Apple
Platform Security

February 2021

Contents

Introduction to Apple platform security

Apple designs security into the core of its platforms. Building on the experience of creating the world's most advanced mobile operating system, Apple has created security architectures that address the unique requirements of mobile, watch, desktop, and home.

Every Apple device combines *hardware*, *software*, and *services* designed to work together for maximum security and a transparent user experience in service of the ultimate goal of keeping personal information safe. For example, Apple-designed silicon and security hardware powers critical security features. And software protections work to keep the operating system and third-party apps protected. Finally, services provide a mechanism for secure and timely software updates, power a protected app ecosystem, and facilitate secure communications and payments. As a result, Apple devices protect not only the device and its data but the entire ecosystem, including everything users do locally, on networks, and with key internet services.

Just as we design our products to be simple, intuitive, and capable, we design them to be secure. Key security features, such as hardware-based device encryption, can't be disabled by mistake. Other features, such as Touch ID and Face ID, enhance the user experience by making it simpler and more intuitive to secure the device. And because many of these features are enabled by default, users or IT departments don't need to perform extensive configurations.

This documentation provides details about how security technology and features are implemented within Apple platforms. It also helps organizations combine Apple platform security technology and features with their own policies and procedures to meet their specific security needs.

The content is organized into the following topic areas:

- **Hardware security and biometrics:** The silicon and hardware that forms the foundation for security on Apple devices, including the Secure Enclave, a dedicated AES cryptographic engine, Touch ID, and Face ID
- **System security:** The integrated hardware and software functions that provide for the safe boot, update, and ongoing operation of Apple operating systems
- **Encryption and Data Protection:** The architecture and design that protects user data if the device is lost or stolen or if an unauthorized person or process attempts to use or modify it
- **App security:** The software and services that provide a safe app ecosystem and enable apps to run securely and without compromising platform integrity
- **Services security:** Apple's services for identification, password management, payments, communications, and finding lost devices

- **Network security:** Industry-standard networking protocols that provide secure authentication and encryption of data in transmission
- **Developer kit security:** Framework "kits" for secure and private management of home and health, as well as extension of Apple device and service capabilities to third-party apps
- **Secure device management:** Methods that allow management of Apple devices, prevent unauthorized use, and enable remote wipe if a device is lost or stolen

A commitment to security

Apple is committed to helping protect customers with leading privacy and security technologies—designed to safeguard personal information—and comprehensive methods, to help protect corporate data in an enterprise environment. Apple rewards researchers for the work they do to uncover vulnerabilities by offering the Apple Security Bounty. Details of the program and bounty categories are available at https://developer.apple.com/security-bounty/.

We maintain a dedicated security team to support all Apple products. The team provides security auditing and testing for products, both under development and released. The Apple team also provides security tools and training, and actively monitors for threats and reports of new security issues. Apple is a member of the Forum of Incident Response and Security Teams (FIRST).

Apple continues to push the boundaries of what's possible in security and privacy. This year Apple devices with Apple SoC's across the product lineup from Apple Watch to iPhone and iPad, and now Mac, utilize custom silicon to power not only efficient computation, but also security. Apple silicon forms the foundation for secure boot, Touch ID and Face ID, and Data Protection, as well as system integrity features never before featured on the Mac including Kernel Integrity Protection, Pointer Authentication Codes, and Fast Permission Restrictions. These integrity features help prevent common attack techniques that target memory, manipulate instructions, and use javascript on the web. They combine to help make sure that even if attacker code somehow executes, the damage it can do is dramatically reduced.

To make the most of the extensive security features built into our platforms, organizations are encouraged to review their IT and security policies to ensure that they are taking full advantage of the layers of security technology offered by these platforms.

To learn more about reporting issues to Apple and subscribing to security notifications, see Report a security or privacy vulnerability.

Apple believes privacy is a fundamental human right and has numerous built-in controls and options that allow users to decide how and when apps use their information, as well as what information is being used. To learn more about Apple's approach to privacy, privacy controls on Apple devices, and the Apple privacy policy, see https://www.apple.com/privacy.

Note: Unless otherwise noted, this documentation covers the following operating system versions: iOS 14.3, iPadOS 14.3, macOS 11.1, tvOS 14.3, and watchOS 7.2.

Hardware security and biometrics

Hardware security overview

For software to be secure, it must rest on hardware that has security built in. That's why Apple devices—running iOS, iPadOS, macOS, watchOS, or tvOS—have security capabilities designed into silicon. These capabilities include a CPU that powers system security features, as well as additional silicon that's dedicated to security functions. Security-focused hardware follows the principle of supporting limited and discretely defined functions in order to minimize attack surface. Such components include a boot ROM, which forms a hardware root of trust for secure boot, dedicated AES engines for efficient and secure encryption and decryption, and a Secure Enclave. The *Secure Enclave* is a system on chip (SoC) that is included on all recent iPhone, iPad, Apple Watch, Apple TV and HomePod devices, and on a Mac with Apple silicon as well as those with the Apple T2 Security Chip. The Secure Enclave itself follows the same principle of design as the SoC does, containing its own discrete boot ROM and AES engine. The Secure Enclave also provides the foundation for the secure generation and storage of the keys necessary for encrypting data at rest, and it protects and evaluates the biometric data for Touch ID and Face ID.

Storage encryption must be fast and efficient. At the same time, it can't expose the data (or *keying material*) it uses to establish cryptographic keying relationships. The AES hardware engine solves this problem by performing fast in-line encryption and decryption *as files are written or read*. A special channel from the Secure Enclave provides necessary keying material to the AES engine without exposing this information to the Application Processor (or CPU) or overall operating system. This ensures that the Apple Data Protection and FileVault technologies protect users' files without exposing long-lived encryption keys.

Apple has designed secure boot to protect the lowest levels of software against tampering and to allow only trusted operating system software from Apple to load at startup. Secure boot begins in immutable code called the Boot ROM, which is laid down during Apple SoC fabrication and is known as the *hardware root of trust*. On Mac computers with a T2 chip, trust for macOS secure boot begins with the T2. (Both the T2 chip and the Secure Enclave also execute their own secure boot processes using their own separate boot ROM—this is an exact analogue to how the A-series and M1 chips boot securely.)

The Secure Enclave also processes fingerprint and face data from Touch ID and Face ID sensors in Apple devices. This provides secure authentication while keeping user biometric data private and secure. It also enables users to benefit from the security of longer and more complex passcodes and passwords with, in many situations, the convenience of swift authentication for access or purchases.

Apple SoC security

Apple-designed silicon forms a common architecture across all Apple products and now powers Mac as well as iPhone, iPad, Apple TV, and Apple Watch. For over a decade, Apple's world-class silicon design team has been building and refining Apple systems on chip (SoCs). The result is a scalable architecture designed for all devices that leads the industry in security capabilities. This common foundation for security features is only possible from a company that designs its own silicon to work with its software.

Apple silicon has been designed and fabricated to specifically enable the system security features detailed below.

Feature	A10	A11, S3	A12, S4	A13, S5	A14, S6	M1
Kernel Integrity Protection	✓	✓	✓	✓	✓	✓
Fast Permission Restrictions		✓	✓	✓	✓	✓
System Coprocessor Integrity Protection			✓	✓	✓	✓
Pointer Authentication Codes			✓	✓	✓	✓
Page Protection Layer		✓	✓	✓	✓	See Note below.

Note: Page Protection Layer (PPL) requires that the platform execute *only* signed and trusted code; this is a security model that isn't applicable on macOS.

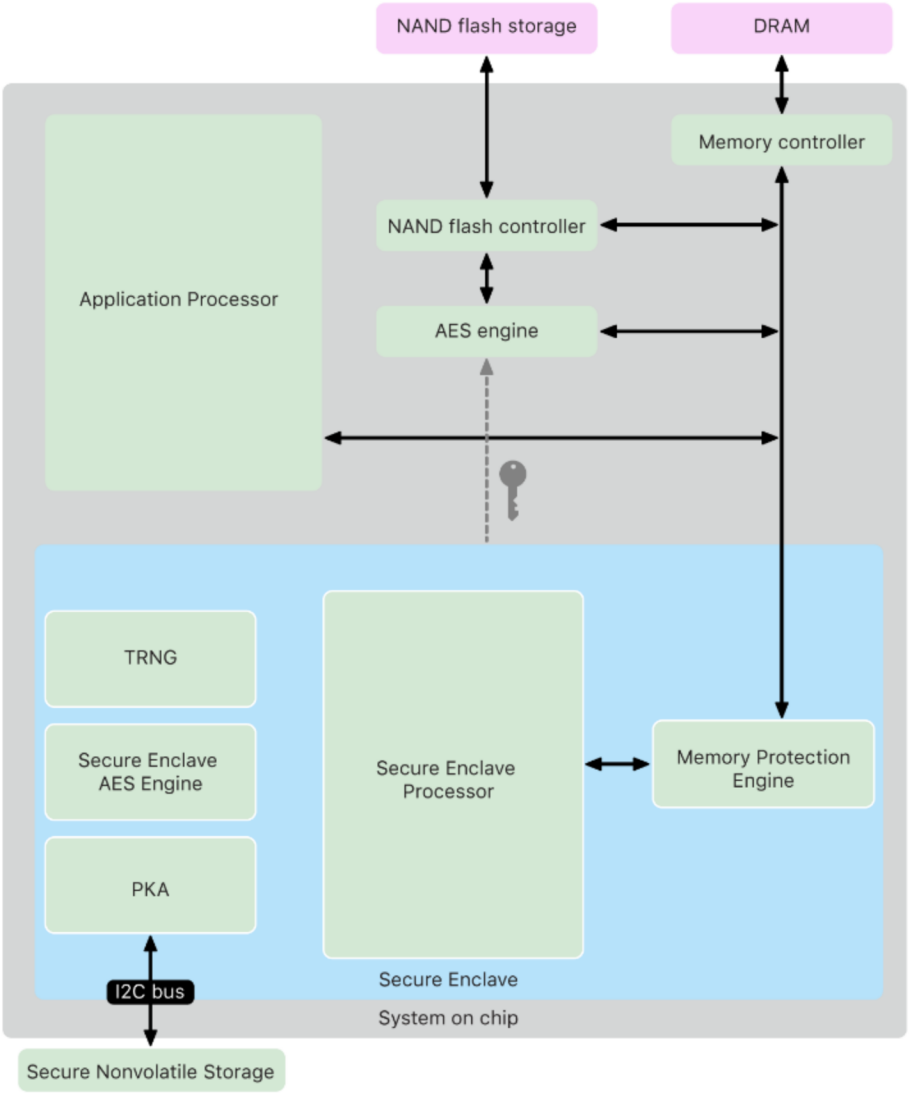
Apple-designed silicon also specifically enables the Data Protection capabilities detailed below.

Feature	A10	A11, S3	A12, S4	A13, S5	A14, M1, S6
Sealed Key Protection (SKP)	✓	✓	✓	✓	✓
recoveryOS - All Data Protection Classes protected	✓	✓	✓	✓	✓
Alternate boots of DFU, Diagnostics, and Update - Class A, B, and C data protected			✓	✓	✓

Secure Enclave

Overview

The Secure Enclave is a dedicated secure subsystem integrated into Apple systems on chip (SoCs). The Secure Enclave is isolated from the main processor to provide an extra layer of security and is designed to keep sensitive user data secure even when the Application Processor kernel becomes compromised. It follows the same design principles as the SoC does—a boot ROM to establish a hardware root of trust, an AES engine for efficient and secure cryptographic operations, and protected memory. Although the Secure Enclave doesn't include storage, it has a mechanism to store information securely on attached storage separate from the NAND flash storage that's used by the Application Processor and operating system.



The Secure Enclave components.

The Secure Enclave is a hardware feature of most versions of iPhone, iPad, Mac, Apple TV, Apple Watch, and HomePod—namely:

- iPhone 5s or later
- iPad Air or later
- MacBook Pro computers with Touch Bar (2016 and 2017) that contain the Apple T1 Chip
- Intel-based Mac computers that contain the Apple T2 Security Chip
- Mac computers with Apple silicon
- Apple TV HD or later
- Apple Watch Series 1 or later
- HomePod and HomePod mini

Secure Enclave Processor

The Secure Enclave Processor provides the main computing power for the Secure Enclave. To provide the strongest isolation, the Secure Enclave Processor is dedicated solely for Secure Enclave use. This helps prevent side-channel attacks that depend on malicious software sharing the same execution core as the target software under attack.

The Secure Enclave Processor runs an Apple-customized version of the L4 microkernel. It's designed to operate efficiently at a lower clock speed that helps to protect it against clock and power attacks. The Secure Enclave Processor, starting with the A11 and S4, includes a memory-protected engine and encrypted memory with anti-replay capabilities, secure boot, a dedicated random number generator, and its own AES engine.

Memory Protection Engine

The Secure Enclave operates from a dedicated region of the the device's DRAM memory. Multiple layers of protection isolate the Secure Enclave protected memory from the Application Processor.

When the device starts up, the Secure Enclave Boot ROM generates a random ephemeral memory protection key for the Memory Protection Engine. Whenever the Secure Enclave writes to its dedicated memory region, the Memory Protection Engine encrypts the block of memory using AES in Mac XEX (xor-encrypt-xor) mode, and calculates a Cipher-based Message Authentication Code (CMAC) authentication tag for the memory. The Memory Protection Engine stores the authentication tag alongside the encrypted memory. When the Secure Enclave reads the memory, the Memory Protection Engine verifies the authentication tag. If the authentication tag matches, the Memory Protection Engine decrypts the block of memory. If the tag doesn't match, the Memory Protection Engine signals an error to the Secure Enclave. After a memory authentication error, the Secure Enclave stops accepting requests until the system is rebooted.

Starting with the Apple A11 and S4 SoCs, the Memory Protection Engine adds replay protection for Secure Enclave memory. To prevent replay of security-critical data, the Memory Protection Engine stores a nonce for the block of memory alongside the authentication tag. The nonce is used as an additional tweak for the CMAC authentication tag. The nonces for all memory blocks are protected using an integrity tree rooted in dedicated SRAM within the Secure Enclave. For writes, the Memory Protection Engine *updates* the nonce and each level of the integrity tree up to the SRAM. For reads, the Memory Protection Engine *verifies* the nonce and each level of the integrity tree up to the SRAM. Nonce mismatches are handled similarly to authentication tag mismatches.

On Apple A14, M1, and later SoCS, the Memory Protection Engine supports two ephemeral memory protection keys. The first is used for data private to the Secure Enclave, and the second is used for data shared with the Secure Neural Engine.

The Memory Protection Engine operates inline and transparently to the Secure Enclave. The Secure Enclave reads and writes memory as if it were regular unencrypted DRAM, whereas an observer outside the Secure Enclave sees only the encrypted and authenticated version of the memory. The result is strong memory protection without performance or software complexity tradeoffs.

Secure Enclave Boot ROM

The Secure Enclave includes a dedicated Secure Enclave Boot ROM. Like the Application Processor Boot ROM, the Secure Enclave Boot ROM is immutable code that establishes the hardware root of trust for the Secure Enclave.

On system startup, iBoot assigns a dedicated region of memory to the Secure Enclave. Before using the memory, the Secure Enclave Boot ROM initializes the Memory Protection Engine to provide cryptographic protection of the Secure Enclave protected memory.

The Application Processor then sends the sepOS image to the Secure Enclave Boot ROM. After copying the sepOS image into the Secure Enclave protected memory, the Secure Enclave Boot ROM checks the cryptographic hash and signature of the image to verify that the sepOS is authorized to run on the device. If the sepOS image is properly signed to run on the device, the Secure Enclave Boot ROM transfers control to sepOS. If the signature isn't valid, the Secure Enclave Boot ROM prevents any further use of the Secure Enclave until the next chip reset.

On Apple A10 and later SoCs, the Secure Enclave Boot ROM locks a hash of the sepOS into a register dedicated to this purpose. The Public Key Accelerator uses this hash for operating-system-bound (OS-bound) keys.

Secure Enclave Boot Monitor

On Apple A13 and later SoCs, the Secure Enclave includes a Boot Monitor to ensure stronger integrity on the hash of the booted sepOS.

At system startup, the Secure Enclave Processor's System Coprocessor Integrity Protection (SCIP) configuration prevents the Secure Enclave Processor from executing any code other than the Secure Enclave Boot ROM. The Boot Monitor prevents the Secure Enclave from modifying the SCIP configuration directly. To make the loaded sepOS executable, the Secure Enclave Boot ROM sends the Boot Monitor a request with the address and size of the loaded sepOS. On receipt of the request, the Boot Monitor resets the Secure Enclave Processor, hashes the loaded sepOS, updates the SCIP settings to allow execution of the loaded sepOS, and starts execution within the newly loaded code. As the system continues booting, this same process is used whenever new code is made executable. Each time, the Boot Monitor updates a running hash of the boot process. The Boot Monitor also includes critical security parameters in the running hash.

When boot completes, the Boot Monitor finalizes the running hash and sends it to the Public Key Accelerator to use for OS-bound keys. This process is designed so that operating system key binding can't be bypassed even with a vulnerability in the Secure Enclave Boot ROM.

True Random Number Generator

The True Random Number Generator (TRNG) is used to generate secure random data. The Secure Enclave uses the TRNG whenever it generates a random cryptographic key, random key seed, or other entropy. The TRNG is based on multiple ring oscillators post processed with CTR_DRBG (an algorithm based on block ciphers in Counter Mode).

Root Cryptographic Keys

The Secure Enclave includes a unique ID (UID) root cryptographic key. The UID is unique to each individual device and isn't related to any other identifier on the device.

A randomly generated UID is fused into the SoC at manufacturing time. Starting with A9 SoCs, the UID is generated by the Secure Enclave TRNG during manufacturing and written to the fuses using a software process that runs entirely in the Secure Enclave. This process protects the UID from being visible outside the device during manufacturing and therefore isn't available for access or storage by Apple or any of its suppliers.

sepOS uses the UID to protect device-specific secrets. The UID allows data to be cryptographically tied to a particular device. For example, the key hierarchy protecting the file system includes the UID, so if the internal SSD storage is physically moved from one device to another, the files are inaccessible. Other protected device-specific secrets include Touch ID or Face ID data. On a Mac, only fully internal storage linked to the AES engine receives this level of encryption. For example, neither external storage devices connected over USB nor PCIe-based storage added to the 2019 Mac Pro are encrypted in this fashion.

The Secure Enclave also has a device group ID (GID), which is common to all devices that use a given SoC (for example, all devices using the Apple A14 SoC share the same GID).

The UID and GID aren't available through Joint Test Action Group (JTAG) or other debugging interfaces.

Secure Enclave AES Engine

The Secure Enclave AES Engine is a hardware block used to perform symmetric cryptography based on the AES cipher. The AES Engine is designed to resist leaking information by using timing and Static Power Analysis (SPA). Starting with the A9 SoC, the AES Engine also includes Dynamic Power Analysis (DPA) countermeasures.

The AES Engine supports hardware and software keys. Hardware keys are derived from the Secure Enclave UID or GID. These keys stay within the AES Engine and aren't made visible even to sepOS software. Although software can request encryption and decryption operations with hardware keys, it can't extract the keys.

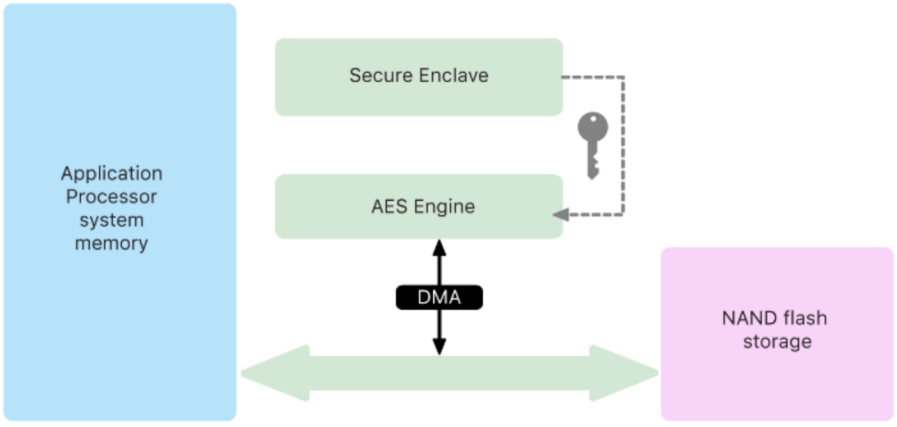
On Apple A10 and newer SoCs, the AES Engine includes lockable seed bits that diversify keys derived from the UID or GID. This allows data access to be conditioned on the device's mode of operation. For example, lockable seed bits are used to deny access to password-protected data when booting from Device Firmware Update (DFU) mode. For more information, see Passcodes and passwords.

AES Engine

Every Apple device with a Secure Enclave also has a dedicated AES256 crypto engine (the "AES Engine") built into the direct memory access (DMA) path between the NAND (nonvolatile) flash storage and main system memory, making file encryption highly efficient. On A9 or later A-series processors, the flash storage subsystem is on an isolated bus that's granted access only to memory containing user data through the DMA crypto engine.

At boot time, sepOS generates an ephemeral wrapping key using the TRNG. The Secure Enclave transmits this key to the AES Engine using dedicated wires to prevent it from being accessed by any software outside the Secure Enclave. sepOS can then use the ephemeral wrapping key to wrap file keys for use by the Application Processor file-system driver. When the file-system driver reads or writes a file, it sends the wrapped key to the AES Engine, which unwraps the key. The AES Engine never exposes the unwrapped key to software.

Note: The AES Engine is a separate component from both the Secure Enclave and the Secure Enclave AES Engine, but its operation is closely tied to the Secure Enclave, as shown below.



The AES Engine supports line-speed encryption on the DMA path for efficient encryption and decryption of data is it is written and read to storage.

Public Key Accelerator

The Public Key Accelerator (PKA) is a hardware block used to perform asymmetric cryptography operations. The PKA supports RSA and ECC (Elliptic Curve Cryptography) signing and encryption algorithms. The PKA is designed to resist leaking information using timing and side-channel attacks such as SPA and DPA.

The PKA supports software and hardware keys. Hardware keys are derived from the Secure Enclave UID or GID. These keys stay within the PKA and aren't made visible even to sepOS software.

Starting with A13 SoCs, the PKA's encryption implementations have been proved to be mathematically correct using formal verification techniques.

On Apple A10 and later SoCs, the PKA supports OS-bound keys, also referred to as Sealed Key Protection (SKP). These keys are generated using a combination of the device's UID and the hash of the sepOS running on the device. The hash is provided by the Secure Enclave Boot ROM, or by the Secure Enclave Boot Monitor on Apple A13 and later SoCs. These keys are also used to verify the sepOS version when making requests to certain Apple services and are also used to improve the security of passcode-protected data by helping to prevent access to keying material if critical changes are made to the system without user authorization.

Secure nonvolatile storage

The Secure Enclave is equipped with a dedicated secure nonvolatile storage device. The secure nonvolatile storage is connected to the Secure Enclave using a dedicated I2C bus, so that it can only be accessed by the Secure Enclave. All user data encryption keys are rooted in entropy stored in the Secure Enclave nonvolatile storage.

In devices with A12, S4, and later SoCs, the Secure Enclave is paired with a Secure Storage Component for entropy storage. The Secure Storage Component is itself designed with immutable ROM code, a hardware random number generator, a per-device unique cryptographic key, cryptography engines, and physical tamper detection. The Secure Enclave and Secure Storage Component communicate using an encrypted and authenticated protocol that provides exclusive access to the entropy.

Devices first released in Fall 2020 or later are equipped with a 2nd-generation Secure Storage Component. The 2nd-generation Secure Storage Component adds counter lockboxes. Each counter lockbox stores a 128-bit salt, a 128-bit passcode verifier, an 8-bit counter, and an 8-bit maximum attempt value. Access to the counter lockboxes is through an encrypted and authenticated protocol.

Counter lockboxes hold the entropy needed to unlock passcode-protected user data. To access the user data, the paired Secure Enclave must derive the correct passcode entropy value from the user's passcode and the Secure Enclave's UID. The user's passcode can't be learned using unlock attempts sent from a source other than the paired Secure Enclave. If the passcode attempt limit is exceeded (for example, 10 attempts on iPhone), the passcode-protected data is erased completely by the Secure Storage Component.

To create a counter lockbox, the Secure Enclave sends the Secure Storage Component the passcode entropy value and the maximum attempt value. The Secure Storage Component generates the salt value using its random number generator. It then derives a passcode verifier value and a lockbox entropy value from the provided passcode entropy, the Secure Storage Component's unique cryptographic key, and the salt value. The Secure Storage Component initializes the counter lockbox with a count of 0, the provided maximum attempt value, the derived passcode verifier value, and the salt value. The Secure Storage Component then returns the generated lockbox entropy value to the Secure Enclave.

To retrieve the lockbox entropy value from a counter lockbox later, the Secure Enclave sends the Secure Storage Component the passcode entropy. The Secure Storage Component first increments the counter for the lockbox. If the incremented counter exceeds the maximum attempt value, the Secure Storage Component completely erases the counter lockbox. If the maximum attempt count hasn't been reached, the Secure Storage Component attempts to derive the passcode verifier value and lockbox entropy value with the same algorithm used to create the counter lockbox. If the derived passcode verifier value matches the stored passcode verifier value, the Secure Storage Component returns the lockbox entropy value to the Secure Enclave and resets the counter to 0.

