2allow on this shell .

Pull the policy version33 from the target `/etc/selinux/mls/policy/policy.33`. This policy is also available in the build tree. Use the mountbind or docker copy commands to share the policy with the docker.

Capture denials

If the command prompt doesn't change when `policy.33` is pulled from the `target /build`, and if `denials.txt` is a file capturing the denials, use the following command:

```
audit2allow -i denails.txt -p policy.33
```

Command not found

If this command isn't found, then use the following command to install the required package:

```
sudo apt install policycoreutils-python-utils
```

9.3 Next steps

- For common logging and debugging techniques, see Debug Qualcomm TEE and secure devices.
- To configure Qualcomm TEE for securing devices that handle sensitive data and run trusted applications, see Configure security services.

10 Debug Qualcomm TEE and secure devices

Debug provides a set of common logging and debugging techniques to troubleshoot issues in Qualcomm TEE, trusted and client applications, and secure devices.

Note: Run all the SSH commands in the SELinux Permissive mode. The Enforcing mode will be supported in the future. For instructions on how to connect to the device, see Qualcomm Linux Build Guide $\rightarrow Howto \rightarrow SigninusingSSH$.

10.1 Debug Qualcomm TEE

Qualcomm TEE kernel logs, also known as the TrustZone diag log, can be used to debug errors that occur in Qualcomm TEE.

The TrustZone diag log is available in the Linux kernel driver, which redirects the logs.

1. Connect to the device as the root using SSH.
2. Capture the TrustZone logs using the following command:

```
cat /proc/tzdbg/log > tzbsp_log.txt
```

The error codes in `tzbsp_log.txt` are encoded in hexadecimal. You can run the following tool to decode `tzbsp_log.txt` from hexadecimal to string.

1. Go to `<TZ.XF.X.X path>/trustzone_images/ssg/bsp/tz/build/tz/A53_64/<BuildFlavor>`
2. Run the following commands using python 3.

```
python3 print_tz_log.py -l tzbsp_log.txt -e errorCodesDict.txt -
t <TZ.XF.X.X path> -o tzbsp_log_decode.txt
```

   For example:

```
Python3 print_tz_log.py -l tzbsp_log.txt -e errorCodesDict.txt -
t //crmhyd/nsid-hyd-05/TZ.XF.5.0-07927-KODIAKAAAAANAAZT-1 -o
tzbsp_log_decode.txt
```

For device log collection, the TrustZone diag log buffer is part of the RAM dump, which can be parsed using `qsee.elf` from TZ.XF software in the crash dump parser tool. For offline or off-device log collection, the TrustZone diag log buffer is part of the RAM dump, which can be parsed using `qsee.elf` (trustzone_images/ssg/bsp/qsee/build/${tz_bid:EACAANAA}) from the TZ.XF software in the crash dump parser tool.

Debug using secure crash dump

You can debug Qualcomm TEE using the RAM dump. The execution region dump of Qualcomm TEE is collected using secure crash dumps.

Devices that trigger the fuse with stage 2 sec.elf are known as secure boot-enabled devices. To debug on these devices, see SecTools v2: Secure Debug User Guide.

Note: The *SecTools* guides are available to licensed developers with authorized access.

10.2 Debug trusted and client applications

The trusted application logs, also known as Qualcomm TEE logs, are used to debug the errors in trusted applications. To debug errors in the client application, the kernel and journalctl logs are used.

For online or on-device log collection, Linux collects the Qualcomm TEE/kernel logs at runtime. You can connect to the device using SSH and use the following commands:

- To collect the Qualcomm TEE logs from Linux:

```
cat /proc/tzdbg/qsee_log > qsee_log.txt
```

- For client applications, to collect the kernel and logcat logs:

```
cat /dev/kmsg > kernel_log.txt
journalctl > journalctl.txt
```

- For offline or off-device log collection, the Qualcomm TEE log is available in RAM dumps along with the kernel and journalctl logs.

