Canonical

WHITEPAPER

Ubuntu Core Security Deep Dive

Canonical Ubuntu Engineering and services

Version: 4.0.1 - 2026-01-27

Contents

The challenge

The classic Linux distribution model has been iterated upon and established over decades, making it extremely flexible and predictable.

However, in a modern application-centric world of interconnected devices and systems, there are new challenges that this classic Linux distribution model cannot solve. In particular:

- The operating system treats installed software as trusted.
- Applications are tightly coupled to specific releases of the OS.
- Applications can change and have side-effects on the OS, and vice versa.
- Broken upgrades in one part of the system often prevent upgrades in the rest of the system.
- Final products require enhanced security guarantees, where users can only perform the tasks the product is designed to support while minimizing attack vectors.

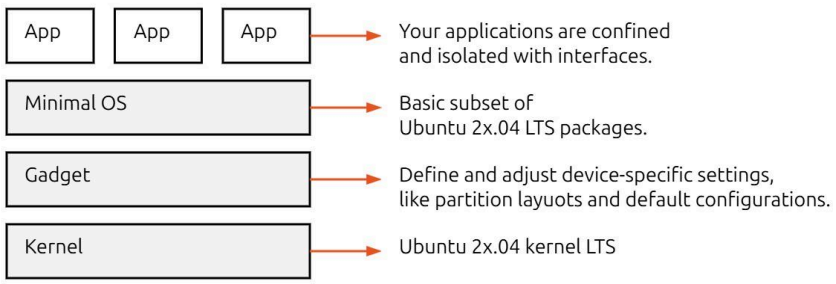
While these challenges affect all Linux systems, they have a particularly large impact on embedded and IoT systems, where security, stability and guaranteed application deployment are absolute requirements.

Introducing Ubuntu Core

Ubuntu Core was designed to solve these challenges. It is image-based and immutable, with every element confined within a separate sandbox. Ubuntu Core is designed specifically to deliver the following key features:

- A security-first approach
- Reliability and robustness out of the box

The base system is a minimal system that consists of three different parts: the kernel, the gadget, and a minimal OS, all of which are packaged and delivered using the snap packaging format. In fact, on Ubuntu Core, all applications are packaged and delivered as snaps.



Developers build Ubuntu Core production images based on their targeted applications and hardware. They control the complete user experience for their Ubuntu Core production images, deciding what applications and configuration the user will be exposed to.

The root filesystem is read-only, and there is a clean separation between the base system and the applications installed on the system as well as a clean separation between the installed applications themselves.

Key advantages

Ubuntu Core is an important evolutionary step for Ubuntu. While it builds upon Linux traditions, Ubuntu Core provides a sharp focus on predictability, reliability and security while at the same time enabling developer freedom and control for deploying their end applications.

Developer velocity and control

Ubuntu Core's app-centric view and trust model puts control in the hands of publishers by decoupling snaps from the system. Application developers do not have to wait for a new release of the Ubuntu distribution to get their latest features to users. Developers publish their applications to the store and make the snap available to devices on a timeline that makes sense for them. They also do not have to worry about unpredictable changes to the system and can avoid problems caused by untested package combinations.

Developers can still take advantage of the vast Ubuntu archive if they want. By building applications on Ubuntu, all of the benefits of Ubuntu toolchain hardening are present. When using snapcraft, developers can bundle Ubuntu archive debs in their snaps. Snapcraft also allows developers to quickly rebuild their snaps with the latest debs, which contain security updates published by the Ubuntu Security team.

Safe, reliable updates for all devices and images

Keeping a system up to date is paramount to good system security. Ubuntu Core's update mechanisms are robust and safe. If a problem surfaces with an application upgrade, it will not cause other application or system upgrades to halt. Snap updates are smaller than traditional updates due to delta upgrades, and systems can run completely unattended to receive application and system updates automatically, rebooting as necessary. If there is a problem, the system also provides rollback mechanisms.

Canonical provides several supported reference kernels, as well as the Ubuntu Core OS snap, reference gadget snaps and pre-built images, all of which are available for anyone to use. All of these layers receive official security support from Canonical. Device manufacturers can partner with Canonical for vendor kernels, gadget snaps and store branding to meet the flexibility needs of any project. Canonical maintains the OS snap and kernel snaps (reference or in partnership) while developers, OEMs and ISVs focus on their applications.

Security provides assurances

In addition to reliable updates, Ubuntu Core provides strong security assurances.. The system design, store policies, and security sandboxing protect the system from tampering, sensitive system information disclosure, and data theft from subverted or misbehaving applications. Developers can also rely on a stable system for their applications and not worry about unpredictable changes causing instability or security problems.

Flexibility

Ubuntu Core provides flexibility for developers. Snaps can ship multiple services and commands that freely interact with each other, and the system provides convenient methods for safe hardware access. Snap interfaces can be used to connect snaps to system resources as well as other snaps available in the public Snap Store and brand stores.