The keys used to access password-protected data are rooted in the entropy stored in counter lockboxes. For more information, see Data Protection overview.

The secure nonvolatile storage is used for all anti-replay services in the Secure Enclave. Anti-replay services on the Secure Enclave are used for revocation of data over events that mark anti-replay boundaries including, but not limited to, the following:

- Passcode change
- Enabling or disabling Touch ID or Face ID
- Adding or removing a Touch ID fingerprint or Face ID face
- Touch ID or Face ID reset
- Adding or removing an Apple Pay card
- Erase All Content and Settings

On architectures that don't feature a Secure Storage Component, EEPROM (electrically erasable programmable read-only memory) is utilized to provide secure storage services for the Secure Enclave. Just like the Secure Storage Components, the EEPROM is attached and accessible only from the Secure Enclave, but it doesn't contain dedicated hardware security features nor does it guarantee exclusive access to entropy (aside from its physical attachment characteristics) nor counter lockbox functionality.

Secure Neural Engine

On devices with Face ID, the Secure Neural Engine converts 2D images and depth maps into a mathematical representation of a user's face.

On A11 through A13 SoCs, the Secure Neural Engine is integrated into the Secure Enclave. The Secure Neural Engine uses direct memory access (DMA) for high performance. An input-output memory management unit (IOMMU) under the sepOS kernel's control limits this direct access to authorized memory regions.

Starting with A14 and the M1, the Secure Neural Engine is implemented as a secure mode in the Application Processor's Neural Engine. A dedicated hardware security controller switches between Application Processor and Secure Enclave tasks, resetting Neural Engine state on each transition to keep Face ID data secure. A dedicated engine applies memory encryption, authentication, and access control. At the same time, it uses a separate cryptographic key and memory range to limit the Secure Neural Engine to authorized memory regions.

Power and clock monitors

All electronics are designed to operate within a limited voltage and frequency envelope. When operated outside this envelope, the electronics can malfunction and then security controls may be bypassed. To ensure that the voltage and frequency stay in a safe range, the Secure Enclave is designed with monitoring circuits. These monitoring circuits are designed to have a much larger operating envelope than the rest of the Secure Enclave. If the monitors detect an illegal operating point, the clocks in the Secure Enclave automatically stop and don't restart until the next SoC reset.

Secure Enclave feature summary

Note: A12, A13, S4, and S5 products first released in Fall 2020 have a 2nd-generation Secure Storage Component; while earlier products based on these SoCs have 1st-generation Secure Storage Component.

SoC	Memory Protection Engine	Secure Storage	AES Engine	PKA
A8	Encryption and authentication	EEPROM	Yes	No
A9	Encryption and authentication	EEPROM	DPA protection	Yes
A10	Encryption, authentication, and replay prevention	EEPROM	DPA protection and lockable seed bits	OS-bound keys
A11	Encryption, authentication, and replay prevention	EEPROM	DPA protection and lockable seed bits	OS-bound keys
A12 (Apple devices released before Fall 2020)	Encryption, authentication, and replay prevention	Secure Storage Component gen 1	DPA protection and lockable seed bits	OS-bound keys

SoC	Memory Protection Engine	Secure Storage	AES Engine	PKA
A12 (Apple devices released after Fall 2020)	Encryption, authentication, and replay prevention	Secure Storage Component gen 2	DPA protection and lockable seed bits	OS-bound keys
A13 (Apple devices released before Fall 2020)	Encryption, authentication, and replay prevention	Secure Storage Component gen 1	DPA protection and lockable seed bits	OS-bound keys and Boot Monitor
A13 (Apple devices released after Fall 2020)	Encryption, authentication, and replay prevention	Secure Storage Component gen 2	DPA protection and lockable seed bits	OS-bound keys and Boot Monitor
A14	Encryption, authentication, and replay prevention	Secure Storage Component gen 2	DPA protection and lockable seed bits	OS-bound keys and Boot Monitor
S3	Encryption and authentication	EEPROM	DPA protection and lockable seed bits	Yes
S4	Encryption, authentication, and replay prevention	Secure Storage Component gen 1	DPA protection and lockable seed bits	OS-bound keys
S5 (Apple devices released before Fall 2020)	Encryption, authentication, and replay prevention	Secure Storage Component gen 1	DPA protection and lockable seed bits	OS-bound keys
S5 (Apple devices released after Fall 2020)	Encryption, authentication, and replay prevention	Secure Storage Component gen 2	DPA protection and lockable seed bits	OS-bound keys
S6	Encryption, authentication, and replay prevention	Secure Storage Component gen 2	DPA protection and lockable seed bits	OS-bound keys
T2	Encryption and authentication	EEPROM	DPA protection and lockable seed bits	OS-bound keys
M1	Encryption, authentication, and replay prevention	Secure Storage Component gen 2	DPA protection and lockable seed bits	OS-bound keys and Boot Monitor

Touch ID and Face ID

Touch ID and Face ID security

Passcodes and passwords are essential to the security of Apple devices. At the same time, users need to be able to quickly access their devices, even exceeding a hundred times a day. Biometric authentication provides a way to retain the security of a strong passcode—or even strengthen the passcode or password because it won't often need to be entered manually—while providing the convenience of swiftly unlocking with a finger press or glance. Touch ID and Face ID don't replace a passcode or password, but in most situations they do make access faster and easier.

Touch ID security

Touch ID is the fingerprint sensing system that makes secure access to supported Apple devices faster and easier. This technology reads fingerprint data from any angle and learns more about a user's fingerprint over time, with the sensor continuing to expand the fingerprint map as additional overlapping nodes are identified with each use.

Apple devices with a Touch ID sensor can be unlocked using a fingerprint. Touch ID doesn't replace the need for a device passcode or user password, which is still required after device startup, restart, or logout (on a Mac). In some apps, Touch ID can also be used in place of device passcode or user password—for example, to unlock password protected notes in the Notes app, to unlock keychain-protected websites, and to unlock supported app passwords. However, a device passcode or user password is always required in some scenarios (for example, to change an existing device passcode or user password or to remove existing fingerprint enrollments or create new ones).

When the fingerprint sensor detects the touch of a finger, it triggers the advanced imaging array to scan the finger and sends the scan to the Secure Enclave. Communication between the processor and the Touch ID sensor takes place over a serial peripheral interface bus. The processor forwards the data to the Secure Enclave but can't read it. It's encrypted and authenticated with a session key that's negotiated using a shared key provisioned for each Touch ID sensor and its corresponding Secure Enclave at the factory. For every Touch ID sensor the shared key is strong, random, and different. The session key exchange uses AES key wrapping, with both sides providing a random key that establishes the session key and uses transport encryption that provides both authentication and confidentiality (using AES-CCM).

While the fingerprint scan is being vectorized for analysis, the raster scan is temporarily stored in encrypted memory within the Secure Enclave and then it's discarded. The analysis utilizes subdermal ridge flow angle mapping, a lossy process that discards "finger minutiae data" that would be required to reconstruct the user's actual fingerprint. The resulting map of nodes is stored without any identity information in an encrypted format that can be read only by the Secure Enclave. This data never leaves the device. It's not sent to Apple, nor is it included in device backups.

Face ID security

With a simple glance, Face ID securely unlocks supported Apple devices. It provides intuitive and secure authentication enabled by the TrueDepth camera system, which uses advanced technologies to accurately map the geometry of a user's face. Face ID uses neural networks for determining attention, matching, and antispoofing, so a user can unlock their phone with a glance. Face ID automatically adapts to changes in appearance, and carefully safeguards the privacy and security of a user's biometric data.

Face ID is designed to confirm user attention, provide robust authentication with a low false-match rate, and mitigate both digital and physical spoofing.

The TrueDepth camera automatically looks for the user's face when the user wakes an Apple device that features Face ID (by raising it or tapping the screen), as well as when those devices attempt to authenticate the user in order to display an incoming notification or when a supported app requests Face ID authentication. When a face is detected, Face ID confirms attention and intent to unlock by detecting that the user's eyes are open and their attention is directed at their device; for accessibility, the Face ID attention check is disabled when VoiceOver is activated and, if required, can be disabled separately.

After the TrueDepth camera confirms the presence of an attentive face, it projects and reads over 30,000 infrared dots to form a depth map of the face along with a 2D infrared image. This data is used to create a sequence of 2D images and depth maps, which are digitally signed and sent to the Secure Enclave. To counter both digital and physical spoofs, the TrueDepth camera randomizes the sequence of 2D images and depth map captures, and projects a device-specific random pattern. A portion of the Secure Neural Engine—protected within the Secure Enclave—transforms this data into a mathematical representation and compares that representation to the enrolled facial data. This enrolled facial data is itself a mathematical representation of the user's face captured across a variety of poses.

Touch ID, Face ID, passcodes, and passwords

To use Touch ID or Face ID, the user must set up their device so that a passcode or password is required to unlock it. When Touch ID or Face ID detects a successful match, the user's device unlocks without asking for the device passcode or password. This makes using a longer, more complex passcode or password far more practical because the user doesn't need to enter it as frequently. Touch ID and Face ID don't replace the user's passcode or password; instead, they provide easy access to the device within thoughtful boundaries and time constraints. This is important because a strong passcode or password forms the foundation for how a user's iPhone, iPad, Mac, or Apple Watch cryptographically protects that user's data.

When a device passcode or password is required

Users can use their passcode or password anytime instead of Touch ID or Face ID, but there are situations where biometrics aren't permitted. The following security-sensitive operations always require entry of a passcode or password:

- Updating the software
- Erasing the device
- Viewing or changing passcode settings
- Installing configuration profiles
- Unlocking the Security & Privacy pane in System Preferences on Mac
- Unlocking the Users & Groups pane in System Preferences on Mac (if FileVault is turned on)

A passcode or password is also required if the device is in the following states:

- The device has just been turned on or restarted.
- The user has logged out of their Mac account (or hasn't yet logged in).
- The user hasn't unlocked their device for more than 48 hours.
- The user hasn't used their passcode or password to unlock their device for 156 hours (six and a half days), and the user hasn't used a biometric to unlock their device in 4 hours.
- The device has received a remote lock command.
- The user exited power off/Emergency SOS by pressing and holding either volume button and the Sleep/Wake button simultaneously for 2 seconds and then pressing Cancel.

- There were five unsuccessful biometric match attempts (though for usability, the device might offer entering a passcode or password instead of using biometrics after a smaller number of failures).

When Touch ID or Face ID is enabled on an iPhone or iPad, the device immediately locks when the Sleep/Wake button is pressed, and the device locks every time it goes to sleep. Touch ID and Face ID require a successful match—or optionally the passcode—at every wake.

The probability that a random person in the population could unlock a user's iPhone, iPad, or Mac is 1 in 50,000 with Touch ID or 1 in 1,000,000 with Face ID. This probability increases with multiple enrolled fingerprints (up to 1 in 10,000 with five fingerprints) or appearances (up to 1 in 500,000 with two appearances). For additional protection, both Touch ID and Face ID allow only five unsuccessful match attempts before a passcode or password is required to obtain access to the user's device or account. With Face ID, the probability of a false match is different for twins and siblings who look like the user and for children under the age of 13 (because their distinct facial features may not have fully developed). If a user is concerned about a false match, Apple recommends using a passcode to authenticate.

Facial matching security

Facial matching is performed within the Secure Enclave using neural networks trained specifically for that purpose. Apple developed the facial matching neural networks using over a billion images, including infrared (IR) and depth images collected in studies conducted with the participants' informed consent. Apple then worked with participants from around the world to include a representative group of people accounting for gender, age, ethnicity, and other factors. The studies were augmented as needed to provide a high degree of accuracy for a diverse range of users. Face ID is designed to work with hats, scarves, eyeglasses, contact lenses, and many types of sunglasses. Furthermore, it's designed to work indoors, outdoors, and even in total darkness. An additional neural network—that's trained to spot and resist spoofing—defends against attempts to unlock the device with photos or masks. Face ID data, including mathematical representations of a user's face, is encrypted and available only to the Secure Enclave. This data never leaves the device. It's not sent to Apple, nor is it included in device backups. The following Face ID data is saved, encrypted only for use by the Secure Enclave, during normal operation:

- The mathematical representations of a user's face calculated during enrollment
- The mathematical representations of a user's face calculated during some unlock attempts if Face ID deems them useful to augment future matching

Face images captured during normal operation aren't saved but are instead immediately discarded after the mathematical representation is calculated for either enrollment or comparison to the enrolled Face ID data.

Improving Face ID matches

To improve match performance and keep pace with the natural changes of a face and look, Face ID augments its stored mathematical representation over time. Upon a successful match, Face ID may use the newly calculated mathematical representation—if its quality is sufficient—for a finite number of additional matches before that data is discarded. Conversely, if Face ID fails to recognize a face but the match quality is higher than a certain threshold and a user immediately follows the failure by entering their passcode, Face ID takes another capture and augments its enrolled Face ID data with the newly calculated mathematical representation. This new Face ID data is discarded if the user stops matching against it or after a finite number of matches. These augmentation processes allow Face ID to keep up with dramatic changes in a user's facial hair or makeup use while minimizing false acceptance.

Uses for Touch ID and Face ID

Unlocking a device or user account

With Touch ID or Face ID disabled, when a device or account locks, the keys for the highest class of Data Protection—which are held in the Secure Enclave—are discarded. The files and keychain items in that class are inaccessible until the user unlocks the device or account by entering their passcode or password.

With Touch ID or Face ID enabled, the keys aren't discarded when the device or account locks; instead, they're wrapped with a key that's given to the Touch ID or Face ID subsystem inside the Secure Enclave. When a user attempts to unlock the device or account, if the device detects a successful match, it provides the key for unwrapping the Data Protection keys, and the device or account is unlocked. This process provides additional protection by requiring cooperation between the Data Protection and Touch ID or Face ID subsystems to unlock the device.

When the device restarts, the keys required for Touch ID or Face ID to unlock the device or account are lost; they're discarded by the Secure Enclave after any condition is met that requires passcode or password entry.

Securing purchases with Apple Pay

The user can also use Touch ID and Face ID with Apple Pay to make easy and secure purchases in stores, apps, and on the web:

- *Using Touch ID:* For Touch ID, the intent to pay is confirmed using the gesture of activating the Touch ID sensor combined with successfully matching the user's fingerprint.
- *Using Face ID in stores:* To authorize an in-store payment with Face ID, the user must first confirm intent to pay by double-clicking the side button. This double-click captures user intent using a physical gesture directly linked to the Secure Enclave and is resistant to forgery by a malicious process. The user then authenticates using Face ID before placing the device near the contactless payment reader. A different Apple Pay payment method can be selected after Face ID authentication, which requires reauthentication, but the user won't have to double-click the side button again.

- *Using Face ID in apps and on the web:* To make a payment within apps and on the web, the user confirms their intent to pay by double-clicking the side button and then authenticates using Face ID to authorize the payment. If the Apple Pay transaction isn't completed within 60 seconds of double-clicking the side button, the user must reconfirm intent to pay by double-clicking again.

Using system-provided APIs

Third-party apps can use system-provided APIs to ask the user to authenticate using Touch ID or Face ID or a passcode or password, and apps that support Touch ID automatically support Face ID without any changes. When using Touch ID or Face ID, the app is notified only as to whether the authentication was successful; it can't access Touch ID, Face ID, or the data associated with the enrolled user.

Protecting keychain items

Keychain items can also be protected with Touch ID or Face ID, to be released by the Secure Enclave only by a successful match or with the device passcode or account password. App developers have APIs to verify that a passcode or password has been set by the user before requiring Touch ID or Face ID or a passcode or password to unlock keychain items. App developers can do any of the following:

- Require that authentication API operations don't fall back to an app password or the device passcode. They can query whether a user is enrolled, allowing Touch ID or Face ID to be used as a second factor in security-sensitive apps.
- Generate and use Elliptic Curve Cryptography (ECC) keys inside the Secure Enclave that can be protected by Touch ID or Face ID. Operations with these keys are always performed inside the Secure Enclave after it authorizes their use.

Making and approving purchases

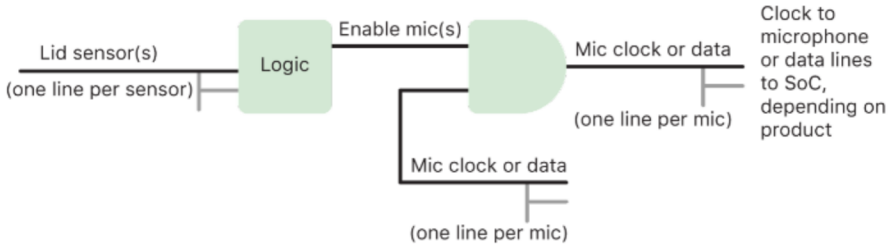
Users can also configure Touch ID or Face ID to approve purchases from the iTunes Store, the App Store, Apple Books, and more, so users don't have to enter their Apple ID password. When purchases are made, the Secure Enclave verifies that a biometric authorization occurred and then releases ECC keys used to sign the store request.

Hardware microphone disconnect

All Apple silicon-based Mac notebooks and Intel-based Mac notebooks with the Apple T2 Security Chip feature a hardware disconnect that disables the microphone whenever the lid is closed. On all 13-inch MacBook Pro and MacBook Air notebooks with the T2 chip, all MacBook notebooks with a T2 chip from 2019 or later, and Mac notebooks with Apple silicon, this disconnect is implemented in hardware alone. The disconnect prevents any software—even with root or kernel privileges in macOS, and even the software on the T2 chip or other firmware—from engaging the microphone when the lid is closed. (The camera isn't disconnected in hardware, because its field of view is completely obstructed with the lid closed.)

iPad models beginning in 2020 also feature the hardware microphone disconnect. When an MFi-compliant case (including those sold by Apple) is attached to the iPad and closed, the microphone is disconnected in hardware, preventing microphone audio data being made available to any software—even with root or kernel privileges in iPadOS, or any device firmware.

The protections in this section are implemented directly with hardware logic, according to the following circuit diagram:



Circuit diagram.

In each product with a hardware microphone cutoff, one or more lid sensors detect the physical closure of the lid or case using some physical property (for example, a Hall effect sensor or a hinge angle sensor) of the interaction. For sensors where calibration is necessary, parameters are set during production of the device and the calibration process includes a nonreversible hardware lock out of any subsequent changes to sensitive parameters on the sensor. These sensors emit a direct hardware signal that goes through a simple set of nonreprogrammable hardware logic. This logic provides debounce, hysteresis, and/or a delay of up to 500 ms before disabling the microphone. Depending on the product, this signal can be implemented either by disabling the lines transporting data between the microphone and the System on Chip (SoC) or by disabling one of the input lines to the microphone module that's allowing it to be active—for example, the clock line or a similar effective control.

Express Cards with power reserve

If iOS isn't running because iPhone needs to be charged, there may still be enough power in the battery to support Express Card transactions. Supported iPhone devices automatically support this feature with:

- A payment or transit card designated as the Express Transit card
- Student ID cards with Express Mode turned on

Pressing the side button (or on iPhone SE 2nd generation, the Home button) displays the low-battery icon as well as text indicating that Express Cards are available to use. The NFC controller performs Express Card transactions under the same conditions as when iOS is running, except that transactions are indicated only with haptic notification (no visible notification is shown). On iPhone SE 2nd generation, completed transactions may take a few seconds to appear on screen. This feature isn't available when a standard user-initiated shutdown is performed.

System security

System security overview

Building on the unique capabilities of Apple hardware, system security is responsible for controlling access to system resources in Apple devices without compromising usability. System security encompasses the boot-up process, software updates, and protection of computer system resources such as CPU, memory, disk, software programs, and stored data.

The most recent versions of Apple operating systems are the most secure. An important part of Apple security is *secure boot*, which protects the system from malware infection at boot time. Secure boot begins in hardware and builds a chain of trust through software, where each step ensures that the next is functioning properly before handing over control. This security model supports not only the default boot of Apple devices but also the various modes for recovery and timely updates on Apple devices. Subcomponents like the T2 Chip and the Secure Enclave also perform their own secure boot to help ensure they only boot known-good code from Apple. The update system can even prevent downgrade attacks, so that devices can't be rolled back to an older version of the operating system (which an attacker knows how to compromise) as a method of stealing user data.

Apple devices also include boot and runtime protections so that they maintain their integrity during ongoing operation. Apple-designed silicon on iPhone, iPad, Apple Watch, Apple TV, HomePod, and a Mac with Apple silicon provide a common architecture for protecting operating system integrity. macOS also features an expanded and configurable set of protection capabilities in support of its differing computing model, as well as capabilities supported on all Mac hardware platforms.

Secure boot

Boot process for iOS and iPadOS devices

Each step of the startup process contains components that are cryptographically signed by Apple to enable integrity checking so that boot proceeds only after verifying the chain of trust. These components include the bootloaders, the kernel, kernel extensions, and cellular baseband firmware. This secure boot chain is designed to verify that the lowest levels of software aren't tampered with.

When an iOS or iPadOS device is turned on, its Application Processor immediately executes code from read-only memory referred to as Boot ROM. This immutable code, known as the *hardware root of trust*, is laid down during chip fabrication and is implicitly trusted. The Boot ROM code contains the Apple Root certificate authority (CA) public key—used to verify that the iBoot bootloader is signed by Apple before allowing it to load. This is the first step in the chain of trust, in which each step checks that the next is signed by Apple. When the iBoot finishes its tasks, it verifies and runs the iOS or iPadOS kernel. For devices with an A9 or earlier A-series processor, an additional Low-Level Bootloader (LLB) stage is loaded and verified by the Boot ROM and in turn loads and verifies iBoot.

A failure to load or verify following stages is handled differently depending on the hardware:

- *Boot ROM can't load LLB (older devices):* Device Firmware Upgrade (DFU) mode
- *LLB or iBoot:* Recovery mode

In either case, the device must be connected to iTunes (in macOS 10.14 or earlier) or the Finder (macOS 10.15 or later) through USB and restored to factory default settings.

The Boot Progress Register (BPR) is used by the Secure Enclave to limit access to user data in different modes and is updated before entering the following modes:

- *DFU mode:* Set by Boot ROM on devices with an Apple A12 or later SoCs
- *Recovery mode:* Set by iBoot on devices with Apple A10, S2, or later SoCs

On devices with cellular access, a cellular baseband subsystem performs additional secure booting using signed software and keys verified by the baseband processor.

The Secure Enclave also performs a secure boot that checks its software (sepOS) is verified and signed by Apple.

Memory safe iBoot implementation

In iOS 14 and iPadOS 14, Apple modified the C compiler toolchain used to build the iBoot bootloader to improve its security. The modified toolchain implements code to prevent memory- and type-safety issues that are typically encountered in C programs. For example, it prevents:

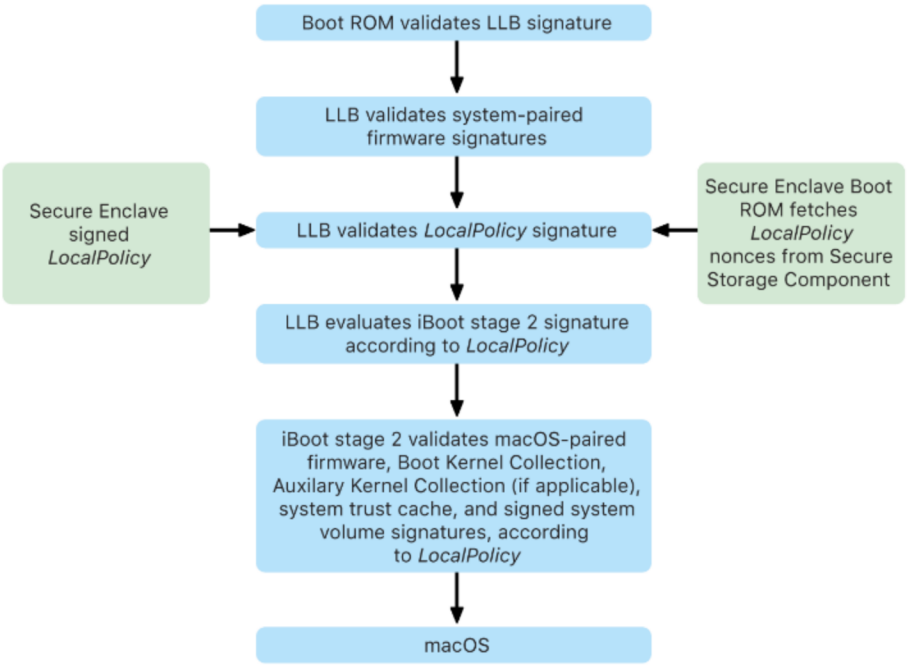
- Buffer overflows, by ensuring that all pointers carry bounds information that is verified when accessing memory
- Heap exploitation, by separating heap data from its metadata and accurately detecting error conditions such as double free errors
- Type confusion, by ensuring that all pointers carry runtime type information that's verified during pointer cast operations
- Type confusion caused by use after free errors, by segregating all dynamic memory allocations by static type

This technology is available on iPhone with Apple A13 Bionic or later, and iPad with the A14 Bionic chip.

Mac computers with Apple silicon

Boot process for a Mac with Apple silicon

When a Mac with Apple silicon is turned on, it performs a boot process much like that of iPhone and iPad.



Boot process steps when a Mac with Apple silicon is started.

The chip executes code from the Boot ROM in the first step in the chain of trust. macOS secure boot on a Mac with Apple silicon verifies not only the operating system code itself, but also the security policies and even kexts (supported, though not recommended) configured by authorized users.