10.3 Debug on secure devices

As part of the secure boot procedure, blowing debug disable fuses disable debugging capabilities on the devices. This includes RAM dumps, INV, and NINV debug on the subsystems.

The debug policy feature allows control over the debug capability for a device enabled with secure boot.

The debug policy image allows debug capabilities such as JTAG re-enable (INV debug), RAM dump, and TrustZone logging (NINV debug) on commercial secure devices.

For security reasons, the serial number of the device controls the debug policy for secure RAM dumps, Qualcomm TEE logs, and JTAG.

Enabling JTAG on the Qualcomm TEE subsystem disables the device security with respect to hardware key generation. As a result, existing secure storage like user data, SFS, and RPMB becomes inaccessible. Sometimes, the device may prompt for a factory data reset. Use the following command to debug on secure devices:

```
<meta>/common/sectoolsv2/ext/linux/sectools secure-debug --security-
profile <meta>/common/sectoolsv2/<chipset>_security_profile.xml --
generate --outfile apdp_out.mbn --all-flags --sign --signing-mode
LOCAL --oem-id=0x1 --root-certificate=./RSA-OEM-KEYS/qpsa_rootca.cer
--ca-certificate=./RSA-OEM-KEYS/qpsa_attestca.cer --ca-key=./RSA-OEM-
KEYS/qpsa_attestca.key --oem-product-id=0xabcd --serial-
number=0xabcdabcd
```

Ensure that you configure the OEM_ID, PRODUCT_ID, serial number and keys, and certification paths appropriately.

For more information, see SecTools v2: Secure Debug User Guide.

Note: The *SecTools* guides are available to licensed developers with authorized access.

10.4 Flash APDP on device

To flash APDP on the device, run the following command:

```
Fastboot flash apdp_a <path to apdp.mbn>
```

Table : Debug policy flags for dump collection

Stage	**Full dump**			**Mini dump**
Stage	Applications (DCC and scan dump) aDSP/Video/RPM/ SLPI	Modem/Qualcomm TEE/Secure dump	TZDiag	–
Non-secure	No debug policy needed	No debug policy needed	No debug policy needed	No debug policy needed
Stages 1 secure	No APDP image needed			No APDP image needed

Stage	Full dump			Mini dump
Stages 2 secure	`--nonsecure-crash-dumps`	–offline-crash dumps with device serial number	QCS6490/QCS5430: “–logs” with device serial number QCS9075: “–tz-diag-logs” with device serial number Or Encrypted TZDiag with `--nonsecure-crash-dumps` + TZDiag encryption public key/exp in devcfg can be configured in the following location: `/trustzone_images/ssg/securemsm/trustzone/qsee/mink/oem/config<chipset>/oem_config.xml`	• Apps minidump: `--apps-encrypted-mini-dumps` • Modem and WLAN: * `--mpss-encrypted-mini-dumps` * `--wlan-encrypted-mini-dumps` • aDSP minidump: `--adsp-encrypted-mini-dumps` • cDSP minidump: `--cdsp-encrypted-mini-dumps`

See KBA-191202045020-1 (ZIP). For more information, see MiniDump Software User Guide.

Note: The *SecTools* and *MiniDump* guides are available to licensed user with authorized access.

10.5 Qualcomm TEE/TrustZone diag log collection on secure device

On the secure device, the Qualcomm TEE/TrustZone log that's collected from Linux is disabled by default. Qualcomm provides an encrypted log feature for logging. Follow these steps for enabling this feature:

1. Generate an RSA key for encryption using:

```
openssl genrsa -out rsa_key 2048
```

2. Show RSA key information and modulus using:

```
openssl rsa -in rsa_key -text
openssl rsa -in rsa_key -modulus
Private-key: (2048 bit)
modulus: 00:a0:48:99:99:83:26:65:57:fc:75:52:25:45:53:
92:fc:27:29:cb:14:35:94:7c:89:bc:d4:0a:c6:3d:
0d:6d:8a:7d:72:1d:e3:4f:f0:32:66:41:a9:f6:c1:
2f:79:aa:58:ea:57:3b:29:6d:cf:40:33:4e:ad:ec:
bf:78:44:4b:28:52:c8:e3:6e:77:01:e5:a3:c6:25:
65:8c:8b:cc:32:20:2d:29:58:03:f0:d5:b7:f4:c0:
d6:09:b2:8e:59:c1:3c:ac:e5:61:04:36:78:e3:da:
95:b3:e3:b7:71:90:50:ee:a9:70:5a:15:1a:af:d9:
a5:4f:c2:70:f1:f8:f1:67:d1:78:0e:b8:95:6e:93:
73:6a:23:f1:31:e1:e2:49:ff:18:54:a3:73:d0:70:
91:de:7a:92:53:11:aa:cb:b0:f9:d0:e1:83:9f:74:
67:bc:1a:89:6d:b1:d2:de:4f:ab:3c:1c:63:c9:bc:
75:f0:c0:80:fc:db:73:d1:8a:e3:f4:60:57:dd:66:
f1:3a:fa:18:ed:7f:47:72:3e:49:50:94:8e:19:ae:
6b:69:62:3d:74:ca:44:fb:d4:1c:1d:59:43:30:31:
0d:fb:ab:70:44:9d:d9:d0:ce:cb:43:f3:2a:98:a4:
83:e7:76:ae:a8:b8:ea:63:64:e1:11:1b:99:92:b3: 9b:3f
publicExponent: 65537 (0x10001)
```

Note: The modulus is used in the `pub_mod in oem_config.xml` file. The `pub_exp` exponent is usually 65537. 0x10001 is known as the `publicExponent`.

3. Set the RSA public key (exponent and modulus) in the `trustzone_images/ssg/securemsm/trustzone/qsee/mink/oem/config/<chipset>/oem_config.xml` file.

   Enable this feature by adding the following lines to the `oem_config.xml` file using:

```
<driver name="NULL">
<global_def>
<var_seq name="pub_mod" type=DALPROP_DATA_TYPE_STRING>
a0489999832665557fc755225455392fc2729cb1435947c89bcd40ac63d0d6d
8a7d721de34ff0326641a9f6c12f79aa58ea573b296dcf40334eadecbf7844
4b2852c8e36e7701e5a3c625658c8bcc32202d295803f0d5b7f4c0d609b28e
59c13cace561043678e3da95b3e3b7719050eea9705a151aafd9a54fc270f1
f8f167d1780eb8956e93736a23f131e1e249ff1854a373d07091de7a925311
aacbb0f9d0e1839f7467bc1a896db1d2de4fab3c1c63c9bc75f0c080fcdb73
d18ae3f46057dd66f13afa18ed7f47723e4950948e19ae6b69623d74ca44fb
d41c1d594330310dfbab70449dd9d0cecb43f32a98a483e776aea8b8ea6364e1
111b9992b39b3f
</var_seq>
<var_seq name="pub_exp" type=DALPROP_DATA_TYPE_STRING>
000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000100
01
</var_seq>
</global_def>
```

Note: When the public key in the `oem_config.xml` file is updated, ensure that there are no new line characters, tabs, or spaces inserted between due to the Notepad or Wordpad editors.

4. Enable the encryption feature configuration flag from the `trustzone_images/ssg/securemsm/trustzone/qsee/mink/oem/config/<chipset>/oem_config.xml` file, using:

```
< props name="OEM_log_encr_enable" type=DALPROP_ATTR_TYPE_
UINT32>
1
</props>
```

5. To build the TrustZone devcfg image, enter the OEM_ID field value and sign the `devcfg.mbn` image.

6. Flash the signed `devcfg.mbn` image using:

```
fastboot flash devcfg_a devcfg.mbn
```

Note: Use `devcfg.mbn` for QCS6490 and `devcfg_iot.mbn` for QCS9100.