Security

Ubuntu Core is designed with security in mind, from the initial boot, through system updates, to the way devices operate after deployment.

The security model of Ubuntu Core is different from classic Ubuntu, and this is due in large part to its software distribution mechanism. Software is either:

- Part of the minimal OS snap (constructed by Canonical from debs from the Ubuntu archive).
- Pre-installed via the gadget snap (snaps installed by the developer during provisioning).
- Installed after deployment via the store by the developer as application snaps (aka 'app snaps') or snap runtimes (base snaps).

Software on an Ubuntu Core system can then be categorized as:

Trusted by the OS

Software installed as part of the minimal OS snap is considered trusted by the OS because it is built from the Ubuntu archive[1], which means the archive integrity checks for classic Ubuntu also apply for the software shipped in the OS snap. This software may or may not run under confinement. Applications inside the OS snap are trusted by the OS and:

- Can typically access any resources or data available within the user's session.
- Have limited access to system services and data as permitted by the OS (ie, traditional file system permissions, Linux kernel capabilities, DBus bus policy, etc all apply).
- Have been selected and are maintained by Canonical.

Untrusted by the OS

To facilitate developer velocity and enable users to access the latest versions of applications without updating their OS, snaps are distributed publicly via the Snap Store, or privately through Dedicated Snap Stores. Application snaps are considered untrusted (more on this later) and run in a restricted sandbox which provides controlled access through interfaces. This allows for store reviews of app snaps to be shallow and automated as per store policies. Untrusted applications:

- Can freely access their own data (inside their snap package).
- Cannot access other applications' data (outside their snap package).
- Cannot access non-application-specific user data.
- Cannot access privileged portions of the OS.
- Cannot access privileged system APIs.
- May access sensitive APIs with user permission provided the API asks for permission at time of access or the permission is granted to the application outside of snap installation.

Security primitives

Ubuntu Core uses several technologies to implement the security sandbox. The security sandbox is designed so that snaps are integrated into the OS and can interact with one another in controlled ways. A high level interface is broken down into pieces of sandbox configuration applicable to each of the supported primitives described below. For example, a block-devices interface enables access through both AppArmor and device cgroups. The AppArmor profile allows access to path patterns

[1] Kernel and gadget snaps are either supplied by Canonical or the vendor

matching typical block device nodes under /dev, while the device cgroup permits users to open block devices with specific major/minor numbers or those created by relevant kernel subsystems.

Traditional permissions

The Linux kernel enforces Discretionary Access Controls (DAC) via traditional UNIX 'owner' permissions as well as Linux kernel capabilities. The Ubuntu Core base system uses these permissions extensively. For app snaps on Ubuntu Core, by default services run as root and therefore traditional permissions alone do not play as important of a role in the confinement of services. It is also possible to configure a snap to run a service as the snap_daemon / _daemon_ user / group. This user / group is created by snapd and a snap daemon service can drop privileges to this user / group via the system-usernames[2] feature.

AppArmor

AppArmor is a Mandatory Access Control (MAC) system which ensures kernel level enforcement of programs and processes to a limited set of resources. AppArmor restricts processes running either as root or non-root, with confinement policy being provided via profiles loaded into the kernel. AppArmor in Ubuntu Core mediates:

- File access and library loading
- Execution of applications
- Coarse-grained networking
- Linux kernel capabilities
- Coarse owner checks (euid/fsuid matching)
- Mount
- UNIX named, abstract and anonymous sockets
- DBus communications
- UNIX signals
- Process tracing (ptrace)
- POSIX message queues
- io_uring
- User namespaces

An important concept to understand is that when a process is started, an AppArmor label is attached to it. This label maps the process to its policy and is consulted in process interactions and file accesses. The label for a snap's process is the same as its security policy ID (this is performed by the launcher). In this manner, all commands from a snap are given a unique label and all mediation is performed against the AppArmor policy associated with that label. When a process executes another binary, an execution transition is performed (if the policy allows it). AppArmor supports several different execution transitions, but the most important one to remember for typical application confinement in Ubuntu Core is that the child process will inherit the parent's label (and therefore policy).

Seccomp

When user space programs need to interact with hardware (for example, to open a file or to connect to a machine over the network) or other kernel functionality, they do so via the syscall interface to the Linux kernel. The kernel has a few hundred syscalls for a given architecture and a process may set up a syscall filter using the seccomp facility in the Linux kernel to limit the syscalls the process may use. Child processes inherit the parent's seccomp filter and while they can make the filter more strict, they may not make it less strict. The launcher will set a seccomp filter for the command before executing it.

[2] https://snapcraft.io/docs/system-usernames

Namespaces

Namespaces are a facility provided by the Linux kernel that allow for separating processes in such a manner that one namespace cannot see or access resources from another namespace. Several namespaces exist, such as file, network, and mount namespaces. Namespaces play a vital role in container technologies such as LXC/LXD and Docker, but their use is not limited to these full-blown container implementations. Ubuntu Core uses a mount namespace to implement a per-snap private `/tmp` directory and other features such as sharing content between snaps and the system.