When LLB is launched, it then verifies the signatures and loads system-paired firmware for intra-SoC cores such as the storage, display, system management, and Thunderbolt controllers. LLB is also responsible for loading the LocalPolicy, which is a file signed by the Secure Enclave Processor. The LocalPolicy file describes the configuration that the user has chosen for the system boot and runtime security policies. The LocalPolicy has the same data structure format as all other boot objects, but it's signed locally by a private key that's available only within a particular machine's Secure Enclave instead of being signed by a central Apple server (like software updates).

To prevent replay of any previous LocalPolicy, LLB must look up a nonce from the Secure Enclave-attached Secure Storage Component. To do this, it uses the Secure Enclave Boot ROM and makes sure the nonce in the LocalPolicy matches the nonce in the Secure Storage Component. This prevents an old LocalPolicy—which could have been configured for lower security—from being reapplied to the system after security has been upgraded. The result is that secure boot on a Mac with Apple silicon helps protect not only against rollback of operating system versions but also against security policy downgrades.

The LocalPolicy file captures whether the operating system is configured for Full, Reduced, or Permissive security.

- *Full Security:* The system behaves like iOS and iPadOS, and allows only booting software which was known to be the latest that was available at install time.
- *Reduced Security:* LLB is directed to trust "global" signatures, which are bundled with the operating system. This allows the system to run older versions of macOS. Because older versions of macOS inevitably have unpatched vulnerabilities, this security mode is described as *Reduced*. This is also the policy level required to support booting kernel extensions (kexts).
- *Permissive Security:* The system behaves like Reduced Security in that it uses global signature verification for iBoot and beyond, but it also tells iBoot that it should accept some boot objects being signed by the Secure Enclave with the same key used to sign the LocalPolicy. This policy level supports users that are building, signing, and booting their own custom XNU kernels.

If the LocalPolicy indicates to LLB that the selected operating system is running in Full Security, LLB evaluates the personalized signature for iBoot. If it's running in Reduced Security or Permissive Security, it evaluates the global signature. Any signature verification errors cause the system to boot to recoveryOS to provide repair options.

After LLB hands off to iBoot, it loads macOS-paired firmware such as that for the Secure Neural Engine, the Always On Processor, and other firmware. iBoot also looks at information about the LocalPolicy handed to it from LLB. If the LocalPolicy indicates that there should be an Auxiliary Kernel Collection (AuxKC), iBoot looks for it on the file system, verifies that it was signed by the Secure Enclave with the same key as the LocalPolicy, and verifies that its hash matches a hash stored in the LocalPolicy. If the AuxKC is verified, iBoot places it into memory with the Boot Kernel Collection before locking the full memory region covering the Boot Kernel Collection and AuxKC with the System Coprocessor Integrity Protection (SCIP). If the policy indicates that an AuxKC should be present but it isn't found, the system continues to boot into macOS without it. iBoot is also responsible for verifying the root hash for the signed system volume (SSV), to check that the file system the kernel will mount is fully integrity verified.

Boot modes for a Mac with Apple silicon

A Mac with Apple silicon has the boot modes described below.

Mode	Key combo	Description
macOS	From a shutdown state, press and **release** the power button.	1. Boot ROM hands off to LLB. 2. LLB loads system-paired firmware and the Local Policy for the selected macOS. 3. LLB locks an indication into the Boot Progress Register (BPR) that it's booting into normal macOS, and hands off to iBoot. 4. iBoot loads the macOS-paired firmware, the static trust cache, the device tree, and the Boot Kernel Collection. 5. If the LocalPolicy allows it, iBoot loads the Auxiliary Kernel Collection (AuxKC) of third-party kexts. 6. If the LocalPolicy didn't disable it, iBoot verifies the root signature hash for the signed system volume (SSV).
recoveryOS	From a shutdown state, press and **hold** the power button.	1. Boot ROM hands off to LLB. 2. LLB loads system-paired firmware and the Local Policy for the recoveryOS. 3. LLB locks an indication into the Boot Progress Register that it's booting into recoveryOS, and hands off to iBoot for recoveryOS. 4. iBoot loads the macOS-paired firmware, the trust cache, the device tree, and the Boot Kernel Collection. *Note:* Security downgrades aren't allowed on the recoveryOS LocalPolicy.
Fallback recovery OS	From a shutdown state, **double-press and hold** the power button.	The same process as recoveryOS boot, except that it boots to a second copy of recoveryOS that is kept for resiliency. However, LLB doesn't lock an indication into the Boot Progress Register saying it is going into recoveryOS, and therefore the fallback recovery OS doesn't have the capability to change the system security state.
Safe mode	Boot into recoveryOS per the above, then hold **Shift** while selecting the startup volume.	1. Boots to recoveryOS as per the above. 2. Holding the Shift key while selecting a volume causes the BootPicker application to approve that macOS for booting, as normal, but to also set an `nvram` variable that tells iBoot to not load the AuxKC on the next boot. 3. System reboots and boots to the targeted volume, but iBoot doesn't load AuxKC.

Startup Disk security policy control for a Mac with Apple silicon

Overview

Unlike security policies on an Intel-based Mac, security policies on a Mac with Apple silicon are for each installed operating system. This means that multiple installed macOS instances with different versions and security policies are supported on the same machine. For this reason, an operating system picker has been added to Startup Security Utility.

macOS storage selection to change the security policy.

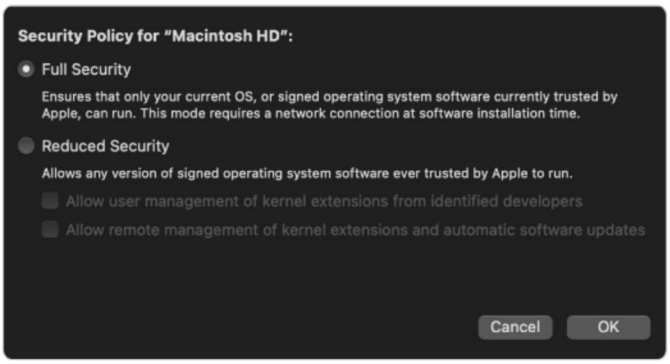
On a Mac with Apple silicon, System Security Utility indicates the overall user-configured security state of macOS, such as the booting of a kext or the configuration of System Integrity Protection (SIP). If changing a security setting would significantly degrade security or make the system easier to compromise, users must enter into recoveryOS by holding the power button (so that malware can't trigger the signal, only a human with physical access can), in order to make the change. Because of this, an Apple-silicon based Mac also won't require (or support) a firmware password—all critical changes are already gated by user authorization. For more information on SIP, see System Integrity Protection.

On a Mac with Apple silicon, System Security Utility indicates the overall user-configured security state of macOS. And in many cases, if changing a security setting would significantly degrade security or make the system easier to compromise, it's a prerequisite that a user enter into recoveryOS by holding the power button (so that malware can't trigger the signal, only a human with physical access can), in order to lower the operating system security.

Full Security and Reduced Security can be set using Startup Security Utility from recoveryOS. But Permissive Security can be accessed only from command-line tools for users who accept the risk of making their Mac much less secure.

Full Security policy

Full Security is the default, and it behaves like iOS and iPadOS. At the time software is downloaded and prepared to install, rather than using the global signature that comes with the software, macOS contacts the same Apple signing server used for iOS and iPadOS and requests a fresh, "personalized" signature. A signature is personalized when it includes the Exclusive Chip Identification (ECID)—a unique ID specific to the Apple CPU in this case—as part of the signing request. The signature given back by the signing server is then unique and usable only by that particular Apple CPU. When the Full Security policy is in effect, the Boot ROM and LLB ensures that a given signature isn't just signed by Apple but is signed for this specific Mac, essentially tying that version of macOS to that Mac.



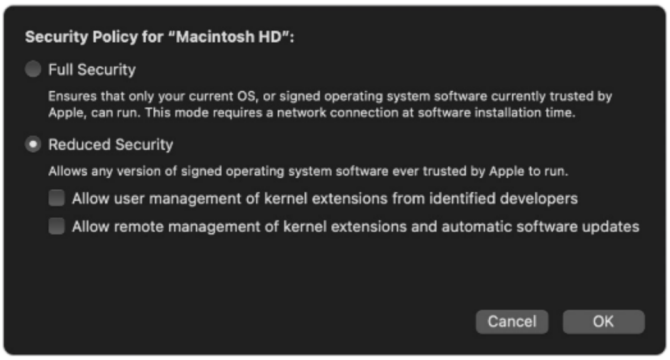
macOS Full Security policy selection.

Using an online signing server also provides better protection against rollback attacks than typical global signature approaches. In a global signing system, the security epoch could have rolled many times, but a system that has never seen the latest firmware won't know this. For example, a computer that currently believes it's in security epoch 1 accepts software from security epoch 2, even if the current actual security epoch is 5. With an Apple silicon online signing system, the signing server can reject creating signatures for software that's in anything except the latest security epoch.

Additionally, if an attacker discovers a vulnerability after a security epoch change, they can't simply pick up the vulnerable software from a previous epoch off system A and apply it to system B in order to attack it. The fact that the vulnerable software from an older epoch was personalized to system A prevents it from being transferable and thus being used to attack a system B. All these mechanisms work together to provide much stronger guarantees that attackers can't purposely place vulnerable software on a Mac in order to circumvent the protections provided by the latest software. But a user that's in possession of an administrator user name and password for the Mac can always choose the security policy that works best for their use cases.

Reduced Security policy

Reduced Security is similar to Medium Security behavior on an Intel-based Mac with a T2 chip, in which a vendor (in this case, Apple) generates a digital signature for the code to assert it came from the vendor. In this way, attackers are prevented from inserting unsigned code. Apple refers to this signature as a "global" signature because it can be used on any Mac, for any amount of time, for a Mac that currently has a Reduced Security policy set. Reduced security doesn't itself provide protection against rollback attacks (although unauthorized operating system changes can result in user data being rendered inaccessible. For more information, see Kernel extensions in a Mac with Apple silicon.



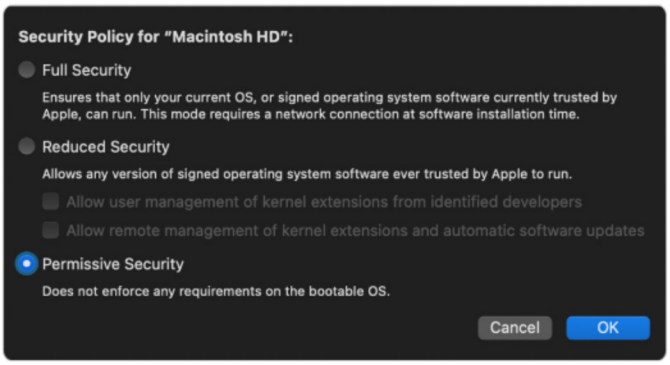
macOS Reduced Security policy selection.

In addition to enabling users to run older versions of macOS, Reduced Security is required for other actions that can put a user's system security at risk, such as introducing third-party kernel extensions (kexts). Kexts have the same privileges as the kernel, and thus any vulnerabilities in third-party kexts can lead to full operating system compromise. This is why developers are being strongly encouraged to adopt system extensions before kext support is removed from macOS for future Mac computers with Apple silicon. Even when third-party kexts are enabled, they can't be loaded into the kernel on demand. Instead, the kexts are merged into an Auxiliary Kernel Collection (AuxKC)—whose hash is stored in the LocalPolicy—and thus they require a reboot. For more information about AuxKC generation, see Kernel extensions in macOS.

Permissive Security policy

Permissive Security is for users who accept the risk of putting their Mac into a much more insecure state. This mode differs from No Security mode on an Intel-based Mac with a T2 chip. With Permissive Security, signature verification is still performed along the entire secure boot chain, but setting the policy to Permissive signals to iBoot that it should accept locally Secure Enclave–signed boot objects, such as a user-generated Boot Kernel Collection built from a custom XNU kernel. In this way, Permissive Security also provides an architectural capability for running an arbitrary "fully untrusted operating system" kernel. When a custom Boot Kernel Collection or fully untrusted operating system is loaded on the system, some decryption keys become unavailable, to prevent a fully untrusted operating systems from accessing data from trusted operating systems.

Important: Apple doesn't provide or support custom XNU kernels.



macOS Permissive Security policy selection.

There's another way that Permissive Security differs from No Security on an Intel-based Mac with a T2 chip: It's a prerequisite for some security downgrades that in the past have been independently controllable. Most notably, to disable System Integrity Protection (SIP) on a Mac with Apple silicon, a user must acknowledge that they're putting the system into Permissive Security. This is required because disabling SIP has always put the system into a state that makes the kernel much easier to compromise. In particular, disabling SIP on a Mac with Apple silicon disables kext signature enforcement during AuxKC generation time, thus allowing any arbitrary kext to be loaded into kernel memory. Another improvement to SIP that's been made on a Mac with Apple silicon is that the policy store has been moved out of NVRAM and into the LocalPolicy. So now, disabling SIP requires authentication by a user who has access to the LocalPolicy signing key from recoveryOS (reached by pressing and holding the power button). This makes it significantly more difficult for a software-only attacker, or even a physically present attacker, to disable SIP.

It isn't possible to downgrade to Permissive Security from the Startup Security Utility app. Users can downgrade only by running command-line tools from Terminal in recoveryOS, such as `csrutil` (to disable SIP). After the user has downgraded, the fact that it's occurred is reflected in Startup Security Utility, and so a user can easily set the security to a more secure mode.

Note: A Mac with Apple silicon doesn't require or support a specific media boot policy, because technically all boots are performed locally. If a user chooses to boot from external media, that operating system version must first be personalized using an authenticated reboot from recoveryOS. This reboot creates a LocalPolicy file on the internal drive that's used to perform a trusted boot from the operating system stored on the external media. This means the configuration of booting from external media is always explicitly enabled on a per operating system basis, and already requires user authorization, so no additional secure configuration is necessary.

LocalPolicy signing-key creation and management

Creation

When macOS is first installed in the factory, or when a tethered erase-install is performed, the Mac runs code from temporary restore RAM disk to initialize the default state. During this process, the restore environment creates a new pair of public and private keys which are held in the Secure Enclave. The private key is referred to as the *Owner Identity Key (OIK)*. If any OIK already exists, it's destroyed as part of this process. The restore environment also initializes the key used for Activation Lock; the *User Identity Key (UIK)*. Part of that process which is unique to a Mac with Apple silicon is when UIK certification is requested for Activation Lock, a set of requested constraints to be enforced at validation-time on the LocalPolicy are included. If the device can't get a UIK certified for Activation Lock (for example, because the device is currently associated with a Find My Mac account and reported as lost), its unable to proceed further to create a Local Policy. If a device is issued a *User identity Certificate (ucrt)*, that ucrt contains server imposed policy constraints and user requested policy constraints in an X.509 v3 extension.

When an Activation Lock/ucrt is successfully retrieved, it's stored in a database on the server side and also returned to the device. After the device has a ucrt, a certification request for the public key which corresponds to the OIK is sent to the *Basic Attestation Authority (BAA)* server. BAA verifies the OIK certification request using the public key from the ucrt stored in the BAA accessible database. If the BAA can verify the certification, it certifies the public key, returning the *Owner Identity Certificate (OIC)* which is signed by the BAA and contains the constraints stored in ucrt. The OIC is sent back to the Secure Enclave. From then on, whenever the Secure Enclave signs a new LocalPolicy, it attaches the OIC to the Image4. LLB has built-in trust in the BAA root certificate, which causes it to trust the OIC, which causes it to trust the overall LocalPolicy signature.

RemotePolicy constraints

All Image4 files, not just Local Policies, contain constraints on Image4 manifest evaluation. These constraints are encoded using special object identifiers (OIDs) in the leaf certificate. The Image4 verification library looks up the special certificate constraint OID from a certificate during signature evaluation and then mechanically evaluates the constraints specified in it. The constraints are of the form:

- X must exist
- X must not exist
- X must have a specific value

So, for instance, for "personalized" signatures, the certificate constraints will contain "ECID must exist," and for "global" signatures, it will contain "ECID must not exist." These constraints ensure that all Image4 files signed by a given key must conform to certain requirements to avoid erroneous signed Image4 manifest generation.

In the context of each LocalPolicy, these Image4 certificate constraints are referred to as the *RemotePolicy*. A different RemotePolicy can exist for different boot environments' LocalPolicies. The RemotePolicy is used to restrict the recoveryOS LocalPolicy so that when recoveryOS is booted it can only ever behave as if it's booting with Full Security. This increases trust in the integrity of the recoveryOS boot environment as a place where policy can be changed. The RemotePolicy restricts the LocalPolicy to contain the ECID of the machine on which the LocalPolicy was generated, and the specific Remote Policy Nonce Hash (`rpnh`) stored in the Secure Storage Component on that Mac. The `rpnh`, and therefore the RemotePolicy, change only when actions are taken for Find My Mac and Activation Lock, such as enrollment, unenrollment, remote lock, and remote wipe. Remote Policy constraints are determined and specified at User Identity Key (UIK) certification time and are signed in to the issued User identity Certificate (ucrt). Some Remote Policy constraints are determined by the server such as ECID, ChipID and BoardID to prevent one device from signing LocalPolicy files for another device. Other Remote Policy constraints may be specified by the device to prevent Security downgrade of the Local Policy without providing both the local authentication required to access the current OIK and remote authentication of the account to which the device is Activation Locked.

Contents of a LocalPolicy file for a Mac with Apple silicon

The LocalPolicy is an Image4 file signed by the Secure Enclave. Image4 is an ASN.1 (Abstract Syntax Notation One) DER-encoded data structure format that's used to describe information about secure boot chain objects on Apple platforms. In an Image4-based secure boot model, security policies are requested at software installation time initiated by a signing request to a central Apple signing server. If the policy was acceptable, the signing server returns a signed Image4 file containing a variety of four-character code (4CC) sequences. These signed Image4 files and 4CCs are evaluated at startup by software like the Boot ROM or LLB.

Ownership handoff between operating systems

Access to the Owner Identity Key (OIK) is referred to as "Ownership." Ownership is required to allow users to resign the LocalPolicy after making policy or software changes. The OIK is protected with the same key hierarchy as described in Sealed Key Protection (SKP), with the OIK being protected by the same Key encryption key (KEK) as the Volume encryption key (VEK). This means it's normally protected by both user passwords and measurements of the operating system and policy. There's only a single OIK for all operating systems on the Mac. Therefore, when installing a second operating system, explicit consent is required from the users on the first operating system to hand off Ownership to the users on the second operating system. However, users don't yet exist for the second operating system, when the installer is running from the first operating system. Users in operating systems aren't normally generated until the operating system is booted and the Setup Assistant is running. Thus two new actions are required when installing a second operating system on a Mac with Apple silicon:

- Creating a LocalPolicy for the second operating system
- Preparing an "Install User" for handing off Ownership

When running an Install Assistant and targeting installation for a secondary blank volume, a prompt asks the user if they'd like to copy a user from the current volume to be the first user of the second volume. If the user says yes, the "Install User" which is created is, in reality, a KEK which is derived from the selected user's password and hardware keys, which is then used to encrypt the OIK as it's being handed to the second operating system. Then from within the second operating system Install Assistant, it prompts for that user's password, to allow it to access the OIK in the Secure Enclave for the new operating system. If users opt not to copy a user, the Install User is still created the same way, but an empty password is used instead of a user's password. This second flow exists for certain system administration scenarios. However, users who want to have multi-volume installs and want to perform Ownership handoff in the most secure fashion should always opt to copy a user from the first operating system to the second operating system.

LocalPolicy on a Mac with Apple silicon

For a Mac with Apple silicon, local security policy control has been delegated to an application running in the Secure Enclave. This software can utilize the user's credentials and the boot mode of the primary CPU to determine who can change the security policy and from what boot environment. This helps prevent malicious software from using the security policy controls against the user by downgrading them to gain more privileges.

LocalPolicy manifest properties

The LocalPolicy file contains some architectural 4CCs that are found in most all Image4 files—such as a board or model ID (BORD), indicating a particular Apple chip (CHIP), or Exclusive Chip Identification (ECID). But the 4CCs below focus only on the security policies that users can configure.

Note: Apple uses the term *One True recoveryOS (1TR)* to indicate a boot into the primary recoveryOS which is achieved using a physical power button press. This is different from a normal recoveryOS boot, which can be achieved using NVRAM or which may happen when errors occur on startup. The physical button press increases trust that the boot environment isn't reachable by a software-only attacker who has broken into macOS.

LocalPolicy Nonce Hash (lpnh)

- *Type:* OctetString (48)
- *Mutable environments:* 1TR, recoveryOS, macOS
- *Description:* The `lpnh` is used for anti-replay of the LocalPolicy. This is an SHA384 hash of the LocalPolicy Nonce (LPN), which is stored in the Secure Storage Component and accessible using the Secure Enclave Boot ROM or Secure Enclave. The raw nonce is never visible to the Application Processor, only to the sepOS. An attacker wanting to convince LLB that a previous LocalPolicy they had captured was valid would need to place a value into the Secure Storage Component, which hashes to the same `lpnh` value found in the LocalPolicy they want to replay. Normally there is a single LPN valid on the system—except during software updates, when two are simultaneously valid—to allow for the possibility of falling back to booting the old software in the event of an update error. When any LocalPolicy for any operating system is changed, all policies are re-signed with the new *lpnh* value corresponding to the new LPN found in the Secure Storage Component. This change happens when the user changes security settings or creates new operating systems with a new LocalPolicy for each.

Remote Policy Nonce Hash (rpnh)

- *Type:* OctetString (48)
- *Mutable environments:* 1TR, recoveryOS, macOS
- *Description:* The `rpnh` behaves the same way as the `lpnh` but is updated only when the remote policy is updated, such as when changing the state of Find My enrollment. This change happens when the user changes the state of Find My on their Mac.

recoveryOS Nonce Hash (ronh)

- *Type:* OctetString (48)
- *Mutable environments:* 1TR, recoveryOS, macOS
- *Description:* The `ronh` behaves the same way as the `lpnh`, but is found exclusively in the LocalPolicy for recoveryOS. It's updated when the recoveryOS is updated, such as on software updates. A separate nonce from the `lpnh` and `rpnh` is used so that when a device is put into a disabled state by Find My, existing operating systems can be disabled (by removing their LPN and RPN from the Secure Storage Component), while still leaving the recoveryOS bootable. In this way, the operating systems can be reenabled when the system owner proves their control over the system by putting in their iCloud password used for the Find My account. This change happens when a user updates the recoveryOS or creates new operating systems.

Next Stage Image4 Manifest Hash (nsih)

- *Type:* OctetString (48)
- *Mutable environments:* 1TR, recoveryOS, macOS
- *Description:* The *nsih* field represents an SHA384 hash of the Image4 manifest data structure that describes the booted macOS. The macOS Image4 manifest contains measurements for all the boot objects—such as iBoot, the static trust cache, device tree, Boot Kernel Collection, and signed system volume (SSV) volume root hash. When LLB is directed to boot a given macOS, it ensures that the hash of the macOS Image4 manifest attached to iBoot matches what's captured in the `nsih` field of the LocalPolicy. In this way, the `nsih` captures the user intention of what operating system the user has created a LocalPolicy for. Users change the `nsih` value implicitly when they perform a software update.

Auxiliary Kernel Collection (AuxKC) Policy Hash (auxp)

- *Type:* OctetString (48)
- *Mutable environments:* macOS
- *Description:* The `auxp` is an SHA384 hash of the user-authorized kext list (UAKL) policy. This is used at AuxKC generation time to ensure that only user-authorized kexts are included in the AuxKC. `smb2` is a prerequisite for setting this field. Users change the `auxp` value implicitly when they change the UAKL by approving a kext from the Security & Privacy pane in System Preferences.

Auxiliary Kernel Collection (AuxKC) Image4 Manifest Hash (auxi)

- *Type:* OctetString (48)
- *Mutable environments:* macOS
- *Description:* After the system verifies that the UAKL hash matches what's found in the auxp field of the LocalPolicy, it requests that the AuxKC be signed by the Secure Enclave processor application that's responsible for LocalPolicy signing. Next, an SHA384 hash of the AuxKC Image4 manifest signature is placed into the LocalPolicy to avoid the potential for mixing and matching previously signed AuxKCs to an operating system at boot time. If iBoot finds the `auxi` field in the LocalPolicy, it attempts to load the AuxKC from storage and validate its signature. It also verifies that the hash of the Image4 manifest attached to the AuxKC matches the value found in the `auxi` field. If the AuxKC fails to load for any reason, the system continues to boot without this boot object and (so) without any third-party kexts loaded. The `auxp` field is a prerequisite for setting the auxi field in the LocalPolicy. Users change the `auxi` value implicitly when they change the UAKL by approving a kext from the Security & Privacy pane in System Preferences.

Auxiliary Kernel Collection (AuxKC) Receipt Hash (auxr)

- *Type:* OctetString (48)
- *Mutable environments:* macOS
- *Description:* The `auxr` is an SHA384 hash of the AuxKC receipt, which indicates the exact set of kexts that were included into the AuxKC. The AuxKC receipt can be a subset of the UAKL, because kexts can be excluded from the AuxKC even if they're user authorized if they're known to be used for attacks. In addition, some kexts that can be used to break the user-kernel boundary may lead to decreased functionality, such as an inability to use Apple Pay or play 4K and HDR content. Users who want these capabilities opt in to a more restrictive AuxKC inclusion. The `auxp` field is a prerequisite for setting the `auxr` field in the LocalPolicy. Users change the `auxr` value implicitly when they build a new AuxKC from the Security & Privacy pane in System Preferences.