7. Collect the Qualcomm TEE/TrustZone log using:

```
cat /proc/tzdbg/qsee_log > qsee_log.txt
cat /proc/tzdbg/log > tz_log.txt
```

10.6 Qualcomm TEE/TrustZone diag log decryption steps

1. Download the Python decryption tool `decrypt_tzdiag_qsee_log_tools.py` from KBA-200917004544-1 (ZIP).
2. To install, run the following commands:

```
Python Version 3.x
pip install pycryptodome
pip install cryptography
```

3. To decrypt, run the following command:

```
python decrypt_tzdiag_qsee_log_tools.py -pk <RSA private key
file> -a RSA -I <input encrypted qsee/tz diag log collected from
device> -o <decrypted qsee/tzdiag log filename>
```

4. After successful decryption:
   a. Navigate the plain text of the Qualcomm TEE log to a readable string format.
   b. Convert the hexadecimal encoded error codes to string, using:

   ```
   print_tz_log.py
   ```

10.7 See also

- For sample code and examples, see Security services examples.
- To configure Qualcomm TEE for securing devices that handle sensitive data and run trusted applications, see Configure security services.

11 Security services examples

To run security services, sample code and examples to load client and trusted applications using different interfaces are available to licensed users with authorized access. If you have access, see Qualcomm Linux Security Guide - Addendum → Security services examples.

11.1 See also

- To learn about APIs that can be used to interact with Linux and hardware, see Security APIs.
- To learn how to develop and run trusted applications and client applications, see Develop trusted and client applications.

12 References (Security resouces and acronyms)

12.1 Related documents

Title	Document number
MiniDump Software User Guide	80-P8754-71
Qualcomm Linux Build Guide	80-70018-254
Qualcomm Linux Kernel Guide	80-70018-3
Qualcomm Linux Security Guide - Addendum	80-70018-11A
Qualcomm Linux Wireless Edge Services Guide	80-70018-11B
SecTools v2: Secure Debug User Guide	80-NM248-23
SSecTools V2: Metabuild Secure Image User Guide	80-NM248-17
SecTools V2: Fuse Blower User Guide	80-NM248-9
SecTools V2: ELF Tool User Guide	80-NM248-18
SecTools V2: MBN Tool User Guide	80-NM248-19
SecTools V2: ELF Consolidator User Guide	80-NM248-20
SecTools V2: Secure Image User Guide	80-NM248-12

Note: *MiniDump*, *Qualcomm Linux Security - Addendum*, *Qualcomm Linux Wireless Edge Services*, and *SecTools* guides are available to licensed user with authorized access.

12.2 Acronyms and terms

Acronym or term	Definition
DRM	Digital rights management
EL0, EL1, EL2, and EL3	Exception levels
eMMC	Embedded multimedia card

Acronym or term	**Definition**
GPCE	General purpose cryptographic engine
HAL	Hardware abstraction layer
HLOS	High-level operating system
HMAC	Hashed message authentication code
I2C	Inter integrated circuit
ICE	Inline crypto engine
IOCTL	I/O control
KEK	Key exchange keys
MAC	Message authentication code
MINK	Mini kernel
MPU	Memory protection unit
OCIMEM	On-chip internal memory
OEM	Original equipment manufacturer
PIL	Peripheral image loader
pIMEM	Protected memory
PRNG	Pseudo-random number generator
RMA	Returned material for analysis
QRNG	Qualcomm-random number generator
Qualcomm TEE	Qualcomm Trusted Execution Environment
Qualcomm WES	Qualcomm wireless edge services
RPMB	Replay protected memory block
SELinux	Security enhanced Linux
SEL0 and SEL1	Secure exception levels
SFS	Secure file system
SKU	Stock keeping unit
SMC	Secure monitor call
SPI	Serial peripheral interface
SSL	Secure sockets layer
TZBSP	TrustZone board support package
UEFI	Unified extensible firmware interface
UFS	Universal flash storage
XBL	eXtensible Boot Loader
xPU	External protection unit