APFS volume group UUID (vuid)

- *Type:* OctetString (16)
- *Mutable environments:* 1TR, recoveryOS, macOS
- *Description:* The `vuid` indicates the volume group the kernel should use as root. This field is primarily informational and isn't used for security constraints. This `vuid` is set by the user implicitly when creating a new operating system install.

Key encryption key (KEK) Group UUID (kuid)

- *Type:* OctetString (16)
- *Mutable environments:* 1TR, recoveryOS, macOS
- *Description:* The `kuid` indicates the volume that was booted. The key encryption key has typically been used for Data Protection. For each LocalPolicy, it's used to protect the LocalPolicy signing key. The `kuid` is set by the user implicitly when creating a new operating system install.

Paired recoveryOS Trusted Boot Policy Measurement (prot)

- *Type:* OctetString (48)
- *Mutable environments:* 1TR, recoveryOS, macOS
- *Description:* A paired recoveryOS Trusted Boot Policy Measurement (TBPM) is a special iterative SHA384 hash calculation over the Image4 manifest of a LocalPolicy, excluding nonces, in order to give a consistent measurement over time (because nonces like `lpnh` are frequently updated). The `prot` field, which is found only in each macOS LocalPolicy, provides a pairing to indicate the recoveryOS LocalPolicy that corresponds to the macOS LocalPolicy.

Has Secure Enclave Signed recoveryOS Local Policy (hrlp)

- *Type:* Boolean
- *Mutable environments:* 1TR, recoveryOS, macOS
- *Description:* The `hrlp` indicates whether or not the `prot` value (above) is the measurement of a Secure Enclave–signed recoveryOS LocalPolicy. If not, then the recoveryOS LocalPolicy is signed by the Apple online signing server, which signs things such as macOS Image4 files.

Secure Multi-Boot (smb0)

- *Type:* Boolean
- *Mutable environments:* 1TR, recoveryOS
- *Description:* If `smb0` is present and true, LLB allows the next stage Image4 manifest to be globally signed instead of requiring a personalized signature. Users can change this field with Startup Security Utility or `bputil` to downgrade to Reduced Security.

Secure Multi-Boot (smb1)

- *Type:* Boolean
- *Mutable environments:* 1TR
- *Description:* If `smb1` is present and true, iBoot allows objects such as a custom kernel collection to be Secure Enclave signed with the same key as the LocalPolicy. Presence of `smb0` is a prerequisite for presence of `smb1`. Users can change this field using command-line tools such as `csrutil` or `bputil` to downgrade to Permissive Security.

Secure Multi-Boot (smb2)

- *Type:* Boolean
- *Mutable environments:* 1TR
- *Description:* If `smb2` is present and true, iBoot allows the Auxiliary Kernel Collection to be Secure Enclave signed with the same key as the LocalPolicy. The presence of `smb0` is a prerequisite for the presence of `smb2`. Users can change this field using Startup Security Utility or `bputil` to downgrade to Reduced Security and enable third-party kexts.

Secure Multi-Boot (smb3)

- *Type:* Boolean
- *Mutable environments:* 1TR
- *Description:* If `smb3` is present and true, a user at the device has opted in to mobile device management (MDM) control of their system. Presence of this field makes the LocalPolicy-controlling Secure Enclave processor application accept MDM authentication instead of requiring local user authentication. Users can change this field using Startup Security Utility or `bputil` to enable managed control over third-party kexts and software updates.

Secure Multi-Boot (smb4)

- *Type:* Boolean
- *Mutable environments:* recoveryOS, macOS
- *Description:* If `smb4` is present and true, the device has opted in to MDM control of the operating system using the Apple School Manager or Apple Business Manager. Presence of this field makes the LocalPolicy-controlling Secure Enclave application accept MDM authentication instead of requiring local user authentication. This field is changed by the MDM solution when it detects that a device's serial number appears in Apple School Manager or Apple Business Manager.

System Integrity Protection (sip0)

- *Type:* 64 bit unsigned integer
- *Mutable environments:* 1TR
- *Description:* The `sip0` holds the existing System Integrity Protection (SIP) policy bits that previously were stored in NVRAM. New SIP policy bits are added here (instead of using LocalPolicy fields like the below) if they're used only in macOS and not used by LLB. Users can change this field using `csrutil` from 1TR to disable SIP and downgrade to Permissive Security.

System Integrity Protection (sip1)

- *Type:* Boolean
- *Mutable environments:* 1TR
- *Description:* If `sip1` is present and true, iBoot should allow failures to verify the SSV volume root hash. Users can change this field using `csrutil` or `bputil` from 1TR.

System Integrity Protection (sip2)

- *Type:* Boolean
- *Mutable environments:* 1TR
- *Description:* `sip2` allows disabling of the kernel System Coprocessor Integrity Protection (SCIP). If this tag is present and true, iBoot tells the kernel that it should not enforce SCIP. Users can change this field using `csrutil` or `bputil` from 1TR.

System Integrity Protection (sip3)

- *Type:* Boolean
- *Mutable environments:* 1TR
- *Description:* If `sip3` is present and true, iBoot doesn't enforce its built-in allow list for the boot-args NVRAM variable, which would otherwise filter the options passed to the kernel. Users can change this field using `csrutil` or `bputil` from 1TR.

Certificates and RemotePolicy

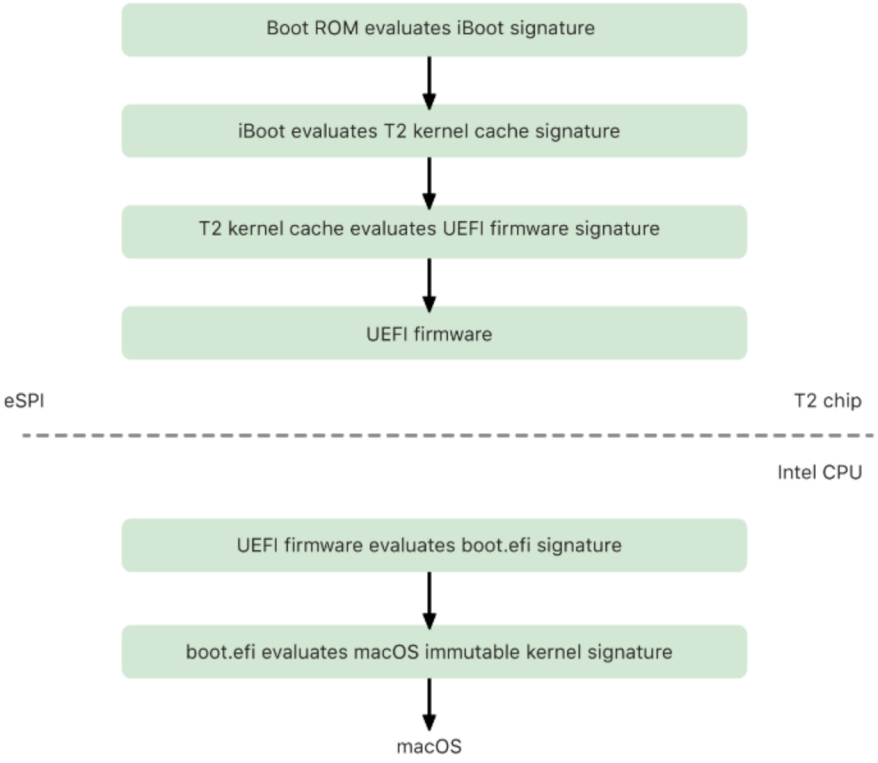
As described in LocalPolicy signing-key creation and management, the LocalPolicy Image4 also contains the Owner Identity Certificate (OIC) and the embedded RemotePolicy.

Intel-based Mac computers

Boot process for an Intel-based Mac

Intel-based Mac with an Apple T2 Security Chip

When an Intel-based Mac computer with the Apple T2 Security Chip is turned on, the chip performs a secure boot from its Boot ROM in the same fashion as iPhone, iPad, and a Mac with Apple silicon. This verifies the iBoot bootloader and is the first step in the chain of trust. iBoot checks the kernel and kernel extension code on the T2 chip, which then checks the Intel UEFI firmware. The UEFI firmware and the associated signature are initially available only to the T2 chip.



macOS T2 secure boot chain.

After verification, the UEFI firmware image is mapped into a portion of the T2 chip memory. This memory is made available to the Intel CPU through the enhanced Serial Peripheral Interface (eSPI). When the Intel CPU first boots, it fetches the UEFI firmware through the eSPI from the integrity-checked, memory-mapped copy of the firmware located on the T2 chip.

The evaluation of the chain of trust continues on the Intel CPU, with the UEFI firmware evaluating the signature for boot.efi, which is the macOS bootloader. The Intel-resident macOS secure boot signatures are stored in the same Image4 format used for iOS, iPadOS, and T2 chip secure boot, and the code that parses the Image4 files is the same hardened code from the current iOS and iPadOS secure boot implementation. Boot.efi in turn verifies the signature of a new file, called immutablekernel. When secure boot is enabled, the immutablekernel file represents the complete set of Apple kernel extensions required to boot macOS. The secure boot policy terminates at the handoff to the immutablekernel, and after that, macOS security policies (such as System Integrity Protection and signed kernel extensions) take effect.

If there are any errors or failures in this process, the Mac enters Recovery mode, Apple T2 Security Chip Recovery mode, or Apple T2 Security Chip Device Firmware Upgrade (DFU) mode mode.

Microsoft Windows on an Intel-based Mac with a T2 chip

By default, an Intel-based Mac that supports secure boot trust only content signed by Apple. However, to improve the security of Boot Camp installations, Apple also supports secure booting for Windows. The UEFI firmware includes a copy of the Microsoft Windows Production CA 2011 certificate used to authenticate Microsoft bootloaders.

Note: There is currently no trust provided for the Microsoft Corporation UEFI CA 2011 that would allow verification of code signed by Microsoft partners. This UEFI CA is commonly used to verify the authenticity of bootloaders for other operating systems, such as Linux variants.

Support for secure boot of Windows isn't enabled by default; instead, it's enabled using Boot Camp Assistant (BCA). When a user runs BCA, macOS is reconfigured to trust Microsoft first-party signed code during boot. After BCA completes, if macOS fails to pass the Apple first-party trust evaluation during secure boot, the UEFI firmware attempts to evaluate the trust of the object according to UEFI secure boot formatting. If the trust evaluation succeeds, the Mac proceeds and boots Windows. If not, the Mac enters recoveryOS and informs the user of the trust evaluation failure.

Intel-based Mac computers without a T2 chip

An Intel-based Mac without a T2 chip doesn't support secure boot. Therefore the UEFI firmware loads the macOS booter (boot.efi) from the file system without verification, and the booter loads the kernel (prelinkedkernel) from the file system without verification. To protect the integrity of the boot chain, users should enable all of the following security mechanisms:

- *System Integrity Protection (SIP):* Enabled by default, this protects the booter and kernel against malicious writes from within a running macOS.
- *FileVault:* This can be enabled in two ways: by the user or by a mobile device management (MDM) administrator. This protects against a physically present attacker using Target Disk Mode to overwrite the booter.

- *Firmware Password:* This can be enabled in two ways: by the user or by an MDM administrator. This protects a physically present attacker from launching alternate boot modes such as recoveryOS, Single User Mode, or Target Disk Mode from which the booter can be overwritten. This also prevents booting from alternate media, by which an attacker could run code to overwrite the booter.

The unlocking process of an Intel-based Mac without a T2 chip.

Boot modes of an Intel-based Mac with an Apple T2 Security Chip

An Intel-based Mac with an Apple T2 Security Chip has a variety of boot modes that can be entered at boot time by pressing key combinations, which are recognized by the UEFI firmware or booter. Some boot modes, such as Single User Mode, won't work unless the security policy is changed to No Security in Startup Security Utility.

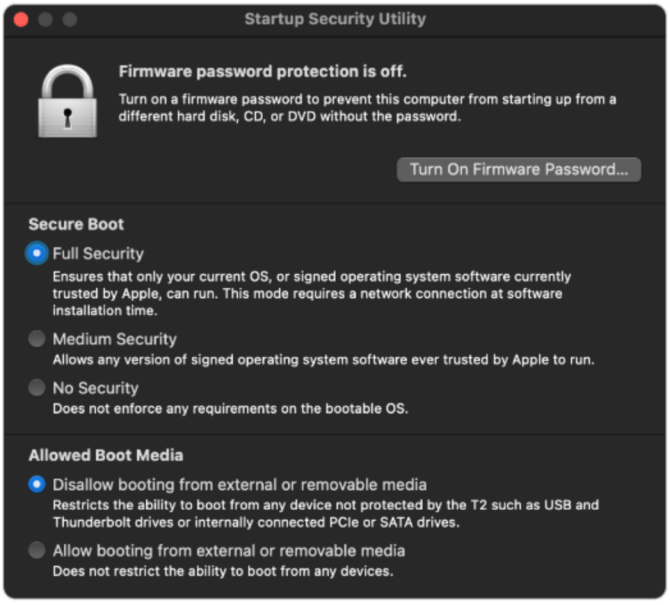
Mode	Key combo	Description
macOS boot	None	The UEFI firmware hands off to the macOS booter (a UEFI application), which hands off to the macOS kernel. On standard booting of a Mac with FileVault enabled, the macOS booter presents the Login Window interface, which takes the password to decrypt the storage.
Startup Manager	Option (⌥)	The UEFI firmware launches the built-in UEFI application that presents the user with a boot device selection interface.
Target Disk Mode (TDM)	T	The UEFI firmware launches the built-in UEFI application that exposes the internal storage device as a raw, block-based storage device over FireWire, Thunderbolt, USB, or any combination of the three (depending on the Mac model).
Single User Mode	Command (⌘)-S	The macOS kernel passes the –s flag in launchd's argument vector, then launchd creates the single-user shell in the Console app's tty. *Note:* If the user exits the shell, macOS continues boot to the Login Window.
recoveryOS	Command (⌘)-R	The UEFI firmware loads a minimal macOS from a signed disk image (.dmg) file on the internal storage device.
Internet recoveryOS	Option (⌥)-Command (⌘)-R	The signed disk image is downloaded from the internet using HTTP.

Mode	Key combo	Description
Diagnostics	D	The UEFI firmware loads a minimal UEFI diagnostic environment from a signed disk image file on the internal storage device.
Internet Diagnostics	Option (⌥)-D	The signed disk image is downloaded from the internet using HTTP.
Windows boot	None	If Windows has been installed using Boot Camp, the UEFI firmware hands off to the Windows booter, which hands off to the Windows kernel.

Startup Security Utility on a Mac with an Apple T2 Security Chip

Overview

On an Intel-based Mac with an Apple T2 Security Chip, Startup Security Utility handles a number security policy settings. The utility is accessible by booting into recoveryOS and selecting Startup Security Utility from the Utilities menu and protects supported security settings from easy manipulation by an attacker.



A screenshot of the Startup Security Utility.

Critical policy changes require authentication, even in Recovery mode. When Startup Security Utility is first opened, it prompts the user to enter an administrator password from the primary macOS installation associated with the currently booted recoveryOS. If no administrator exists, one must be created before the policy can be changed. The T2 chip requires that the Mac computer is currently booted into recoveryOS and that an authentication with a Secure Enclave–backed credential has occurred before such a policy change can be made. Security policy changes have two implicit requirements. recoveryOS must:

- Be booted from a storage device directly connected to the T2 chip, because partitions on other devices don't have Secure Enclave–backed credentials bound to the internal storage device.
- Reside on an APFS-based volume, because there is support only for storing the Authentication in Recovery credentials sent to the Secure Enclave on the "Preboot" APFS volume of a drive. HFS plus-formatted volumes can't use secure boot.

This policy is shown only in Startup Security Utility on an Intel-based Mac with a T2 chip. Although most use cases shouldn't require changes to the secure boot policy, users are ultimately in control of their device's settings and may choose, depending on their needs, to disable or downgrade the secure boot functionality on their Mac.

Secure boot policy changes made from within this app apply only to the evaluation of the chain of trust being verified on the Intel processor. The option "Secure boot the T2 chip" is always in effect.

Secure boot policy can be configured to one of three settings: Full Security, Medium Security, and No Security. No Security completely disables secure boot evaluation on the Intel processor and allows the user to boot whatever they want.

Full Security boot policy

Full Security is the default boot policy, and it behaves similar to iOS and iPadOS or Full Security on a Mac with Apple silicon. At the time that software is downloaded and prepared to install, it is personalized with a signature that includes the Exclusive Chip Identification (ECID)—a unique ID specific to the T2 chip in this case—as part of the signing request. The signature given back by the signing server is then unique and usable only by that particular T2 chip. When the Full Security policy is in effect, the UEFI firmware ensures that a given signature isn't just signed by Apple but is signed for this specific Mac, essentially tying that version of macOS to that Mac. This helps prevent rollback attacks as described for Full Security on a Mac with Apple silicon.

Medium Security boot policy

Medium Security boot policy is somewhat like a traditional UEFI secure boot, in which a vendor (here, Apple) generates a digital signature for the code to assert it came from the vendor. In this way, attackers are prevented from inserting unsigned code. We refer to this signature as a "global" signature because it can be used on any Mac, for any amount of time, for a Mac that currently has a Medium Security policy set. Neither iOS, iPadOS, nor the T2 chip itself support global signatures. This setting doesn't attempt to prevent rollback attacks.

Media boot policy

Media boot policy exists only on an Intel-based Mac with a T2 chip and is independent from the secure boot policy. So even if a user disables secure boot, this doesn't change the default behavior of preventing anything other than the storage device directly connected to the T2 chip to boot the Mac. (Media boot policy is not required on a Mac with Apple silicon. For more information, see Startup Disk security policy control.

Firmware password protection in an Intel-based Mac

macOS on Intel-based Mac computers with an Apple T2 Security Chip supports the use of a Firmware Password to prevent unintended modifications of firmware settings on a specific Mac. The Firmware Password is used to prevent selecting alternate boot modes such as booting into recoveryOS or Single User Mode, booting from an unauthorized volume, or booting into Target Disk Mode.

Note: The firmware password isn't required on a Mac with Apple silicon, because the critical firmware functionality it restricted has been moved into the recoveryOS and (when FileVault is enabled) recoveryOS requires user authentication before its critical functionality can be reached.

The most basic mode of firmware password can be reached from the recoveryOS Firmware Password Utility on an Intel-based Mac *without* a T2 chip, and from the Startup Security Utility on an Intel-based Mac *with* a T2 chip. Advanced options (such as the ability to prompt for the password at every boot) are available from the `firmwarepasswd` command-line tool in macOS.

Setting a Firmware Password is especially important to reduce the risk of attacks on Intel-based Macs without a T2 chip from a physically present attacker. The Firmware Password can help prevent an attacker from booting to recoveryOS, from where they could otherwise disable System Integrity Protection (SIP). And by restricting boot of alternative media, an attacker can't execute privileged code from another operating system to attack peripheral firmwares.

A firmware password reset mechanism exists to help users who forget their password. Users press a key combination at startup, and are presented with a model-specific string to provide to AppleCare. AppleCare digitally signs a resource that is signature checked by the Uniform Resource Identifier (URI). If the signature is validated and the content is for the specific Mac, the UEFI firmware removes the firmware password.

For users who want no one but themselves to remove their firmware password by software means, the `–disable–reset–capability` option has been added to the `firmwarepasswd` command-line tool in macOS 10.15. Before setting this option, users must acknowledge that if the password is forgotten and needs removal, the user must bear the cost of the logic board replacement necessary to achieve this. Organizations that want to protect their Mac computers from external attackers and from employees must set a firmware password on organization-owned systems. This can be accomplished on the device in any of the following ways:

- At provisioning time, by manually using the `firmwarepasswd` command-line tool
- With third-party management tools that use the `firmwarepasswd` command-line tool
- Using mobile device management (MDM)

recoveryOS and diagnostics environments for an Intel-based Mac

recoveryOS

The recoveryOS is completely separate from the main macOS, and the entire contents are stored in a disk image file named BaseSystem.dmg. There is also an associated BaseSystem.chunklist which is used to verify the integrity of the BaseSystem.dmg. The chunklist is a series of hashes for 10 MB chunks of the BaseSystem.dmg. The UEFI firmware evaluates the signature of the chunklist file and then evaluates the hash for one chunk at a time from the BaseSystem.dmg, to ensure that it matches the signed content present in the chunklist. If any of these hashes don't match, booting from the local recoveryOS is aborted and the UEFI firmware attempts to boot from Internet recoveryOS instead.

If the verification is successfully completed, the UEFI firmware mounts the BaseSystem.dmg as a RAM disk and launches the boot.efi file that's in it. There's no need for the UEFI firmware to do a specific check of the boot.efi, nor for the boot.efi to do a check of the kernel, because the completed contents of the operating system (of which these elements are only a subset) have already been integrity checked.

Apple Diagnostics

The procedure for booting the local diagnostic environment is mostly the same as launching the recoveryOS. Separate AppleDiagnostics.dmg and AppleDiagnostics.chunklist files are used, but they're verified in the same way as the BaseSystem files are. Instead of launching boot.efi, the UEFI firmware launches a file inside the disk image (.dmg file) named diags.efi, which is in turn responsible for invoking a variety of other UEFI drivers that can interface with and check for errors in the hardware.

Internet recoveryOS and diagnostic environment

If an error has occurred in the launching of the local recovery or diagnostic environments, the UEFI firmware attempts to download the images from the internet instead. (A user can also specifically request the images to be fetched from the internet using special key sequences held at boot.) The integrity validation of the disk images and chunklists downloaded from the OS Recovery Server is performed the same way as with images retrieved from a storage device.

While the connection to the OS Recovery Server is done using HTTP, the complete downloaded contents are still integrity checked as previously described, and as such are protected against manipulation by an attacker with control of the network. In the event that an individual chunk fails integrity verification, it is re-requested from the OS Recovery Server 11 times, before giving up and displaying an error.

When the internet recovery and diagnostic modes were added to Mac computers in 2011, it was decided that it would be better to use the simpler HTTP transport, and handle content authentication using the chunklist mechanism, rather than implement the more complicated HTTPS functionality in the UEFI firmware, and thus increase the firmware's attack surface.

Secure software updates

Overview

Security is a process; it isn't enough to reliably boot the operating system version installed at the factory—there must also exist a mechanism to quickly and securely obtain the latest security updates. Apple regularly releases software updates to address emerging security concerns. Users of iOS and iPadOS devices receive update notifications on the device and through the Finder (in macOS 10.15 or later) and iTunes (in macOS 10.14 or earlier). Mac users find available updates in System Preferences. Updates are delivered wirelessly, for rapid adoption of the latest security fixes.

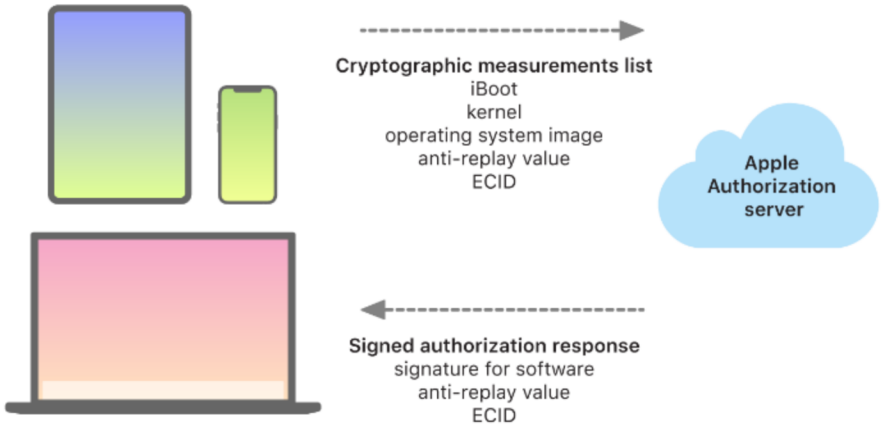
The update process uses the same hardware-based root of trust that secure boot uses that is designed to install only Apple-signed code. The update process also uses system software authorization to check that only copies of operating system versions that are actively being signed by Apple can be installed on iOS and iPadOS devices, or on Mac computers with the Full Security setting configured as the secure boot policy in Startup Security Utility. With these secure processes in place, Apple can stop signing older operating system versions with known vulnerabilities and help prevent downgrade attacks.

For greater software update security, when the device to be upgraded is physically connected to a Mac, a full copy of iOS or iPadOS is downloaded and installed. But for over-the-air (OTA) software updates, *only the components required to complete an update are downloaded,* improving network efficiency by not downloading the entire operating system. What's more, software updates can be cached on a Mac running macOS 10.13 or later with Content Caching turned on, so that iOS and iPadOS devices don't need to redownload the necessary update over the internet. (They still need to contact Apple servers to complete the update process.)

Personalized update process

During upgrades and updates, a connection is made to the Apple installation authorization server, which includes a list of cryptographic measurements for each part of the installation bundle to be installed (for example, iBoot, the kernel, and the operating system image), a random anti-replay value (the nonce), and the device's unique Exclusive Chip Identification (ECID).

The authorization server checks the presented list of measurements against versions for which installation is permitted and, if it finds a match, adds the ECID to the measurement and signs the result. The server passes a complete set of signed data to the device as part of the upgrade process. Adding the ECID "personalizes" the authorization for the requesting device. By authorizing and signing only for known measurements, the server ensures that the update takes place exactly as Apple provided.



How Apple devices interact with the Apple Authorization server.

The boot-time chain-of-trust evaluation verifies that the signature comes from Apple and that the measurement of the item loaded from the storage device, combined with the device's ECID, matches what was covered by the signature. These steps ensure that, on devices that support personalization, the authorization is for a specific device and that an older operating system of firmware version from one device can't be copied to another. The nonce prevents an attacker from saving the server's response and using it to tamper with a device or otherwise alter the system software.

The personalization process is why a network connection to Apple is always required to update any device with Apple-designed silicon, including an Intel-based Mac with the Apple T2 Security Chip.

Finally, the user's data volume is never mounted during a software update, to help prevent anything being read from or written to that volume during updates.

On devices with the Secure Enclave, that hardware similarly uses system software authorization to check the integrity of its software and prevent downgrade installations.

Operating system integrity

Apple designs its operating systems with security at the core, with a hardware root of trust leveraged to enable secure boot and a secure software update process that enables quick and safe application of security updates. Apple operating systems also leverage purpose-built silicon-based hardware capabilities to help prevent exploitation as the operating system runs. These capabilities are not focused on whether or not to execute code (see App security for the mechanisms that help prevent malware and ensure only trusted code is executed) but instead focus on the capabilities afforded to code that is executed. In short, they help mitigate attack and exploit techniques whether those originate from a malicious app, from the web, or through any other channel. Protections listed here are available on devices with supported Apple-designed SoCs—this includes iOS, iPadOS, tvOS, watchOS, and now macOS on a Mac with Apple silicon.

Feature	A10	A11, S3	A12, S4	A13, S5	A14, S6	M1
Kernel Integrity Protection	✓	✓	✓	✓	✓	✓
Fast Permission Restrictions		✓	✓	✓	✓	✓
System Coprocessor Integrity Protection			✓	✓	✓	✓
Pointer Authentication Codes			✓	✓	✓	✓
Page Protection Layer		✓	✓	✓	✓	See Note below.

Note: Page Protection Layer (PPL) requires that the platform execute *only* signed and trusted code; this is a security model that isn't applicable on macOS.

Kernel Integrity Protection

After the operating system kernel completes initialization, Kernel Integrity Protection (KIP) is enabled to prevent modifications of kernel and driver code. The memory controller provides a protected physical memory region that iBoot uses to load the kernel and kernel extensions. After startup completes, the memory controller denies writes to the protected physical memory region. The Application Processor's Memory Management Unit (MMU) is configured to prevent mapping privileged code from physical memory outside the protected memory region and to prevent writeable mappings of physical memory within the kernel memory region.

To prevent reconfiguration, the hardware used to enable KIP is locked after the boot process is complete.

Fast Permission Restrictions

Starting with the Apple A11 Bionic and S3 SoCs, a new hardware primitive was introduced. This primitive, Fast Permission Restrictions, includes a CPU register that quickly restricts permissions per thread. With Fast Permission Restrictions (also known as APRR registers), supported operating systems can remove execute permissions from memory without the overhead of a system call and a page table walk or flush. These registers provide one more level of mitigation for attacks from the web, particularly for code compiled at runtime (just-in-time compiled)—because memory can't be effectively executed at the same time it's being read from and written to.

System Coprocessor Integrity Protection

Coprocessor firmware handles many critical system tasks—for example, the Secure Enclave, the image sensor processor, and the motion coprocessor. Therefore its security is a key part of the security of the overall system. To prevent modification of coprocessor firmware, Apple uses a mechanism called *System Coprocessor Integrity Protection (SCIP)*.

SCIP works much like Kernel Integrity Protection (KIP): At boot time, iBoot loads each coprocessor's firmware into a protected memory region, one that's reserved and separate from the KIP region. iBoot configures each coprocessor's memory unit to prevent:

- Executable mappings outside its part of the protected memory region
- Writeable mappings inside its part of the protected memory region

Also at boot time, to configure SCIP for the Secure Enclave, the Secure Enclave operating system is used. After the boot process is complete, the hardware used to enable SCIP is locked to prevent reconfiguration.

Pointer Authentication Codes

Pointer Authentication Codes (PACs) are used to protect against exploitation of memory corruption bugs. System software and built-in apps use PAC to prevent modification of function pointers and return addresses (code pointers). PAC uses five secret 128-bit values to sign kernel instructions and data, and each user space process has its own B keys. Items are salted and signed as indicated below.

Item	Key	Salt
Function Return Address	IB	Storage address
Function Pointers	IA	0
Block Invocation Function	IA	Storage address
Objective-C Method Cache	IB	Storage address + Class + Selector
C++ V-Table Entries	IA	Storage address + Hash (mangled method name)
Computed Goto Label	IA	Hash (function name)
Kernel Thread State	GA	•
User Thread State Registers	IA	Storage address
C++ V-Table Pointers	DA	0

The signature value is stored in the unused padding bits at the top of the 64-bit pointer. The signature is verified before use, and the padding is restored to ensure a functioning pointer address. Failure to verify results in an abort. This verification increases the difficulty of many attacks, such as a return-oriented programming (ROP) attack, which attempts to trick the device into executing existing code maliciously by manipulating function return addresses stored on the stack.

Page Protection Layer

Page Protection Layer (PPL) in iOS, iPadOS, and watchOS prevents user space code from being modified after code signature verification is complete. Building on Kernel Integrity Protection and Fast Permission Restrictions, PPL manages the page table permission overrides to make sure only the PPL can alter protected pages containing user code and page tables. The system provides a massive reduction in attack surface by supporting systemwide code integrity enforcement, even in the face of a compromised kernel. This protection isn't offered in macOS because PPL is only applicable on systems where all executed code must be signed.

Additional macOS system security capabilities

Additional macOS system security capabilities

macOS operates on a broader set of hardware (for example, Intel-based CPUs, Intel-based CPUs in combination with the Apple T2 Security Chip, and Apple silicon-based SoCs) and supports a range of general-purpose computing use cases. Whereas some users use only the basic preinstalled apps or those available from the App Store, others are kernel hackers who need to disable essentially all platform protections so they can run and test their executing code as with the highest levels of trust. Most fall somewhere between, and many of those have peripherals and software that require varying levels of access. Apple designed the macOS platform with an integrated approach to hardware, software, and services—a platform that provides security by design and makes it simple to configure, deploy, and manage but that retains the configurability that users expect. macOS also includes the key security technologies that an IT professional needs to help protect corporate data and integrate within secure enterprise networking environments.

The following capabilities support and help secure the varied needs of macOS users. They include:

- Signed system volume security
- System Integrity Protection
- Trust caches
- Protection for peripherals
- Rosetta 2 (automatic translation) support and security for a Mac with Apple silicon
- DMA support and protections
- Kernel extension (kext) support and security
- Option ROM support and security
- UEFI firmware security for Intel-based Mac computers

Signed system volume security in macOS

For macOS 10.15, Apple introduced the read-only system volume, a dedicated, isolated volume for system content. macOS 11 adds strong cryptographic protections to system content with a *signed system volume (SSV)*. SSV features a kernel mechanism that verifies the integrity of the system content at runtime and rejects any data—code and noncode—without a valid cryptographic signature from Apple.

SSV not only helps prevent tampering with any Apple software that's part of the operating system, it also makes macOS software update more reliable and much safer. And because SSV uses APFS (Apple File System) snapshots, if an update can't be performed, the old system version can be restored without reinstallation.

Since its introduction, APFS has provided file-system metadata integrity using noncryptographic checksums on the internal storage device. SSV strengthens the integrity mechanism by adding cryptographic hashes, thus extending it to encompass every byte of file data. Data from the internal storage device (including file system metadata) is cryptographically hashed in the read path, and the hash is then compared with an expected value in the file-system metadata. In case of mismatch, the system assumes the data has been tampered with and won't return it to the requesting software.

Each SSV SHA256 hash is stored in the main file-system metadata tree, which is itself hashed. And because each node of the tree recursively verifies the integrity of the hashes of its children—similar to a binary hash (Merkle) tree—the root node's hash value, called a *seal*, therefore encompasses every byte of data in the SSV, which means the cryptographic signature covers the entire system volume.

During macOS installation and update, the seal is recomputed from the file system on-device and that measurement is verified against the measurement which Apple signed. On a Mac with Apple silicon, the bootloader verifies the seal before transferring control to the kernel. On an Intel-based Mac with an Apple T2 Security Chip, the bootloader forwards the measurement and signature to the kernel, which then verifies the seal directly before mounting the root file system. In either case, if the verification fails, the startup process halts and the user is prompted to reinstall macOS.

This procedure is repeated at every boot unless the user has elected to enter a lower security mode and has separately chosen to disable the signed system volume.

SSV and code signing

Code signing is still present and enforced by the kernel. The signed system volume provides protection when any bytes at all are read from the internal storage device. In contrast, code signing provides protection when Mach objects are memory mapped as executable. Both SSV and code signing protect executable code on all read and execute paths.

SSV and FileVault

In macOS 11, equivalent at-rest protection for system content is provided by the SSV, and therefore the system volume no longer needs to be encrypted. Any modifications made to the file system while it's at rest will be detected by the file system when they're read. If the user has enabled FileVault, the user's content on the data volume is still encrypted with a user-provided secret.

If the user chooses to disable the SSV, the system at rest becomes vulnerable to tampering, and this tampering could enable an attacker to extract encrypted user data when the system next starts up. Therefore the system won't permit the user to disable the SSV if FileVault is enabled. Protection while at rest must be enabled or disabled for both volumes in a consistent manner.

In macOS 10.15 or earlier, FileVault protects operating system software while at rest by encrypting user and system content with a key protected by a user-provided secret. This protects against an attacker with physical access to the device from accessing or effectively modifying the file system containing system software.

SSV and a Mac with an Apple T2 Security Chip

On a Mac with an Apple T2 Security Chip, only macOS itself is protected by the SSV. The software that runs on the T2 chip and verifies macOS is protected by secure boot.

System Integrity Protection

macOS utilizes kernel permissions to limit writability of critical system files with a feature called *System Integrity Protection (SIP)*. This feature is separate and in addition to the hardware-based Kernel Integrity Protection (KIP) available on a Mac with Apple silicon, which protects modification of the kernel in memory. Mandatory access control technology is leveraged to provide this and a number of other kernel level protections, including sandboxing and Data Vault.

Mandatory access controls

macOS uses mandatory access controls—policies that set security restrictions, created by the developer, that can't be overridden. This approach is different from discretionary access controls, which permit users to override security policies according to their preferences.

Mandatory access controls aren't visible to users, but they're the underlying technology that helps enable several important features, including sandboxing, parental controls, managed preferences, extensions, and System Integrity Protection.

System Integrity Protection

System Integrity Protection restricts components to read-only in specific critical file system locations to prevent malicious code from modifying them. System Integrity Protection is a computer-specific setting that's on by default when a user upgrades to OS X 10.11 or later. On an Intel-based Mac, disabling it removes protection for all partitions on the physical storage device. macOS applies this security policy to every process running on the system, regardless of whether it's running sandboxed or with administrative privileges.

Trust caches

One of the objects included in the Secure Boot chain is the static trust cache, a trusted record of all the Mach-O binaries that are mastered into the signed system volume. Each Mach-O is represented by a code directory hash. For efficient searching, these hashes are sorted before being inserted into the trust cache. The code directory is the result of the signing operation performed by `codesign(1)`. To enforce the trust cache, SIP must remain enabled. To disable trust cache enforcement on a Mac with Apple silicon, secure boot must be configured to Permissive Security.

When a binary is executed (whether as part of spawning a new process or mapping executable code into an existing process), its code directory is extracted and hashed. If the resulting hash is found in the trust cache, the executable mappings created for the binary will be granted platform privileges—that is, they may possess any entitlement and execute without further verification as to the authenticity of the signature. This is in contrast to an Intel-based Mac, where platform privileges are conveyed to operating system content by the Apple certificate that signs the binaries. (This certificate doesn't constrain which entitlements the binary may possess.)

Nonplatform binaries (for example, notarized third-party code) must have valid certificate chains in order to execute, and the entitlements they may possess are constrained by the signing profile issued to the developer by the Apple Developer Program.

All binaries shipped within macOS are signed with a *platform identifier*. On a Mac with Apple silicon, this identifier is used to indicate that even though the binary is signed by Apple, its code directory hash must be present in the trust cache in order to execute. On an Intel-based Mac, the platform identifier is used to perform targeted revocation of a binaries from an older release of macOS and prevent them from executing on newer versions.

The static trust cache completely locks a set of binaries to a given version of macOS. This behavior prevents legitimately Apple-signed binaries from older operating systems from being introduced into newer ones in order for an attacker to gain advantage.

Platform code shipped outside the operating system

Apple ships some binaries—for example, Xcode and the development tools stack—that aren't signed with a platform identifier. Even so, they're still permitted to execute with platform privileges on a Mac with Apple silicon and those with a T2 chip. Because this platform software is shipped independently of macOS, it isn't subject to the revocation behaviors imposed by the static trust cache.

Loadable trust caches

Apple ships certain software packages with *loadable trust caches*. These caches have the same data structure as the static trust cache. But although there's only one static trust cache—and its contents are always guaranteed to be locked into read-only ranges after the kernel's early initialization is complete—loadable trust caches are added to the system at runtime.

These trust caches are authenticated either through the same mechanism that authenticates boot firmware (personalization using the Apple trusted signing service) or as globally signed objects (whose signatures don't bind them to a particular device).

One example of a personalized trust cache is the cache, shipped with the disk image that's used to perform field diagnostics on a Mac with Apple silicon. This trust cache is personalized, along with the disk image, and loaded into the subject Mac computer's kernel while it's booted into a diagnostic mode. The trust cache enables the software within the disk image to run with platform privilege.

An example of a globally signed trust cache is shipped with macOS software updates. This trust cache permits a chunk of code within the software update—the *update brain*—to run with platform privilege. The update brain performs any work to stage the software update that the host system lacks the capacity to perform in a consistent fashion across versions.

Peripheral processor security in Mac computers

All modern computing systems have many built-in peripheral processors dedicated to tasks such as networking, graphics, power management, and more. These peripheral processors are often single-purpose and are much less powerful than the primary CPU. Built-in peripherals that don't implement sufficient security become an easier target for attackers to exploit, through which they can persistently infect the operating system. Having infected a peripheral processor firmware, an attacker could target software on the primary CPU or directly capture sensitive data (For example, an Ethernet device could see the contents of packets that aren't encrypted.)

Whenever possible, Apple works to reduce the number of peripheral processors necessary or works to avoid designs that require firmware. But when separate processors with their own firmware are required, efforts are taken to ensure an attacker can't persist on that processor. This can be by verifying the processor in one of two ways:

- Running the processor so that it downloads verified firmware from the primary CPU on startup
- Having the peripheral processor implement its own secure boot chain, to verify the peripheral processor firmware every time the Mac starts up

Apple works with vendors to audit their implementations and enhance their designs to include desired properties such as:

- Ensuring minimum cryptographic strengths
- Ensuring strong revocation of known bad firmware
- Disabling debug interfaces
- Signing the firmware with cryptographic keys that are stored in Apple-controlled hardware security modules (HSMs)

In recent years, Apple has worked with some external vendors to adopt the same "Image4" data structures, verification code, and signing infrastructure used by Apple silicon.

When neither storage-free operation nor storage plus secure boot is an option, the design mandates that firmware updates be cryptographically signed and verified before the persistent storage can be updated.

Rosetta 2 on a Mac with Apple silicon

A Mac with Apple silicon is capable of running code compiled for the x86_64 instruction set using a translation mechanism called *Rosetta 2*. There are two types of translation offered: just in time and ahead of time.

Just-in-time translation

In the just-in-time (JIT) translation pipeline, an x86_64 Mach object is identified early in the image execution path. When these images are encountered, the kernel transfers control to a special Rosetta translation stub rather than to the dynamic link editor, `dyld(1)`. The translation stub then translates x86_64 pages during the image's execution. This translation takes place entirely within the process. The kernel still verifies the code hashes of each x86_64 page against the code signature attached to the binary as the page is faulted in. In the event of a hash mismatch, the kernel enforces the remediation policy appropriate for that process.

Ahead-of-time translation

In the ahead-of-time (AOT) translation path, x86_64 binaries are read from storage at times the system deems optimal for responsiveness of that code. The translated artifacts are written to storage as a special type of Mach object file. That file is similar to an executable image, but it's marked to indicate it's the translated product of another image.

In this model, the AOT artifact derives all of its identity information from the original x86_64 executable image. To enforce this binding, a privileged userspace entity signs the translation artifact using a device-specific key that's managed by the Secure Enclave. This key is released only to the privileged userspace entity, which is identified as such using a restricted entitlement. The code directory created for the translation artifact includes the code directory hash of the original x86_64 executable image. The signature on the translation artifact itself is known as the *supplemental signature*.

The AOT pipeline begins similarly to the JIT pipeline, with the kernel transferring control to the Rosetta runtime rather than to the dynamic link editor, `dyld(1)`. But the Rosetta runtime then sends an interprocess communication (IPC) query to the Rosetta system service, which asks whether there's an AOT translation available for the current executable image. If found, the Rosetta service provides a handle to that translation, and it's mapped into the process and executed. During execution, the kernel enforces the code directory hashes of the translation artifact which are authenticated by the signature rooted in the device-specific signing key. The original x86_64 image's code directory hashes aren't involved in this process.

Translated artifacts are stored in a Data Vault which isn't runtime-accessible by any entity except for the Rosetta service. The Rosetta service manages access to its cache by distributing read-only file descriptors to individual translation artifacts; this limits access to the AOT artifact cache. This service's interprocess communication and dependent footprint are kept intentionally very narrow to limit its attack surface.

If the code directory hash of the original x86_64 image doesn't match with the one encoded into the AOT translation artifact's signature, this result is considered the equivalent of an invalid code signature, and appropriate enforcement action is taken.

If a remote process queries the kernel for the entitlements or other code identity properties of an AOT-translated executable, the identity properties of the original x86_64 image are returned to it.

Static trust cache content

macOS 11 or later ships with Mach "fat" binaries that contain slices of x86_64 and arm64 machine code. On a Mac with Apple silicon, the user may decide to execute the x86_64 slice of a system binary through the Rosetta pipeline—for example to load a plug-in that has no native arm64 variant. To support this approach, the static trust cache that ships with macOS, generally, contains three code directory hashes per Mach object file:

- A code directory hash of the arm64 slice
- A code directory hash of the x86_64 slice
- A code directory hash of the AOT translation of the x86_64 slice

The Rosetta AOT translation procedure is deterministic in that it reproduces identical output for any given input, irrespective of when the translation was performed or on what device it was performed.

During the macOS build, every Mach object file is run through the Rosetta AOT translation pipeline associated with the version of macOS being built, and the resulting code directory hash is recorded into the trust cache. For efficiency, the actual translated products don't ship with the operating system and are reconstituted on demand when the user requests them.

When an x86_64 image is being executed on a Mac with Apple silicon, if that image's code directory hash is in the static trust cache, the resulting AOT artifact's code directory hash is *also* expected to be in the static trust cache. Such products aren't signed by the device-specific key, because the signing authority is rooted in the Apple secure boot chain.

Unsigned x86_64 code

A Mac with Apple silicon doesn't permit native arm64 code to execute unless a valid signature is attached. This signature can be as simple as an ad hoc code signature (cf. `codesign(1)`) that doesn't bear any actual identity from the secret half of an asymmetric key pair (it's simply an unauthenticated measurement of the binary).

For binary compatibility, translated x86_64 code is permitted to execute through Rosetta with no signature information at all. No specific identity is conveyed to this code through the device-specific Secure Enclave signing procedure, and it executes with precisely the same limitations that native unsigned code executing on an Intel-based Mac.

Direct memory access protections for Mac computers

To achieve high throughput on high-speed interfaces like PCIe, FireWire, Thunderbolt, and USB, computers must support direct memory access (DMA) from peripherals. That is, they must be able to read and write to RAM without continuous involvement of the CPU. Since 2012, Mac computers have implemented numerous technologies to protect DMA, resulting in the best and most comprehensive set of DMA protections on any PC.

Direct memory access protections for a Mac with Apple silicon

Apple systems on chip contain an Input/Output Memory Management Unit (IOMMU) for each DMA agent in the system, including PCIe and Thunderbolt ports. Because each IOMMU has its own set of address translation tables to translate DMA requests, peripherals connected by PCIe or Thunderbolt can access only memory that has been explicitly mapped for their use. Peripherals can't access memory belonging to other parts of the system—such as the kernel or firmware—memory assigned to other peripherals. If an IOMMU detects an attempt by a peripheral to access memory that isn't mapped for that peripheral's use, it triggers a kernel panic.

Direct memory access protections for an Intel-based Mac

On an Intel-based Mac with Intel Virtualization Technology for Directed I/O (VT-d) initialize the IOMMU, enabling DMA remapping and interrupt remapping very early in the boot process to mitigate various classes of security vulnerabilities. The Apple IOMMU hardware begins operation with a default-deny policy, so the instant when the system is powered on, it automatically begins blocking DMA requests from peripherals. After being initialized by software, the IOMMUs begin allowing DMA requests from peripherals to memory regions that have been explicitly mapped for their use.

Note: Interrupt remapping for PCIe isn't necessary on a Mac with Apple silicon, because each IOMMU handles MSIs for its own peripherals.

Starting in macOS 11, all Mac computers with an Apple T2 Security Chip run UEFI drivers that facilitate DMA in a restricted ring 3 environment when these drivers are pairing with external devices. This property helps mitigate security vulnerabilities that may occur when a malicious device interacts with a UEFI driver in an unexpected way at boot time. In particular, it reduces the impact of vulnerabilities in a drivers handling of DMA buffers.

Kernel extensions in macOS

Starting with macOS 11, if third-party kernel extensions (kexts) are enabled, they can't be loaded into the kernel on demand. Instead, they're merged into an *Auxiliary Kernel Collection (AuxKC)*, which is loaded during the boot process. For a Mac with Apple silicon, the measurement of the AuxKC is signed into the LocalPolicy (for previous hardware, the AuxKC resided on the data volume). Rebuilding the AuxKC requires the user's approval and restarting of the macOS to load the changes into the kernel, and it requires that the secure boot be configured to Reduced Security.

Important: Kexts are no longer recommended for macOS. Kexts risk the integrity and reliability of the operating system, and Apple recommends users select solutions that don't require extending the kernel.

Kernel extensions in a Mac with Apple silicon

Kexts must be explicitly enabled for a Mac with Apple silicon by holding the power button at startup to enter into One True Recovery (1TR) mode, then downgrading to Reduced Security and checking the box to enable kernel extensions. This action also requires entering an administrator password to authorize the downgrade. The combination of the 1TR and password requirement makes it difficult for software-only attackers starting from within macOS to inject kexts into macOS, which they can then exploit to gain kernel privileges.

After a user authorizes kexts to load, the above User-Approved Kernel Extension Loading flow is used to authorize the installation of kexts. The authorization used for the above flow is also used to capture an SHA384 hash of the user-authorized kext list (UAKL) in the LocalPolicy. The kernel management daemon (`kmd`) is then responsible for validating only those kexts found in the UAKL for inclusion into the AuxKC.

- If System Integrity Protection (SIP) is enabled, the signature of each kext is verified before being included in the AuxKC.
- If SIP is disabled, the kext signature isn't enforced.

This approach allows Permissive Security flows for developers or users who aren't part of the Apple Developer Program to test kexts before they are signed.

After the AuxKC is created, its measurement is sent to the Secure Enclave to be signed and included in an Image4 data structure that can be evaluated by iBoot at startup. As part of the AuxKC construction, a kext receipt is also generated. This receipt contains the list of kexts that were actually included in the AuxKC, because the set could be a subset of the UAKL if banned kexts were encountered. An SHA384 hash of the AuxKC Image4 data structure and the kext receipt are included in the LocalPolicy. The AuxKC Image4 hash is used for extra verification by iBoot at startup to ensure that it isn't possible to start up an older Secure Enclave–signed AuxKC Image4 file with a newer LocalPolicy. The kext receipt is used by subsystems such as Apple Pay to determine whether there are any kexts currently loaded that could interfere with the trustworthiness of macOS. If there are, then Apple Pay capabilities may be disabled.

Alternatives to kexts (macOS 10.15 or later)

macOS 10.15 enables developers to extend the capabilities of macOS by installing and managing system extensions that run in user space rather than at the kernel level. By running in user space, system extensions increase the stability and security of macOS. Even though kexts inherently have full access to the entire operating system, extensions running in user space are granted only the privileges necessary to perform their specified function.

Developers can use frameworks, including DriverKit, EndpointSecurity, and NetworkExtension, to write USB and human interface drivers, endpoint security tools (like data loss prevention or other endpoint agents), and VPN and network tools, all without needing to write kexts. Third-party security agents should be used only if they take advantage of these APIs or have a robust road map to transition to them and away from kernel extensions.

User-Approved Kernel Extension Loading

To improve security, user consent is required to load kernel extensions installed with or after installing macOS 10.13. This process is known as *User-Approved Kernel Extension Loading*. Administrator authorization is required to approve a kernel extension. Kernel extensions don't require authorization if they:

- Were installed on a Mac when running macOS 10.12 or earlier
- Are replacing previously approved extensions
- Are allowed to load without user consent by using the `spctl` command-line tool available when a Mac was booted from recoveryOS
- Are allowed to load using mobile device management (MDM) configuration

  Starting with macOS 10.13.2, users can use MDM to specify a list of kernel extensions that load without user consent. This option requires a Mac running macOS 10.13.2 that's enrolled in MDM—through Apple School Manager, Apple Business Manager, or user-approved MDM enrollment.

Option ROM security in macOS

Note: Option ROMs aren't currently supported on a Mac with Apple silicon.

Option ROM security in a Mac with the Apple T2 Security Chip

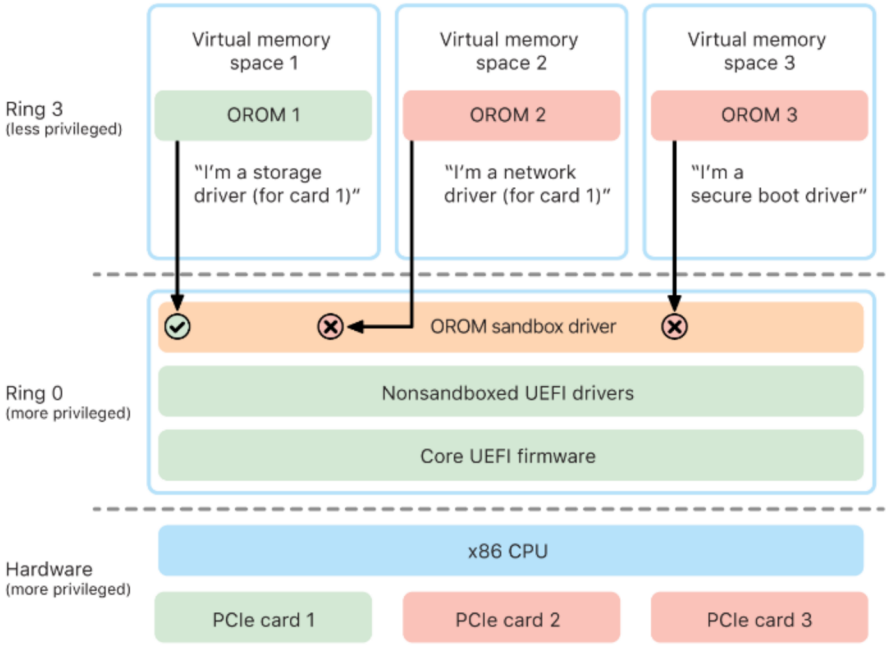
Both Thunderbolt and PCIe devices can have an "Option ROM (OROM)" physically attached to the device. (This is typically not a true ROM but is instead a rewritable chip that stores firmware.) On UEFI-based systems, that firmware is typically a UEFI driver, which is read in by the UEFI firmware and executed. The executed code is supposed to initialize and configure the hardware it was retrieved from, so that the hardware can be made usable by the rest of the firmware. This capability is required so that specialized third-party hardware can load and operate during the earliest startup phases—for example, to start up from external RAID arrays.

However, because OROMs are generally rewritable, if an attacker overwrites the OROM of a legitimate peripheral, the attacker's code executes early in the boot process and is able to tamper with the execution environment and violate the integrity of software that's loaded later. Likewise, if the attacker introduces their own malicious device to the system, they're also able to execute malicious code.

In macOS 10.12.3, the behavior of Mac computers sold after 2011 was changed to not execute OROMs by default at the time the Mac booted unless a special key combination was pressed. This key combination protected against malicious OROMs being inadvertently introduced into the macOS boot sequence. The default behavior of the Firmware Password Utility was also changed so that when the user set a firmware password, OROMs couldn't execute even if the key combination was pressed. This protected against a physically present attacker intentionally introducing a malicious OROM. For users who still need to run OROMs while they have a firmware password set, a nondefault option can be configured using the `firmwarepasswd` command-line tool in macOS.

OROM sandbox security

In macOS 10.15, UEFI firmware was updated to contain a mechanism for sandboxing OROMs and for stripping privileges from them. UEFI firmware typically executes all code, including OROMs, at the maximum CPU privilege level, called ring 0, and has a single shared virtual memory space for all code and data. Ring 0 is the privilege level where the macOS kernel runs, whereas the lower privilege level, ring 3, is where apps run. The OROM sandbox deprivileged OROMs by making use of virtual memory separation like the kernel does and then making the OROMs run in ring 3.



Option ROM (OROM) sandboxing.

The sandbox further significantly restricts both the interfaces that the OROMs can call (much like system call filtering in kernels) and the type of device that an OROM can register as (much like app approval.) The benefit of this design is that malicious OROMs can no longer directly write anywhere within ring 0 memory. Instead, they are limited to a very narrow and well-defined sandbox interface. This limited interface significantly reduces attack surface and forces attackers to first escape the sandbox and escalate privilege.

UEFI firmware security in an Intel-based Mac

Overview

Since 2006, Mac computers with an Intel-based CPU use an Intel firmware based on the Extensible Firmware Interface (EFI) Development Kit (EDK) version 1 or version 2. EDK2-based code conforms to the Unified Extensible Firmware Interface (UEFI) specification. This section refers to the Intel firmware as the *UEFI firmware*. The UEFI firmware was the first code to execute on the Intel chip.

For an Intel-based Mac without the Apple T2 Security Chip, the root of trust for the UEFI firmware is the chip where the firmware is stored. UEFI firmware updates are digitally signed by Apple and verified by the firmware before updating the storage. To prevent rollback attacks, updates must always have a version newer than the existing one. However, an attacker with physical access to the Mac could potentially use hardware to attach to the firmware storage chip and update the chip to contain malicious content. Likewise, if vulnerabilities are found in the early boot process of the UEFI firmware (before it write-restricts the storage chip), this could also lead to persistent infection of the UEFI firmware. This is a hardware architectural limitation common in most Intel-based PCs and present in all Intel-based Mac computers without the T2 chip.

To help prevent physical attacks that subvert UEFI firmware, Mac computers were rearchitected to root the trust in the UEFI firmware in the T2 chip. On these Mac computers, the root of trust for the UEFI firmware is specifically the T2 firmware, as described in Boot process for an Intel-based Mac.

Intel Management Engine (ME) subcomponent

One subcomponent stored within the UEFI firmware is the *Intel Management Engine (ME)* firmware. The ME—a separate processor and subsystem within Intel chips—is used primarily for audio and video copyright protection on a Mac that has only Intel-based graphics. To reduce this subcomponent's attack surface, an Intel-based Mac runs a custom ME firmware from which most components have been removed. Because the resulting Mac ME firmware is smaller than the default minimal build that Intel makes available, many components that have been the subject of public attacks by security researchers in the past are no longer present.

System Management Mode (SMM)

Intel processors have a special execution mode that's distinct from normal operation. Called *System Management Mode (SMM)*, it was originally introduced to handle time-sensitive operations such as power management. However, to perform such actions, Mac computers have historically used a discrete microcontroller called the *System Management Controller (SMC)*. No longer a separate microcontroller, the SMC has been integrated into the T2 chip.

System security for watchOS

Apple Watch uses many of the same hardware-based platform security capabilities to perform secure boot, secure software updates, maintain operating system integrity and help protect data on the device as well as during communication with its paired iPhone and with the internet. Supported technologies include those listed in System Security (for example, KIP and SCIP) as well as Data Protection, keychain, and network technologies.

Secure pairing with iPhone

Apple Watch can be paired with only one iPhone at a time. When Apple Watch is unpaired, iPhone communicates instructions to erase all content and data from the watch.

Pairing Apple Watch with iPhone is secured using an out-of-band process to exchange public keys, followed by the Bluetooth Low Energy (BLE) link shared secret. Apple Watch displays an animated pattern, which is captured by the camera on iPhone. The pattern contains an encoded secret that's used for BLE 4.1 out-of-band pairing. Standard BLE Passkey Entry is used as a fallback pairing method, if necessary.

After the BLE session is established and encrypted using the highest security protocol available in the Bluetooth Core Specification, iPhone and Apple Watch exchange keys using either:

- A process adapted from Apple Identity Service (IDS) as described in the iMessage security overview.
- A key exchange using IKEv2/IPsec. The initial key exchange is authenticated using either the Bluetooth session key (for pairing scenarios) or the IDS keys (for operating system update scenarios). Each device generates a random public and private 256-bit Ed25519 key pair, and during the initial key exchange process, the public keys are exchanged.

The mechanism used for key exchange and encryption depends on which operating system versions are on the iPhone and Apple Watch. iPhone devices running iOS 13 or later when paired with an Apple Watch running watchOS 6 or later use only IKEv2/IPsec for key exchange and encryption.

After keys have been exchanged:

- The Bluetooth session key is discarded and all communications between iPhone and Apple Watch are encrypted using one of the methods listed above—with the encrypted Bluetooth, Wi-Fi, and cellular links providing a secondary encryption layer.
- (IKEv2/IPsec only) The keys are stored in the system keychain and used for authenticating future IKEv2/IPsec sessions between the devices. Further communication between these devices is encrypted and integrity protected using ChaCha20-Poly1305 (256-bit keys).

The Bluetooth Low Energy device address is rotated at 15-minute intervals to reduce the risk of local tracking of the device using the broadcast of a persistent identifier.

To support apps that need streaming data, encryption is provided using methods described in FaceTime security, using either the Apple Identity Service (IDS) provided by the paired iPhone or a direct internet connection.

Apple Watch implements hardware-encrypted storage and class-based protection of files and keychain items. Access-controlled keybags for keychain items are also used. Keys used to communicate between Apple Watch and iPhone are also secured using class-based protection. For more information, see Keybags for Data Protection.

Secure use of Wi-Fi, cellular, iCloud, and Gmail

When Apple Watch isn't within Bluetooth range, Wi-Fi or cellular can be used instead. Apple Watch automatically joins Wi-Fi networks that have been already been joined on the paired iPhone and whose credentials have synced to the Apple Watch while both devices were in range. This Auto-Join behavior can then be configured on a per-network basis in the Wi-Fi section of the Apple Watch Settings app. Wi-Fi networks that have never been joined before on either device can be manually joined in Wi-Fi section of the Apple Watch Settings app.

When Apple Watch and iPhone are out of range, Apple Watch connects directly to iCloud and Gmail servers to fetch Mail, as opposed to syncing Mail data with the paired iPhone over the internet. For Gmail accounts, the user is required to authenticate to Google in the Mail section of the Watch app on iPhone. The OAuth token received from Google is sent over to Apple Watch in encrypted format over Apple Identity Service (IDS) so it can be used to fetch Mail. This OAuth token is never used for connectivity with the Gmail server from the paired iPhone.

Locking and unlocking Apple Watch

If wrist detection is enabled, the device locks automatically shortly after it's removed from the user's wrist. If wrist detection is disabled, Control Center provides an option for locking Apple Watch. When Apple Watch is locked, Apple Pay can be used only by entering the watch's passcode. Wrist detection is turned off using the Apple Watch app on iPhone. This setting can also be enforced using a mobile device management (MDM) solution.

The paired iPhone can also unlock the watch, provided the watch is being worn. This is accomplished by establishing a connection authenticated by the keys established during pairing. iPhone sends the key, which the watch uses to unlock its Data Protection keys. The watch passcode isn't known to iPhone nor is it transmitted. This feature can be turned off using the Apple Watch app on iPhone.

Enabling Find My on the paired iPhone also allows the use of Activation Lock on Apple Watch. Activation Lock makes it harder for anyone to use or sell an Apple Watch that's been lost or stolen. Activation Lock requires the user's Apple ID and password to unpair, erase, or reactivate an Apple Watch.

Updating system software

Apple Watch can be configured for a system software update the same night. For more information on how the Apple Watch passcode gets stored and used during the update, see Keybags.

Auto Unlock with Apple Watch in macOS

Users with Apple Watch can use it to automatically unlock their Mac. Bluetooth Low Energy (BLE) and peer-to-peer Wi-Fi allow Apple Watch to securely unlock a Mac after ensuring proximity between the devices. This requires an iCloud account with two-factor authentication configured.

When enabling an Apple Watch to unlock a Mac, a secure link using Auto Unlock Identities is established. The Mac creates a random one-time-use unlock secret and transmits it to the Apple Watch over the link. The secret is stored on Apple Watch and can be accessed only when Apple Watch is unlocked. The unlock token isn't the user's password.

During an unlock operation, the Mac uses BLE to create a connection to the Apple Watch. A secure link is then established between the two devices using the shared keys used when it was first enabled. The Mac and Apple Watch then use peer-to-peer Wi-Fi and a secure key derived from the secure link to determine the distance between the two devices. If the devices are within range, the secure link is then used to transfer the preshared secret to unlock the Mac. After successful unlock, the Mac replaces the current unlock secret with a new one-time use unlock secret and transmits the new unlock secret to the Apple Watch over the link.

Approve with Apple Watch

When Auto Unlock with Apple Watch is enabled, the Apple Watch can be used in place or together with Touch ID to approve authorization and authentication prompts from:

- macOS and Apple apps that request authorization
- Third-party apps that request authentication
- Saved Safari passwords
- Secure Notes

Random number generation

Cryptographic pseudorandom number generators (CPRNGs) are an important building block for secure software. To this end, Apple provides a trusted software CPRNG running in the iOS, iPadOS, macOS, tvOS, and watchOS kernels. It's responsible for aggregating raw entropy from the system and providing secure random numbers to consumers in both the kernel and user space.

Entropy sources

The kernel CPRNG is seeded from multiple entropy sources during boot and over the lifetime of the device. These include (contingent on availability):

- The Secure Enclave hardware TRNG
- Timing-based jitter collected during boot
- Entropy collected from hardware interrupts
- A seed file used to persist entropy across boots
- Intel random instructions—for example, RDSEED and RDRAND (only on an Intel-based Mac)

The kernel CPRNG

The kernel CPRNG is a Fortuna-derived design targeting a 256-bit security level. It provides high-quality random numbers to user-space consumers using the following APIs:

- The `getentropy`(2) system call
- The random device (/dev/random)

The kernel CPRNG accepts user-supplied entropy through writes to the random device.

Apple Security Research Device

The Apple Security Research Device is a specially fused iPhone that allows security researchers to perform research on iOS without having to defeat or disable the platform security features of iPhone. With this device, a researcher can side-load content that runs with platform-equivalent permissions and thus perform research on a platform that more closely models that of production devices.

To ensure that user devices aren't affected by the security research device execution policy, the policy changes are implemented in a variant of iBoot and the Boot Kernel Collection. These fail to boot on user hardware. The research iBoot checks for a new fusing state and enters a panic loop if it's being run on nonresearch fused hardware.

The cryptex subsystem allows a researcher to load a personalized trust cache and a disk image containing corresponding content. A number of defense in-depth measures have been implemented to ensure that this subsystem doesn't allow execution on user devices:

- `launchd` won't load the `cryptexd` launchd property list if it's unable to detect the research fuse.
- `cryptexd` aborts if it doesn't detect the research fuse.

- The entitlement that grants `cryptexd` the ability to mount a disk image is honored only by the research kernel cache. The relevant code path isn't compiled into the release kernel cache.
- The signing server refuses to personalize a cryptex disk image for a device not on an explicit allow list.

To respect the privacy of the security researcher, only the measurements (for example, hashes) of the executables and the security research device identifiers are sent to Apple during personalization. Apple doesn't receive the content of the cryptex being loaded onto the device.

To avoid having a malicious party attempt to masquerade a research device as a user device to trick a target into using it for everyday usage, the security research device has the following differences:

- The security research device starts up only while charging. This can be using a lightning cable or a Qi-compatible charger. If the device isn't charging during startup, the device enters Recovery mode. If the user starts charging and restarts the device, it starts up as normal. As soon as XNU starts, the device doesn't need to be charging to continue operation.
- The words *Security Research Device* are displayed below the Apple logo during iBoot startup.
- The XNU kernel boots in verbose mode.
- The device is etched on the side with the message "Property of Apple. Confidential and Proprietary. Call +1 877 595 1125."

The following are additional measures that are implemented in software that appears after boot:

- The words *Security Research Device* are displayed during device setup.
- The words *Security Research Device* are displayed on the lock screen and in the Settings app.

The Security Research Device affords researchers the following abilities that a user device doesn't:

- Side-load executable code onto the device with arbitrary entitlements at the same permission level as Apple operating system components.
- Start services at startup.
- Persist content across restarts.

Encryption and Data Protection

Encryption and Data Protection overview

The secure boot chain, system security, and app security capabilities all help to verify that only trusted code and apps run on a device. Apple devices have additional encryption features to safeguard user data, even when other parts of the security infrastructure have been compromised (for example, if a device is lost or is running untrusted code). All of these features benefit both users and IT administrators, protecting personal and corporate information and providing methods for instant and complete remote wipe in the case of device theft or loss.

iOS and iPadOS devices used a file encryption methodology called *Data Protection*, whereas the data on an Intel-based Mac is protected with a volume encryption technology called *FileVault*. A Mac with Apple silicon uses a hybrid model that supports Data Protection, with two caveats: The lowest protection level Class (D) isn't supported, and the default level (Class C) uses a volume key and acts just like the FileVault on an Intel-based Mac. In all cases, key management hierarchies are rooted in the dedicated silicon of the Secure Enclave, and a dedicated AES Engine supports line-speed encryption and helps ensure that long-lived encryption keys aren't exposed to the kernel operating system or CPU (where they might be compromised). (An Intel-based Mac with a T1 or lacking a Secure Enclave doesn't use dedicated silicon to protect its FileVault encryption keys.)

Besides using Data Protection and FileVault to prevent unauthorized access to data, Apple *operating system kernels* enforce protection and security. The kernel uses access controls to sandbox apps (which restricts what data an app can access) and a mechanism called a *Data Vault* (which rather than restricting the calls an app can make, restricts access to the data of an app from all other requesting apps).

Passcodes and passwords

Passcodes in devices that support Data Protection

By setting up a device passcode or password, the user automatically enables Data Protection. iOS and iPadOS support six-digit, four-digit, and arbitrary-length alphanumeric passcodes. Besides unlocking the device, a passcode or password provides entropy for certain encryption keys. This means an attacker in possession of a device can't get access to data in specific protection classes without the passcode.

The passcode or password is entangled with the device's UID, so brute-force attempts must be performed on the device under attack. A large iteration count is used to make each attempt slower. The iteration count is calibrated so that one attempt takes approximately 80 milliseconds. In fact, it would take more than five and one-half years to try all combinations of a six-character alphanumeric passcode with lowercase letters and numbers.

The stronger the user passcode is, the stronger the encryption key becomes. And by using Touch ID and Face ID, the user can establish a much stronger passcode than would otherwise be practical. The stronger passcode increases the effective amount of entropy protecting the encryption keys used for Data Protection, without adversely affecting the user experience of unlocking a device multiple times throughout the day.

To further discourage brute-force passcode attacks, there are escalating time delays after the entry of an invalid passcode at the Lock Screen.

Delays between passcode attempts

Attempts	Delay enforced
1–4	None
5	1 minute
6	5 minutes
7–8	15 minutes
9	1 hour

If the Erase Data option is turned on (in Settings > Touch ID & Passcode), after 10 consecutive incorrect attempts to enter the passcode, all content and settings are removed from storage. Consecutive attempts of the same incorrect passcode don't count toward the limit. This setting is also available as an administrative policy through a mobile device management (MDM) solution that supports this feature and through Microsoft Exchange ActiveSync, and can be set to a lower threshold.

On devices with Secure Enclave, the delays are enforced by the Secure Enclave. If the device is restarted during a timed delay, the delay is still enforced, with the timer starting over for the current period.

Specifying longer passcodes

If a long password that contains only numbers is entered, a numeric keypad is displayed at the Lock Screen instead of the full keyboard. A longer numeric passcode may be easier to enter than a shorter alphanumeric passcode, while providing similar security.

Users can specify a longer alphanumeric passcode by selecting Custom Alphanumeric Code in the Passcode Options in Settings > Touch ID & Passcode or Face ID & Passcode.

Delays between passwords attempts in macOS

To prevent brute-force attacks, when Mac starts up, no more than 30 password attempts are allowed at the Login Window or using Target Disk Mode, and escalating time delays are imposed after incorrect attempts. The delays are enforced by the Secure Enclave. If Mac is restarted during a timed delay, the delay is still enforced, with the timer starting over for the current period.

To prevent malware from causing permanent data loss by trying to attack the user's password, these limits aren't enforced after the user has successfully logged into the Mac, but is reimposed after reboot. If the 30 attempts are exhausted, 10 more attempts are available after booting into recoveryOS. And if those are also exhausted, then 60 additional attempts are available for each FileVault recovery mechanism (iCloud recovery, FileVault recovery key, and institutional key), for a maximum of 180 additional attempts. Once those additional attempts are exhausted, the Secure Enclave no longer processes any requests to decrypt the volume or verify the password, and the data on the drive becomes unrecoverable.

To protect data in an enterprise setting, IT should define and enforce FileVault configuration policies using an MDM solution. Organizations have several options for managing encrypted volumes, including institutional recovery keys, personal recovery keys (that can optionally be stored with MDM for escrow), or a combination of both. Key rotation can also be set as a policy in MDM.

Delays between password attempts on a Mac with Apple silicon and those with the T2 chip

Attempts	Delay enforced
5	1 minute
6	5 minutes
7	15 minutes
8	15 minutes
9	1 hour
10	Disabled

In a Mac with the Apple T2 Security Chip, the password serves a similar function except that the key generated is used for FileVault encryption rather than Data Protection. macOS also offers additional password recovery options:

- iCloud recovery
- FileVault recovery
- FileVault institutional key

Data Protection

Data Protection overview

Apple uses a technology called Data Protection to protect data stored in flash storage on the devices that feature an Apple SoC such as iPhone, iPad, Apple Watch, Apple TV, and a Mac with Apple silicon. Data Protection allows the device to respond to common events such as incoming phone calls but also enables a high level of encryption for user data. Certain system apps (such as Messages, Mail, Calendar, Contacts, Photos) and Health data values use Data Protection by default. Third-party apps receive this protection automatically.

Implementation

Data Protection is implemented by constructing and managing a hierarchy of keys and builds on the hardware encryption technologies built into Apple devices. Data Protection is controlled on a per-file basis by assigning each file to a class; accessibility is determined according to whether the class keys have been unlocked. APFS (Apple File System) allows the file system to further subdivide the keys into a per-extent basis (where portions of a file can have different keys).

Every time a file on the data volume is created, Data Protection creates a new 256-bit key (the per-file key) and gives it to the hardware AES Engine, which uses the key to encrypt the file as it is written to flash storage. On A14 and M1 devices, the encryption uses AES-256 in XTS mode where the 256-bit per-file-key goes through a Key Derivation Function (NIST Special Publication 800-108) to derive a 256-bit tweak and a 256-bit cipher key. The hardware generations of A9 through A13, S5, and S6 use AES-128 in XTS mode where the 256-bit per file key was split to provide a 128-bit tweak and a 128-bit cipher key.

On a Mac with Apple silicon, Data Protection defaults to Class C (see Data Protection classes), but utilizes a volume key rather than a per-extent or per-file key—effectively re-creating the security model of FileVault for user data. Users must still opt-in to FileVault in order to receive the full protection of entangling the encryption key hierarchy with their password. Developers can also opt in to a higher protection class that uses a per-file or per-extent key.

Data Protection in Apple devices

On Apple devices with Data Protection, each file is protected with a unique per-file (or per-extent) key. The key, wrapped using the NIST AED key wrap algorithm, is further wrapped with one of several class keys, depending on how the file is meant to be accessed. The wrapped per-file key is then stored in the file's metadata.

Devices with APFS format may support cloning of files (zero-cost copies using copy-on-write technology). If a file is cloned, each half of the clone gets a new key to accept incoming writes so that new data is written to the media with a new key. Over time, the file may become composed of various extents (or fragments), each mapping to different keys. However, all of the extents that comprise a file are guarded by the same class key.

When a file is opened, its metadata is decrypted with the file system key, revealing the wrapped per-file key and a notation on which class protects it. The per-file (or per-extent) key is unwrapped with the class key and then supplied to the hardware AES Engine, which decrypts the file as it's read from flash storage. All wrapped file key handling occurs in the Secure Enclave; the file key is never directly exposed to the Application Processor. At startup, the Secure Enclave negotiates an ephemeral key with the AES Engine. When the Secure Enclave unwraps a file's keys, they're rewrapped with the ephemeral key and sent back to the Application Processor.

The metadata of all files in the data volume file system are encrypted with a random volume key, which is created when the operating system is first installed or when the device is wiped by a user. This key is encrypted and wrapped by a key wrapping key that is known only to the Secure Enclave for long-term storage. The key wrapping key changes every time a user erases their device. On A9 (and newer) SoCs, Secure Enclave relies upon entropy, backed by anti-replay systems, to achieve effaceability and to protect its key wrapping key, among other assets. For more information, see Secure nonvolatile storage.

Just like per-file or per-extent keys, the metadata key of the data volume is never directly exposed to the Application Processor; the Secure Enclave provides an ephemeral, per-boot version instead. When stored, the encrypted file system key is additionally wrapped by an "effaceable key" stored in Effaceable Storage or using a media key-wrapping key, protected by Secure Enclave anti-replay mechanism. This key doesn't provide additional confidentiality of data. Instead, it's designed to be quickly erased on demand (by the user with the "Erase All Content and Settings" option, or by a user or administrator issuing a remote wipe command from a mobile device management (MDM) solution, Microsoft Exchange ActiveSync, or iCloud). Erasing the key in this manner renders all files cryptographically inaccessible.

The contents of a file may be encrypted with one or more per-file (or per-extent) keys that are wrapped with a class key and stored in a file's metadata, which in turn is encrypted with the file system key. The class key is protected with the hardware UID and, for some classes, the user's passcode. This hierarchy provides both flexibility and performance. For example, changing a file's class only requires rewrapping its per-file key, and a change of passcode just rewraps the class key.

Data Protection classes

When a new file is created on devices supporting Data Protection, it's assigned a class by the app that creates it. Each class uses different policies to determine when the data is accessible. The basic classes and policies are described in the following sections. Apple silicon-based Macs do not support Class D: No Protection, and a security boundary is established around logging in and out (not locking or unlocking as on iPhone, iPad, and iPod touch).

Class	Protection type
Class A: Complete Protection	(NSFileProtectionComplete)
Class B: Protected Unless Open	(NSFileProtectionCompleteUnlessOpen)
Class C: Protected Until First User Authentication *Note:* macOS uses a volume key to recreate FileVault protection characteristics.	(NSFileProtectionCompleteUntilFirstUser Authentication)
Class D: No Protection *Note:* Not supported on macOS.	(NSFileProtectionNone)

Complete Protection

(NSFileProtectionComplete): The class key is protected with a key derived from the user passcode or password and the device UID. Shortly after the user locks a device (10 seconds, if the Require Password setting is Immediately), the decrypted class key is discarded, rendering all data in this class inaccessible until the user enters the passcode again or unlocks (logs in to) the device using Touch ID or Face ID.

In macOS, shortly after the last user is logged out, the decrypted class key is discarded, rendering all data in this class inaccessible until a user enters the passcode again or logs into the the device using Touch ID.

Protected Unless Open

(NSFileProtectionCompleteUnlessOpen): Some files may need to be written while the device is locked or the user is logged out. A good example of this is a mail attachment downloading in the background. This behavior is achieved by using asymmetric elliptic curve cryptography (ECDH over Curve25519). The usual per-file key is protected by a key derived using One-Pass Diffie-Hellman Key Agreement as described in NIST SP 800-56A.

The ephemeral public key for the Agreement is stored alongside the wrapped per-file key. The KDF is Concatenation Key Derivation Function (Approved Alternative 1) as described in 5.8.1 of NIST SP 800-56A. AlgorithmID is omitted. PartyUInfo and PartyVInfo are the ephemeral and static public keys, respectively. SHA256 is used as the hashing function. As soon as the file is closed, the per-file key is wiped from memory. To open the file again, the shared secret is re-created using the Protected Unless Open class's private key and the file's ephemeral public key, which are used to unwrap the per-file key that is then used to decrypt the file.

In macOS, the private part of NSFileProtectionCompleteUnlessOpen is accessible as long as any users on the system are logged in or are authenticated.

Protected Until First User Authentication

(NSFileProtectionCompleteUntilFirstUserAuthentication): This class behaves in the same way as Complete Protection, except that the decrypted class key isn't removed from memory when the device is locked or the user logged out. The protection in this class has similar properties to desktop full-volume encryption, and protects data from attacks that involve a reboot. This is the default class for all third-party app data not otherwise assigned to a Data Protection class.

In macOS, this class utilizes a volume key which is accessible as long as the volume is mounted, and acts just like FileVault.

No Protection

(NSFileProtectionNone): This class key is protected only with the UID, and is kept in Effaceable Storage. Since all the keys needed to decrypt files in this class are stored on the device, the encryption only affords the benefit of fast remote wipe. If a file isn't assigned a Data Protection class, it is still stored in encrypted form (as is all data on an iOS and iPadOS device).

This isn't supported in macOS.

Note: In macOS, for volumes that don't correspond to a booted operating system, all data protection classes are accessible as long as the volume is mounted. The default data protection class is *NSFileProtectionCompleteUntilFirstUserAuthentication*. Per-extent key functionality is available to both Rosetta 2 and native apps.

Keybags for Data Protection

The keys for both file and keychain Data Protection classes are collected and managed in keybags on iOS, iPadOS, watchOS, and tvOS. These operating systems use the following keybags: user, device, backup, escrow, and iCloud Backup.

User keybag

The user keybag is where the wrapped class keys used in normal operation of the device are stored. For example, when a passcode is entered, *NSFileProtectionComplete* is loaded from the user keybag and unwrapped. It is a binary property list (.plist) file stored in the No Protection class.

For devices with SoCs earlier than the A9, the .plist file contents are encrypted with a key held in Effaceable Storage. To give forward security to keybags, this key is wiped and regenerated each time a user changes their passcode.

For devices with the A9 or later SoCs, the .plist file contains a key that indicates that the keybag is stored in a locker protected by the Secure Enclave–controlled anti-replay nonce.

The Secure Enclave manages the user keybag and can be queried regarding a device's lock state. It reports that the device is unlocked only if all the class keys in the user keybag are accessible and have been unwrapped successfully.

Device keybag

The device keybag is used to store the wrapped class keys used for operations involving device-specific data. iOS and iPadOS devices configured for shared use sometimes need access to credentials before any user has logged in; therefore, a keybag that isn't protected by the user's passcode is required.

iOS and iPadOS don't support cryptographic separation of per-user file system content, which means the system uses class keys from the device keybag to wrap per-file keys. The keychain, however, uses class keys from the user keybag to protect items in the user keychain. In iOS and iPadOS devices configured for use by a single user (the default configuration), the device keybag and the user keybag are one and the same, and are protected by the user's passcode.

Backup keybag

The backup keybag is created when an encrypted backup is made by the Finder (macOS 10.15 or later) or iTunes (in macOS 10.14 or earlier) and stored on the computer to which the device is backed up. A new keybag is created with a new set of keys, and the backed-up data is reencrypted to these new keys. As explained previously, nonmigratory keychain items remain wrapped with the UID-derived key, allowing them to be restored to the device they were originally backed up from but rendering them inaccessible on a different device.

The keybag—protected with the password set—is run through 10 million iterations of the key derivation function PBKDF2. Despite this large iteration count, there's no tie to a specific device, and therefore a brute-force attack parallelized across many computers could theoretically be attempted on the backup keybag. This threat can be mitigated with a sufficiently strong password.

If a user chooses not to encrypt the backup, the files aren't encrypted regardless of their Data Protection class but the keychain remains protected with a UID-derived key. This is why keychain items migrate to a new device only if a backup password is set.

Escrow keybag

The escrow keybag is used for syncing with iTunes (in macOS 10.14 or earlier) or the Finder (macOS 10.15 or later) through USB and restored to factory default settings. syncing and mobile device management (MDM). This keybag allows iTunes or the Finder to back up and sync without requiring the user to enter a passcode, and it allows an MDM solution to remotely clear a user's passcode. It is stored on the computer that's used to sync with iTunes or the Finder, or on the MDM solution that remotely manages the device.

The escrow keybag improves the user experience during device syncing, which potentially requires access to all classes of data. When a passcode-locked device is first connected to the Finder or iTunes, the user is prompted to enter a passcode. The device then creates an escrow keybag containing the same class keys used on the device, protected by a newly generated key. The escrow keybag and the key protecting it are split between the device and the host or server, with the data stored on the device in the Protected Until First User Authentication class. This is why the device passcode must be entered before the user backs up with the Finder or iTunes for the first time after a reboot.

In the case of an over-the-air (OTA) software update, the user is prompted for their passcode when initiating the update. This is used to securely create a one-time unlock token, which unlocks the user keybag after the update. This token can't be generated without entering the user's passcode, and any previously generated token is invalidated if the user's passcode changed.

One-time unlock tokens are either for attended or unattended installation of a software update. They're encrypted with a key derived from the current value of a monotonic counter in the Secure Enclave, the UUID of the keybag, and the Secure Enclave UID.

On A9 (and later) SoCs, one-time Unlock token no longer relies on counters or Effaceable Storage. Instead, it's protected by Secure Enclave controlled anti-replay nonce.

The one-time unlock token for attended software updates expires after 20 minutes. In iOS 13 and iPadOS 13.1 or later, the token is stored in a locker protected by the Secure Enclave. Prior to iOS 13, this token was exported from the Secure Enclave and written to Effaceable Storage or was protected by the Secure Enclave anti-replay mechanism. A policy timer incremented the counter if the device hadn't rebooted within 20 minutes.

Unattended software updates occur when the system detects an update is available and when one of the following is true:

- Automatic updates are configured in iOS 12 or later.
- The user chooses Install Later when notified of the update.

After the user enters their passcode, a one-time unlock token is generated and can remain valid in Secure Enclave for up to 8 hours. If the update hasn't yet occurred, this one-time unlock token is destroyed on every lock and re-created on every subsequent unlock. Each unlock restarts the 8 hour window. After 8 hours a policy timer invalidates the one-time unlock token.

iCloud Backup keybag

The iCloud Backup keybag is similar to the backup keybag. All the class keys in this keybag are asymmetric (using Curve25519, like the Protected Unless Open Data Protection class). An asymmetric keybag is also used for the backup in the keychain recovery aspect of iCloud Keychain.

Protecting keys in alternate boot modes

Data Protection is designed to provide access to user data only after successful authentication, and only to the authorized user. Data protection classes are designed to support a variety of use cases, such as the ability to read and write some data even when a device is locked (but after first unlock). Additional steps are taken to protect access to user data during alternate boot modes such as those used for Device Firmware Update (DFU) mode, Recovery mode, Apple Diagnostics, or even during software update. These capabilities are based on a combination of hardware and software features, and have been expanded as Apple-designed silicon has evolved.

Feature	A10	A11, S3	A12, S4	A13, S5	A14, M1, S6
Recovery: All Data Protection Classes protected	✓	✓	✓	✓	✓
Alternate boots of DFU mode, Recovery, and software updates: Class A, B, and C data protected			✓	✓	✓

The Secure Enclave AES Engine is equipped with lockable software seed bits. When keys are created from the UID, these seed bits are included in the key derivation function to create additional key hierarchies. How the seed bit is used varies according to the system on chip:

- Starting with the Apple A10 and S3 SoCs, a seed bit is dedicated to distinguish keys protected by the user's passcode. The seed bit is set for keys that require the user's passcode (including Data Protection Class A, Class B, and Class C keys), and cleared for keys that don't require the user's passcode (including the file system metadata key and Class D keys).
- In iOS 13 or later and iPadOS 13.1 or later on devices with an A10 or later, all user data is rendered cryptographically inaccessible when devices are booted into Diagnostics mode. This is achieved by introducing an additional seed bit whose setting governs the ability to access the media key, which itself is needed to access the metadata (and therefore contents of) all files on the data volume encrypted with Data Protection. This protection encompasses files protected in all classes (A, B, C, and D), not just those that required the user's passcode.
- On A12 SoCs, the Secure Enclave Boot ROM locks the passcode seed bit if the Application Processor has entered Device Firmware Upgrade (DFU) mode or Recovery mode. When the passcode seed bit is locked, no operation to change it is allowed, preventing access to data protected with the user's passcode.

Restoring a device after it enters DFU mode returns it to a known good state with the certainty that only unmodified Apple-signed code is present. DFU mode can be entered manually.

See the following Apple Support articles on how to place a device in DFU mode:

Device	Article
iPhone, iPad, iPod touch	If you forgot the passcode on your iPhone, or your iPhone is disabled
Apple TV	Restore your Apple TV
A Mac with Apple silicon	Revive or restore a Mac with Apple silicon

Protecting user data in the face of attack

Attackers attempting to extract user data often try a number of techniques: extracting the encrypted data to another medium for brute-force attack, manipulating the operating system version, or otherwise changing or weakening the security policy of the device to facilitate attack. Attacking data on a device often requires communicating with the device using physical interfaces like Lightning or USB. Apple devices include features to help prevent such attacks.

Apple devices support a technology called *Sealed Key Protection (SKP)* that works to ensure cryptographic material is rendered unavailable off device, or if manipulations are made to operating system version or security settings without appropriate user authorization. This feature is *not* provided by the Secure Enclave, instead it is supported by hardware registers that exist at a lower layer in order to provide an additional layer of protection to the keys necessary to decrypt user data independent of the Secure Enclave.

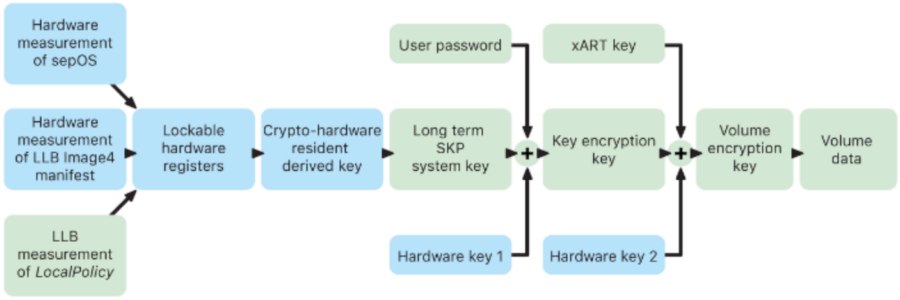
Note: SKP is available only on devices with an Apple-designed SoC.

Feature	A10	A11, S3	A12, S4	A13, S5	A14, M1, S6
Sealed Key Protection	✓	✓	✓	✓	✓

iPhone and iPad can also be configured to only activate data connections in conditions more likely to indicate the device is still under the physical control of the authorized owner.

Sealed Key Protection (SKP)

On Apple devices that support Data Protection, the key encryption key (KEK) is protected (or sealed) with measurements of the software on the system, as well as being tied to the UID available only from the Secure Enclave. On a Mac with Apple silicon, the protection of the KEK is further strengthened by incorporating information about security policy on the system, because macOS supports critical security policy changes (for example, disabling secure boot or SIP) that are unsupported on other platforms. On a Mac with Apple silicon, this protection encompass FileVault keys, because FileVault is implemented using Data Protection (Class C).



The Sealed Key Protection process.

The Secure Enclave Boot Monitor captures the measurement of the Secure Enclave OS that is loaded. When the Application Processor Boot ROM measures the Image4 manifest attached to LLB, that manifest contains a measurement of all other system-paired firmware that is loaded as well. The LocalPolicy contains the core security configurations for the macOS which are loaded. The LocalPolicy also contains the `nsih` field which is a hash of the macOS Image4 manifest. The macOS Image4 manifest contains measurements of all of the macOS-paired firmware and core macOS boot objects such as the Boot Kernel Collection or signed system volume (SSV) root hash.

If an attacker is able to unexpectedly change any of the above measured firmware, software, or security configuration components, it modifies the measurements stored in the hardware registers. The modification of the measurements cause the crypto-hardware-derived system measurement root key (SMRK) to derive to a different value, effectively breaking the seal on the key hierarchy. That causes the system measurement device key (SMDK) to be inaccessible, which in turn causes the KEK, and thus the data, to be inaccessible.

However, the system must accommodate legitimate software updates that change the firmware measurements and the `nsih` field in the LocalPolicy to point at new macOS measurements. In other systems that attempt to incorporate firmware measurements but that don't have a known good source of truth, the user is required to disable the security, update firmware, and then reenable so that a new measurement baseline can be captured. This significantly increases the risk that the attacker could tamper with firmware during a software update. The system is helped by the fact that the Image4 manifest contains all the measurements needed. The hardware that decrypts the SMDK with the SMRK when the measurements match during a normal boot can also encrypt the SMDK to to a proposed future SMRK. By specifying the measurements that are expected after a software update, the hardware can encrypt an SMDK, which is accessible in a current operating system, so that it remains accessible in a future operating system. Similarly, when a customer legitimately changes their security settings in the LocalPolicy, the SMDK must be encrypted to the future SMRK based on the measurement for the LocalPolicy, which LLB computes on the next restart.

Activating data connections securely in iOS and iPadOS

On iOS or iPadOS devices, if no data connection has been established recently, users must use Touch ID, Face ID, or a passcode to activate data connections through a Lightning, USB, or Smart Connector interface. This limits the attack surface against physically connected devices such as malicious chargers while still enabling usage of other accessories within reasonable time constraints. If more than an hour has passed since the iOS or iPadOS device has locked or since an accessory's data connection has been terminated, the device won't allow any new data connections to be established until the device is unlocked. During this hour period, only data connections from accessories that have been previously connected to the device while in an unlocked state will be allowed. These accessories are remembered for 30 days after the last time they were connected. Attempts by an unknown accessory to open a data connection during this period will disable all accessory data connections over Lighting, USB, and Smart Connector until the device is unlocked again. This hour period:

- Ensures that frequent users of connections to a Mac or PC, to accessories, or wired to CarPlay won't need to input their passcodes every time they attach their device
- Is necessary because the accessory ecosystem doesn't provide a cryptographically reliable way to identify accessories before establishing a data connection

In addition, if it's been more than 3 days since a data connection has been established with an accessory, the device will disallow new data connections immediately after it locks. This is to increase protection for users that don't often make use of such accessories. Data connections over Lightning, USB, and Smart Connector are also disabled whenever the device is in a state where it requires a passcode to reenable biometric authentication.

The user can choose to reenable always-on data connections in Settings (setting up some assistive devices does this automatically).

Role of Apple File System

Apple File System (APFS) is a proprietary file system that was designed with encryption in mind. APFS works across all Apple's platforms—for iPhone, iPad, iPod touch, Mac, Apple TV, and Apple Watch. Optimized for Flash/SSD storage, it features strong encryption, copy-on-write metadata, space sharing, cloning for files and directories, snapshots, fast directory sizing, atomic safe-save primitives, and improved file system fundamentals, as well as a unique copy-on-write design that uses I/O coalescing to deliver maximum performance while ensuring data reliability.

Space sharing

APFS allocates storage space on demand. When a single APFS container has multiple volumes, the container's free space is shared and can be allocated to any of the individual volumes as needed. Each volume uses only part of the overall container, so the available space is the total size of the container, minus the space used in all volumes in the container.

Multiple volumes

In macOS 10.15 or later, an APFS container used to start up the Mac must contain at least five volumes, the first three of which are hidden from the user:

- *Preboot volume:* Contains data needed for booting each system volume in the container
- *VM volume:* Used by macOS for swap file storage
- *Recovery volume:* Contains recoveryOS
- *System volume:* Contains the following:
  - All the necessary files to start up the Mac
  - All apps installed natively by macOS (apps that used to reside in the /Applications folder now reside in /System/Applications)

  Note: By default, no process can write to the System volume, even Apple system processes.
- *Data volume:* Contains data that is subject to change, such as:
  - Any data inside the user's folder, including photos, music, videos, and documents
  - Apps the user installed, including AppleScript and Automator applications
  - Custom frameworks and daemons installed by the user, organization, or third-party apps
  - Other locations owned and writable by the user, as /Applications, /Library, /Users, /Volumes, /usr/local, /private, /var, and /tmp

A data volume is created for each additional system volume. The preboot, VM, and recovery volumes are all shared and not duplicated.

In macOS 11, the system volume is captured in a snapshot. The operating system boots from a snapshot of the system volume, not just from a read-only mount of the mutable system volume.

In iOS and iPadOS, storage is divided into at least two APFS volumes:

- System volume
- Data volume

Keychain data protection

Many apps need to handle passwords and other short but sensitive bits of data, such as keys and login tokens. The keychain provides a secure way to store these items. The various Apple operating systems use differing mechanisms to enforce the guarantees associated with the different keychain protection classes. In macOS (including a Mac with Apple silicon), Data Protection is not used directly to enforce these guarantees.

Overview

Keychain items are encrypted using two different AES-256-GCM keys: a table key (metadata) and a per-row key (secret key). Keychain metadata (all attributes other than kSecValue) is encrypted with the metadata key to speed searches, and the secret value (kSecValueData) is encrypted with the secret key. The metadata key is protected by the Secure Enclave but is cached in the Application Processor to allow fast queries of the keychain. The secret key always requires a round trip through the Secure Enclave.

The keychain is implemented as a SQLite database, stored on the file system. There is only one database, and the `securityd` daemon determines which keychain items each process or app can access. Keychain Access APIs result in calls to the daemon, which queries the app's "Keychain-access-groups," "application-identifier," and "application-group" entitlements. Rather than limiting access to a single process, access groups allow keychain items to be shared between apps.

Keychain items can be shared only between apps from the same developer. To share keychain items, third-party apps to use access groups with a prefix allocated to them through the Apple Developer Program in their application groups. The prefix requirement and application group uniqueness are enforced through code signing, provisioning profiles, and the Apple Developer Program.

Keychain data is protected using a class structure similar to the one used in file Data Protection. These classes have behaviors equivalent to file Data Protection classes but use distinct keys and functions.

Availability	File data protection	Keychain data protection
When unlocked	NSFileProtectionComplete	kSecAttrAccessibleWhenUnlocked
While locked	NSFileProtectionCompleteUnlessOpen	N/A
After first unlock	NSFileProtectionCompleteUntilFirstUserAuthentication	kSecAttrAccessibleAfterFirstUnlock
Always	NSFileProtectionNone	kSecAttrAccessibleAlways
Passcode enabled	N/A	kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly

Apps that use background refresh services can use *kSecAttrAccessibleAfterFirstUnlock* for keychain items that need to be accessed during background updates.

The class *kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly* behaves the same as *kSecAttrAccessibleWhenUnlocked*; however, it's available only when the device is configured with a passcode. This class exists only in the system keybag; they:

- Don't sync to iCloud Keychain
- Aren't backed up
- Aren't included in escrow keybags

If the passcode is removed or reset, the items are rendered useless by discarding the class keys.

Other keychain classes have a "This device only" counterpart, which is always protected with the UID when being copied from the device during a backup, rendering it useless if restored to a different device. Apple has carefully balanced security and usability by choosing keychain classes that depend on the type of information being secured and when it's needed by iOS and iPadOS. For example, a VPN certificate must always be available so the device keeps a continuous connection, but it's classified as "nonmigratory," so it can't be moved to another device.

Keychain data class protections

The class protections listed below are enforced for keychain items.

Item	Accessible
Wi-Fi passwords	After first unlock
Mail accounts	After first unlock
Microsoft Exchange ActiveSync accounts	After first unlock
VPN passwords	After first unlock
LDAP, CalDAV, CardDAV	After first unlock
Social network account tokens	After first unlock
Handoff advertisement encryption keys	After first unlock
iCloud token	After first unlock
Home sharing password	When unlocked
Safari passwords	When unlocked
Safari bookmarks	When unlocked
Finder/iTunes backup	When unlocked, nonmigratory
VPN certificates	Always, nonmigratory
Bluetooth® keys	Always, nonmigratory
Apple Push Notification service (APNs) token	Always, nonmigratory
iCloud certificates and private key	Always, nonmigratory
iMessage keys	Always, nonmigratory
Certificates and private keys installed by a configuration profile	Always, nonmigratory
SIM PIN	Always, nonmigratory
Find My token	Always
Voicemail	Always

Keychain access control

Keychains can use access control lists (ACLs) to set policies for accessibility and authentication requirements. Items can establish conditions that require user presence by specifying that they can't be accessed unless authenticated using Touch ID, Face ID, or by entering the device's passcode or password. Access to items can also be limited by specifying that Touch ID or Face ID enrollment hasn't changed since the item was added. This limitation helps prevent an attacker from adding their own fingerprint to access a keychain item. ACLs are evaluated inside the Secure Enclave and are released to the kernel only if their specified constraints are met.

Keychain architecture in macOS

macOS also provides access to the keychain to conveniently and securely stores user names and passwords, digital identities, encryption keys, and secure notes. It can be accessed by opening the Keychain Access app in /Applications/Utilities/. Using a keychain eliminates the requirement to enter—or even remember—the credentials for each resource. An initial default keychain is created for each Mac user, though users can create other keychains for specific purposes.

In addition to relying on user keychains, macOS relies on a number of system-level keychains that maintain authentication assets that aren't user specific, such as network credentials and public key infrastructure (PKI) identities. One of these keychains, System Roots, is immutable and stores internet PKI root certificate authority (CA) certificates to facilitate common tasks like online banking and e-commerce. The user can similarly deploy internally provisioned CA certificates to managed Mac computers to help validate internal sites and services.

FileVault

Volume encryption with FileVault in macOS

Mac computers offer FileVault, a built-in encryption capability, to secure all data at rest. FileVault uses the AES-XTS data encryption algorithm to protect full volumes on internal and removable storage devices.

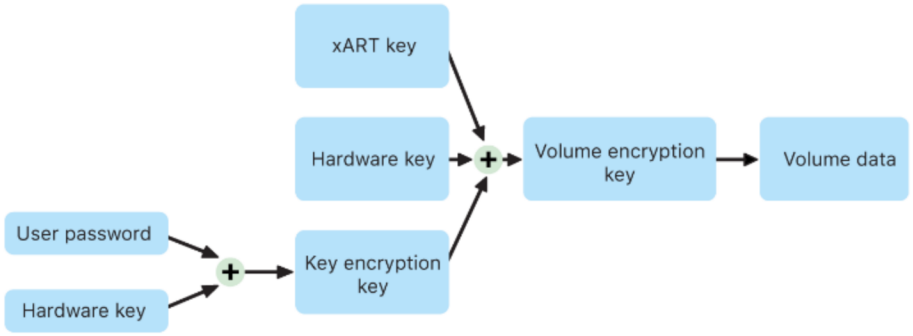
FileVault on a Mac with Apple silicon is implemented using Data Protection Class C with a volume key. On a Mac with the Apple T2 Security Chip as well as a Mac with Apple silicon, encrypted internal storage devices directly connected to the Secure Enclave leverage its hardware security capabilities as well as that of the AES engine. After a user turns on FileVault on a Mac, their credentials are required during the boot process.

Internal storage with FileVault turned on

Without valid login credentials or a cryptographic recovery key, the internal APFS volumes remain encrypted and are protected from unauthorized access even if the physical storage device is removed and connected to another computer. In macOS 10.15, this includes both the system volume and the data volume. Starting in macOS 11, the system volume is protected by the signed system volume (SSV) feature, but the data volume remains protected by encryption. Internal volume encryption on a Mac with Apple silicon as well as those with the T2 chip is implemented by constructing and managing a hierarchy of keys, and builds on the hardware encryption technologies built into the chip. This hierarchy of keys is designed to simultaneously achieve four goals:

- Require the user's password for decryption
- Protect the system from a brute-force attack directly against storage media removed from Mac
- Provide a swift and secure method for wiping content via deletion of necessary cryptographic material

- Enable users to change their password (and in turn the cryptographic keys used to protect their files) without requiring reencryption of the entire volume

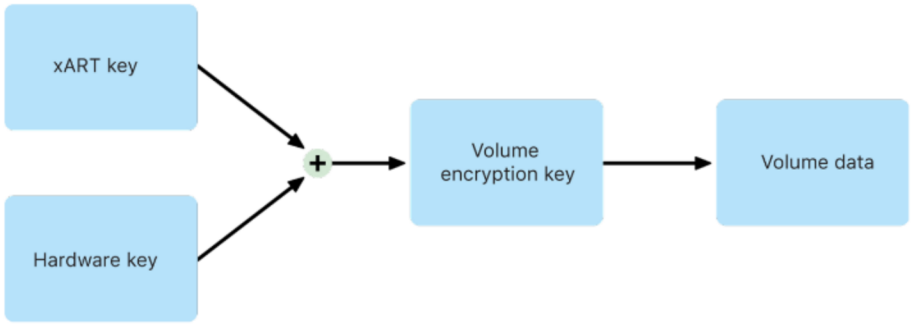


Internal volume encryption when FileVault is turned on in macOS.

On a Mac with Apple silicon and those with the T2 chip, all FileVault key handling occurs in the Secure Enclave; encryption keys are never directly exposed to the Intel CPU. All APFS volumes are created with a volume encryption key by default. Volume and metadata contents are encrypted with this volume encryption key, which is wrapped with the class key. The class key is protected by a combination of the user's password and the hardware UID when FileVault is turned on.

Internal storage with FileVault turned off

If FileVault isn't turned on in a Mac with Apple silicon or a Mac with the T2 chip during the initial Setup Assistant process, the volume is still encrypted but the volume encryption key is protected only by the hardware UID in the Secure Enclave.



Internal volume encryption when FileVault is turned off in macOS.

If FileVault is turned on later—a process that is immediate since the data was already encrypted—an anti-replay mechanism prevents the old key (based on hardware UID only) from being used to decrypt the volume. The volume is then protected by a combination of the user password with the hardware UID as previously described.

Deleting FileVault volumes

When deleting a volume, its volume encryption key is securely deleted by the Secure Enclave. This prevents future access with this key even by the Secure Enclave. In addition, all volume encryption keys are wrapped with a media key. The media key doesn't provide additional confidentiality of data, but instead is designed to enable swift and secure deletion of data because without it, decryption is impossible.

On a Mac with Apple silicon and those with the T2 chip, the media key is guaranteed to be erased by the Secure Enclave supported technology—for example by remote MDM commands. Erasing the media key in this manner renders the volume cryptographically inaccessible.

Removable storage devices

Encryption of removable storage devices doesn't utilize the security capabilities of the Secure Enclave, and its encryption is performed in the same manner as an Intel-based Mac without the T2 chip.

Managing FileVault in macOS

Using Secure Token

Apple File System (APFS) in macOS 10.13 or later changes how FileVault encryption keys are generated. In previous versions of macOS on CoreStorage volumes, the keys used in the FileVault encryption process were created when a user or organization turned on FileVault on a Mac. In macOS on APFS volumes, the keys are generated either during user creation, setting the first user's password, or during the first login by a user of the Mac. This implementation of the encryption keys, when they're generated, and how they're stored are all part of a feature known as *Secure Token*. Specifically, a secure token is a wrapped version of a key encryption key (KEK) protected by a user's password.

When deploying FileVault on APFS, the user can continue to:

- Use existing tools and processes, such as a personal recovery key (PRK) that can be stored with a mobile device management (MDM) solution for escrow
- Create and use an institutional recovery key (IRK)
- Defer enablement of FileVault until a user logs in to or out of the Mac

In macOS 11, setting the initial password for the very first user on the Mac results in that user being granted a secure token. In some workflows, that may not be the desired behavior, as previously, granting the first secure token would have required the user account to log in. To prevent this from happening, add `;DisabledTags;SecureToken` to the programmatically created user's `AuthenticationAuthority` attribute prior to setting the user's password, as shown below:

```
sudo dscl . append /Users/<user name> AuthenticationAuthority
";DisabledTags;SecureToken"
```

Using Bootstrap Token

macOS 10.15 introduced a new feature—Bootstrap Token—to help with granting a secure token to both mobile accounts and the optional device enrollment-created administrator account ("managed administrator"). In macOS 11, the Bootstrap Token can grant a secure token to any user logging in to a Mac computer, including local user accounts. Using the Bootstrap Token feature of macOS 10.15 or later requires:

- Mac enrollment in MDM using Apple School Manager or Apple Business Manager, which makes the Mac supervised
- MDM vendor support

In macOS 10.15.4 or later, a Bootstrap Token is generated and escrowed to MDM on the first login by any user who is Secure Token–enabled if the MDM solution supports the feature. A bootstrap token can also be generated and escrowed to MDM using the `profiles` command-line tool, if needed.

In macOS 11, the bootstrap token may also be used for more than just granting secure token to user accounts. On a Mac with Apple silicon, the bootstrap token, if available, can be used to authorize the installation of both kernel extensions and software updates when managed using MDM.

How Apple protects users' personal data

Protecting app access to user data

In addition to encrypting data at rest, Apple devices help prevent apps from accessing a user's personal information without permission using various technologies including Data Vault. In Settings in iOS and iPadOS, or System Preferences in macOS, users can see which apps they have permitted to access certain information as well as grant or revoke any future access. Access is enforced in the following:

- *iOS, iPadOS, and macOS:* Calendars, Camera, Contacts, Microphone, Photos, Reminders, Speech recognition
- *iOS and iPadOS:* Bluetooth, Home, Media, Media apps and Apple Music, Motion and fitness
- *iOS and watchOS:* Health
- *macOS:* Input monitoring (for example, keyboard strokes), Prompt, Screen recording (for example, static screen shots and video), System Preferences

In iOS 13.4 or later and (iPadOS 13.4) or later, all third-party apps automatically have their data protected in a Data Vault. Data Vault helps protect against unauthorized access to the data even from processes that aren't themselves sandboxed.

If the user signs in to iCloud, apps in iOS and iPadOS are granted access by default to iCloud Drive. Users may control each app's access under iCloud in Settings. iOS and iPadOS also provide restrictions that prevent data movement between apps and accounts installed by a mobile device management (MDM) solution and those installed by the user.

Protecting access to user's health data

HealthKit provides a central repository for health and fitness data on iPhone and Apple Watch. HealthKit also works directly with health and fitness devices, such as compatible Bluetooth Low Energy (BLE) heart rate monitors and the motion coprocessor built into many iOS devices. All HealthKit interaction with health and fitness apps, healthcare institutions, and health and fitness devices require permission of the user. This data is stored in the Data Protection class Protected Unless Open. Access to the data is relinquished 10 minutes after the device locks, and data becomes accessible the next time user enters their passcode or uses Touch ID or Face ID to unlock the device.

Collecting and storing health and fitness data

HealthKit also collects and stores management data, such as access permissions for apps, names of devices connected to HealthKit, and scheduling information used to launch apps when new data is available. This data is stored in the Data Protection class Protected Until First User Authentication. Temporary journal files store health records that are generated when the device is locked, such as when the user is exercising. These are stored in the Data Protection class Protected Unless Open. When the device is unlocked, the temporary journal files are imported into the primary health databases, then deleted when the merge is completed.

Health data can be stored in iCloud. End-to-end encryption for Health data requires iOS 12 or later and two-factor authentication. Otherwise, the user's data is still encrypted in storage and transmission but isn't encrypted end-to-end. After the user turns on two-factor authentication and updates to iOS 12 or later, the user's health data is migrated to end-to-end encryption.

If the user backs up their device using iTunes (in macOS 10.14 or earlier) or the Finder (macOS 10.15 or later), health data is stored only if the backup is encrypted.

Clinical health records

Users can sign in to supported health systems within the Health app to obtain a copy of their clinical health records. When connecting a user to a health system, the user authenticates using OAuth 2 client credentials. After connecting, clinical health record data is downloaded directly from the health institution using a TLS 1.3 protected connection. Once downloaded, clinical health records are securely stored alongside other health data.

Health data integrity

Data stored in the database includes metadata to track the provenance of each data record. This metadata includes an app identifier that identifies which app stored the record. Additionally, an optional metadata item can contain a digitally signed copy of the record. This is intended to provide data integrity for records generated by a trusted device. The format used for the digital signature is the Cryptographic Message Syntax (CMS) specified in RFC 5652.

Health data access by third-party apps

Access to the HealthKit API is controlled with entitlements, and apps must conform to restrictions about how the data is used. For example, apps aren't allowed to use health data for advertising. Apps are also required to provide users with a privacy policy that details its use of health data.

Access to health data by apps is controlled by the user's Privacy settings. Users are asked to grant access when apps request access to health data, similar to Contacts, Photos, and other iOS data sources. However, with health data, apps are granted separate access for reading and writing data, as well as separate access for each type of health data. Users can view, and revoke, permissions they've granted for accessing health data under Settings > Health > Data Access & Devices.

If granted permission to write data, apps can also read the data they write. If granted the permission to read data, they can read data written by all sources. However, apps can't determine access granted to other apps. In addition, apps can't conclusively tell if they have been granted read access to health data. When an app doesn't have read access, all queries return no data—the same response that an empty database would return. This prevents apps from inferring the user's health status by learning which types of data the user is tracking.

Medical ID for users

The Health app gives users the option of filling out a Medical ID form with information that could be important during a medical emergency. The information is entered or updated manually and isn't synced with the information in the health databases.

The Medical ID information is viewed by tapping the Emergency button on the Lock Screen. The information is stored on the device using the Data Protection class No Protection so that it's accessible without having to enter the device passcode. Medical ID is an optional feature that enables users to decide how to balance both safety and privacy concerns. This data is backed up in iCloud Backup in iOS 13 or earlier. In iOS 14, Medical ID is synced between devices using CloudKit and has the same encryption characteristics as the rest of health data.

Digital signing and encryption

Access control lists

Keychain data is partitioned and protected with access control lists (ACLs). As a result, credentials stored by third-party apps can't be accessed by apps with different identities unless the user explicitly approves them. This protection provides a mechanism for securing authentication credentials in Apple devices across a range of apps and services within the organization.

Mail

In the Mail app, users can send messages that are digitally signed and encrypted. Mail automatically discovers appropriate RFC 5322 case-sensitive email address subject or subject alternative names on digital signing and encryption certificates on attached Personal Identification Verification (PIV) tokens in compatible smart cards. If a configured email account matches an email address on a digital signing or encryption certificate on an attached PIV token, Mail automatically displays the signing button in the toolbar of a new message window. If Mail has the recipient's email encryption certificate or can discover it in the Microsoft Exchange global address list (GAL), an unlocked icon appears in the new message toolbar. A locked lock icon indicates the message will be sent encrypted with the recipient's public key.

Per-message S/MIME

iOS, iPadOS, and macOS support per-message S/MIME. This means that S/MIME users can choose to always sign and encrypt messages by default or to selectively sign and encrypt individual messages.

Identities used with S/MIME can be delivered to Apple devices using a configuration profile, a mobile device management (MDM) solution, the Simple Certificate Enrollment Protocol (SCEP), or Microsoft Active Directory Certificate Authority.

Smart cards

macOS 10.12 or later includes native support for PIV cards. These cards are widely used in commercial and government organizations for two-factor authentication, digital signing, and encryption.

Smart cards include one or more digital identities that have a pair of public and private keys and an associated certificate. Unlocking a smart card with the personal identification number (PIN) provides access to the private keys used for authentication, encryption, and signing operations. The certificate determines what a key can be used for, what attributes are associated with it, and whether it's validated (signed) by a certificate authority (CA) certificate.

Smart cards can be used for two-factor authentication. The two factors needed to unlock a card are "something the user has" (the card) and "something the user knows" (the PIN). macOS 10.12 or later also has native support for smart card Login Window authentication and client certificate authentication to websites on Safari. It also supports Kerberos authentication using key pairs (PKINIT) for single sign-on to Kerberos-supported services. To learn more about smart cards and macOS, see Intro to smart card integration in the *Deployment Reference for Mac*.

Encrypted disk images

In macOS, encrypted disk images serve as secure containers in which users can store or transfer sensitive documents and other files. Encrypted disk images are created using Disk Utility, located in /Applications/Utilities/. Disk images can be encrypted using either 128-bit or 256-bit AES encryption. Because a mounted disk image is treated as a local volume connected to a Mac, users can copy, move, and open files and folders stored in it. As with FileVault, the contents of a disk image are encrypted and decrypted in real time. With encrypted disk images, users can safely exchange documents, files, and folders by saving an encrypted disk image to removable media, sending it as a mail message attachment, or storing it on a remote server. For more information on encrypted disk images, see the Disk Utility User Guide.

App security

App security overview

Today, apps are among the most critical elements of a security architecture. Even as apps provide amazing productivity benefits for users, they also have the potential to negatively impact system security, stability, and user data if they're not handled properly.

Because of this, Apple provides layers of protection to help ensure that apps are free of known malware and haven't been tampered with. Additional protections enforce that access from apps to user data is carefully mediated. These security controls provide a stable, secure platform for apps, enabling thousands of developers to deliver hundreds of thousands of apps for iOS, iPadOS, and macOS—all without impacting system integrity. And users can access these apps on their Apple devices without undue fear of viruses, malware, or unauthorized attacks.

On iPhone, iPad, and iPod touch, all apps are obtained from the App Store—and all apps are sandboxed—to provide the tightest controls.

On Mac, many apps are obtained from the App Store, but Mac users also download and use apps from the internet. To safely support internet downloading, macOS layers additional controls. First, by default in macOS 10.15 or later, all Mac apps need to be notarized by Apple to launch. This requirement helps to ensure that these apps are free of known malware without requiring that the apps be provided through the App Store. In addition, macOS includes state-of-the-art antivirus protection to block—and if necessary remove—malware.

As an additional control across platforms, sandboxing helps protect user data from unauthorized access by apps. And in macOS, data in critical areas is itself protected—which helps ensure that users remain in control of access to files in Desktop, Documents, Downloads, and other areas from all apps, whether the apps attempting access are themselves sandboxed or not.

Native capability	Third-party equivalent
Plug-in unapproved list, Safari extension unapproved list	Virus/Malware definitions
File quarantine	Virus/Malware definitions
XProtect/YARA signatures	Virus/Malware definitions
MRT (Malware Removal Tool)	Endpoint protection
Gatekeeper	Endpoint protection; enforces code signing on apps to ensure only trusted software runs.
eficheck (Necessary for a Mac without an Apple T2 Security Chip)	Endpoint protection; rootkit detection
Application firewall	Endpoint protection; firewalling
Packet Filter (pf)	Firewall solutions
System Integrity Protection	Built into macOS
Mandatory Access Controls	Built into macOS
Kext exclude list	Built into macOS
Mandatory app code signing	Built into macOS
App notarization	Built into macOS

App security in iOS and iPadOS

App security overview for iOS and iPadOS

Unlike other mobile platforms, iOS and iPadOS don't allow users to install potentially malicious unsigned apps from websites or run untrusted apps. At runtime, code signature checks that all executable memory pages are made as they are loaded to ensure that an app hasn't been modified since it was installed or last updated.

After an app is verified to be from an approved source, iOS and iPadOS enforce security measures designed to prevent it from compromising other apps or the rest of the system.

App code signing process in iOS and iPadOS

Mandatory code signing

After the iOS or iPadOS kernel has started, it controls which user processes and apps can be run. To ensure that all apps come from a known and approved source and haven't been tampered with, iOS and iPadOS require that all executable code be signed using an Apple-issued certificate. Apps provided with the device, like Mail and Safari, are signed by Apple. Third-party apps must also be validated and signed using an Apple-issued certificate. Mandatory code signing extends the concept of chain of trust from the operating system to apps and prevents third-party apps from loading unsigned code resources or using self-modifying code.

How developers sign their apps

Developers can sign their apps through certificate validation (through the Apple Developer Program). They can also embed frameworks inside their apps and have that code validated with an Apple-issued certificate (through a team identifier string).

- *Certificate validation:* To develop and install apps in iOS or iPadOS devices, developers must register with Apple and join the Apple Developer Program. The real-world identity of each developer, whether an individual or a business, is verified by Apple before their certificate is issued. This certificate enables developers to sign apps and submit them to the App Store for distribution. As a result, all apps in the App Store have been submitted by an identifiable person or organization, serving as a deterrent to the creation of malicious apps. They have also been reviewed by Apple to ensure they generally operate as described and don't contain obvious bugs or other notable problems. In addition to the technology already discussed, this curation process gives users confidence in the quality of the apps they buy.
- *Code signature validation:* iOS and iPadOS allow developers to embed frameworks inside of their apps, which can be used by the app itself or by extensions embedded within the app. To protect the system and other apps from loading third-party code inside of their address space, the system performs a code signature validation of all the dynamic libraries that a process links against at launch time. This verification is accomplished through the team identifier (Team ID), which is extracted from an Apple-issued certificate. A team identifier is a 10-character alphanumeric string—for example, 1A2B3C4D5F. A program may link against any platform library that ships with the system or any library with the same team identifier in its code signature as the main executable. Since the executables shipping as part of the system don't have a team identifier, they can only link against libraries that ship with the system itself.

Verifying enterprise apps

Businesses also have the ability to write in-house apps for use within their organization and distribute them to their employees. Businesses and organizations can apply to the Apple Developer Enterprise Program (ADEP) with a D-U-N-S number. Apple approves applicants after verifying their identity and eligibility. After an organization becomes a member of ADEP, it can register to obtain a provisioning profile that permits in-house apps to run on devices it authorizes.

Users must have the provisioning profile installed to run the in-house apps. This ensures that only the organization's intended users are able to load the apps onto their iOS and iPadOS devices. Apps installed through mobile device management (MDM) are implicitly trusted because the relationship between the organization and the device is already established. Otherwise, users have to approve the app's provisioning profile in Settings. Organizations can restrict users from approving apps from unknown developers. On first launch of any enterprise app, the device must receive positive confirmation from Apple that the app is allowed to run.

Security of runtime process in iOS and iPadOS

Sandboxing

All third-party apps are “sandboxed,” so they are restricted from accessing files stored by other apps or from making changes to the device. Sandboxing prevents apps from gathering or modifying information stored by other apps. Each app has a unique home directory for its files, which is randomly assigned when the app is installed. If a third-party app needs to access information other than its own, it does so only by using services explicitly provided by iOS and iPadOS.

System files and resources are also shielded from the users’ apps. Most iOS and iPadOS system files and resources run as the nonprivileged user “mobile,” as do all third-party apps. The entire operating system partition is mounted as read-only. Unnecessary tools, such as remote login services, aren’t included in the system software, and APIs don’t allow apps to escalate their own privileges to modify other apps or iOS and iPadOS.

Use of entitlements

Access by third-party apps to user information, and to features such as iCloud and extensibility, is controlled using declared entitlements. Entitlements are key-value pairs that are signed in to an app and allow authentication beyond runtime factors, like UNIX user ID. Since entitlements are digitally signed, they can’t be changed. Entitlements are used extensively by system apps and daemons to perform specific privileged operations that would otherwise require the process to run as root. This greatly reduces the potential for privilege escalation by a compromised system app or daemon.

In addition, apps can only perform background processing through system-provided APIs. This enables apps to continue to function without degrading performance or dramatically impacting battery life.

Address Space Layout Randomization

Address Space Layout Randomization (ASLR) protects against the exploitation of memory corruption bugs. Built-in apps use ASLR to ensure that all memory regions are randomized upon launch. Randomly arranging the memory addresses of executable code, system libraries, and related programming constructs reduces the likelihood of many sophisticated exploits. For example, a return-to-libc attack attempts to trick a device into executing malicious code by manipulating memory addresses of the stack and system libraries. Randomizing the placement of these makes the attack far more difficult to execute, especially across multiple devices. Xcode, and the iOS or iPadOS development environments, automatically compile third-party programs with ASLR support turned on.

Execute Never feature

Further protection is provided by iOS and iPadOS using ARM’s Execute Never (XN) feature, which marks memory pages as nonexecutable. Memory pages marked as both writable and executable can be used only by apps under tightly controlled conditions: The kernel checks for the presence of the Apple-only dynamic code-signing entitlement. Even then, only a single `mmap` call can be made to request an executable and writable page, which is given a randomized address. Safari uses this functionality for its JavaScript JIT compiler.

Supporting extensions in iOS, iPadOS, and macOS

iOS, iPadOS, and macOS allow apps to provide functionality to other apps by providing extensions. Extensions are special-purpose signed executable binaries packaged within an app. During installation, the system automatically detects extensions and makes them available to other apps using a matching system.

Extension points

A system area that supports extensions is called an *extension point*. Each extension point provides APIs and enforces policies for that area. The system determines which extensions are available based on extension point–specific matching rules. The system automatically launches extension processes as needed and manages their lifetime. Entitlements can be used to restrict extension availability to particular system apps. For example, a Today view widget appears only in Notification Center, and a sharing extension is available only from the Sharing pane. Examples of extension points are Today widgets, Share, Actions, Photo Editing, File Provider, and Custom Keyboard.

How extensions communicate

Extensions run in their own address space. Communication between the extension and the app from which it was activated uses interprocess communications mediated by the system framework. They don't have access to each other's files or memory spaces. Extensions are designed to be isolated from each other, from their containing apps, and from the apps that use them. They are sandboxed like any other third-party app and have a container separate from the containing app's container. However, they share the same access to privacy controls as the container app. So if a user grants Contacts access to an app, this grant is extended to the extensions that are embedded within the app but not to the extensions activated by the app.

How custom keyboards are used

Custom keyboards are a special type of extension because it's enabled by the user for the entire system. After it's enabled, a keyboard extension is used for any text field except the passcode input and any secure text view. To restrict the transfer of user data, custom keyboards run by default in a very restrictive sandbox that blocks access to the network, to services that perform network operations on behalf of a process, and to APIs that would allow the extension to exfiltrate typing data. Developers of custom keyboards can request that their extension have Open Access, which lets the system run the extension in the default sandbox after getting consent from the user.

MDM and extensions

For devices enrolled in a mobile device management (MDM) solution, document and keyboard extensions obey Managed Open In rules. For example, the MDM solution can prevent a user from exporting a document from a managed app to an unmanaged Document Provider, or using an unmanaged keyboard with a managed app. Additionally, app developers can prevent the use of third-party keyboard extensions within their app.

App protection and app groups in iOS and iPadOS

Adopting Data Protection in apps

The iOS Software Development Kit (SDK) for iOS and iPadOS offers a full suite of APIs that make it easy for third-party and in-house developers to adopt Data Protection and help ensure the highest level of protection in their apps. Data Protection is available for file and database APIs, including NSFileManager, CoreData, NSData, and SQLite.

The Mail app database (including attachments), managed books, Safari bookmarks, app launch images, and location data are also stored through encryption, with keys protected by the user's passcode on their device. Calendar (excluding attachments), Contacts, Reminders, Notes, Messages, and Photos implement the Data Protection entitlement Protected Until First User Authentication.

User-installed apps that don't opt in to a specific Data Protection class receive Protected Until First User Authentication by default.

Joining an App Group

Apps and extensions owned by a given developer account can share content when configured to be part of an App Group. It's up to the developer to create the appropriate groups on the Apple Developer Portal and include the desired set of apps and extensions. Once configured to be part of an App Group, apps have access to the following:

- A shared on-volume container for storage, which stays on the device as long as at least one app from the group is installed
- Shared preferences
- Shared keychain items

The Apple Developer Portal ensures that App group IDs (GID) are unique across the app ecosystem.

Verifying accessories in iOS and iPadOS

The Made for iPhone, iPad, and iPod touch (MFi) licensing program provides vetted accessory manufacturers access to the iPod Accessories Protocol (iAP) and the necessary supporting hardware components.

When an MFi accessory communicates with an iOS or iPadOS device using a Lightning connector or through Bluetooth, the device asks the accessory to prove it's been authorized by Apple by responding with an Apple-provided certificate, which is verified by the device. The device then sends a challenge, which the accessory must answer with a signed response. This process is entirely handled by a custom integrated circuit (IC) that Apple provides to approved accessory manufacturers and is transparent to the accessory itself.

Accessories can request access to different transport methods and functionality—for example, access to digital audio streams over the Lightning cable, or location information provided over Bluetooth. An authentication IC ensures that only approved accessories are granted full access to the device. If an accessory doesn't support authentication, its access is limited to analog audio and a small subset of serial (UART) audio playback controls.