In general, snap processes run in the global (ie, default) namespace to facilitate communications and sharing between snaps, and because this is more familiar for developers and administrators. For those desiring full containers, LXD and Docker[3] snaps are available for installation from the store.

Control Groups (cgroups)

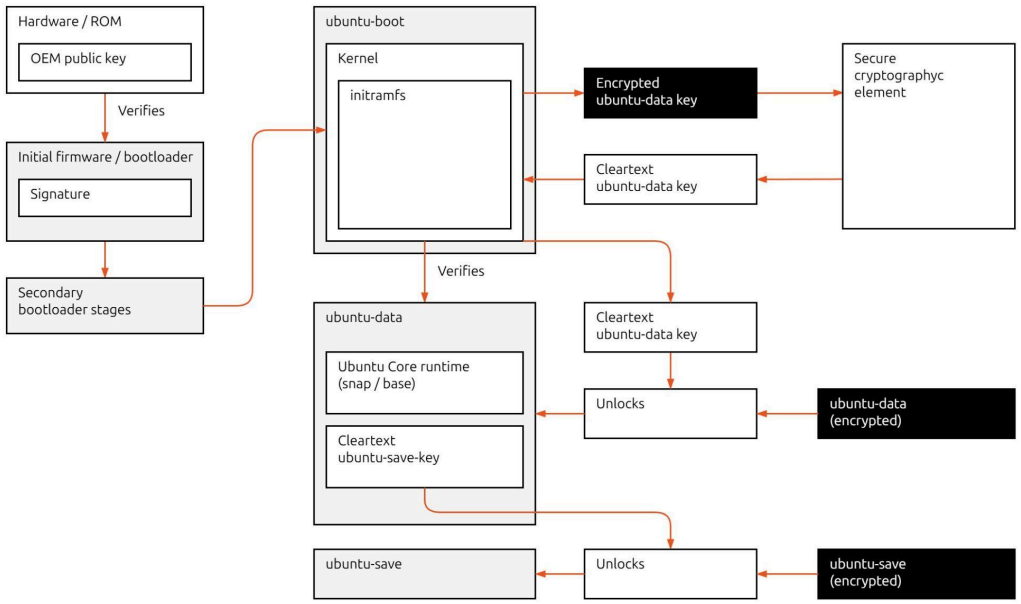
Cgroups can be used to manage various resources, including devices, memory, CPU and network. Ubuntu Core currently uses the devices cgroup for hardware device access controls for hardware assignment.

Boot security

Ubuntu Core devices are primarily designed to be headless or not require system administrator input at boot time. To support such use cases, Ubuntu Core supports Full Disk Encryption (FDE) via either a TPM 2.0 device or some other secure cryptographic element that can be used to store the disk encryption key material.

Boot sequence

The following diagram depicts the high-level flow for a normal boot sequence.



Normal boot sequence

[3] The LXD and docker snaps necessarily have privileged access to the system and are fully functional on Ubuntu Core, therefore users should follow those projects' security guidelines to maintain security on the system.

- The platform hardware verifies the signature of the initial firmware or loader in a platform-specific way, using a key that is burned into the hardware and then executes it.
- The initial firmware or loader may load and execute other bootloaders as required by the platform.
- The final bootloader stage loads and executes the kernel.
- The initramfs recovers the unlock key for ubuntu-data from the secure cryptographic element that protects it.
- The initramfs uses the recovered key to unlock the ubuntu-data container.
- The initramfs uses a key stored inside the unlocked ubuntu-data partition to unlock the ubuntu-save container.
- The initramfs verifies the integrity of the Ubuntu Core system from the unlocked ubuntu-data partition and it ensures it is a system that is authorized to access the confidential data.
- The initramfs builds the root filesystem from the verified snaps and other persistent data in the unlocked ubuntu-data and ubuntu-save partitions.
- The initramfs hands over to continue booting the Ubuntu Core system.

TPM-backed FDE

To protect the integrity of the system, TPM-backed FDE employs UEFI Secure Boot to ensure that the software which is executed comes from a trusted source, as derived from the hardware root-of-trust.

The firmware/BIOS powers on, performs self-checks and verification of firmware modules, then verifies the shim which then verifies the bootloader and passes control to it. If the bootloader is not successfully verified, the boot fails. During this process the Secure Boot policy, firmware modules and bootloader (including shim) are measured into the TPM.

The bootloader verifies the kernel (in this case the unified kernel image (UKI) contains both the kernel itself and the initrd). Once verified, the kernel is booted. If the kernel fails to verify the boot fails. Similarly, the bootloader measures UKI into the TPM, along with the kernel command-line.

The kernel loads the initrd and starts the init process from it, which measures the snap device model into the TPM.

The kernel then loads the various kernel modules to support the hardware platform. For each module, the kernel verifies that it is cryptographically signed by a trusted key. If a module fails to verify, it is blocked from loading and the failed verification is logged.

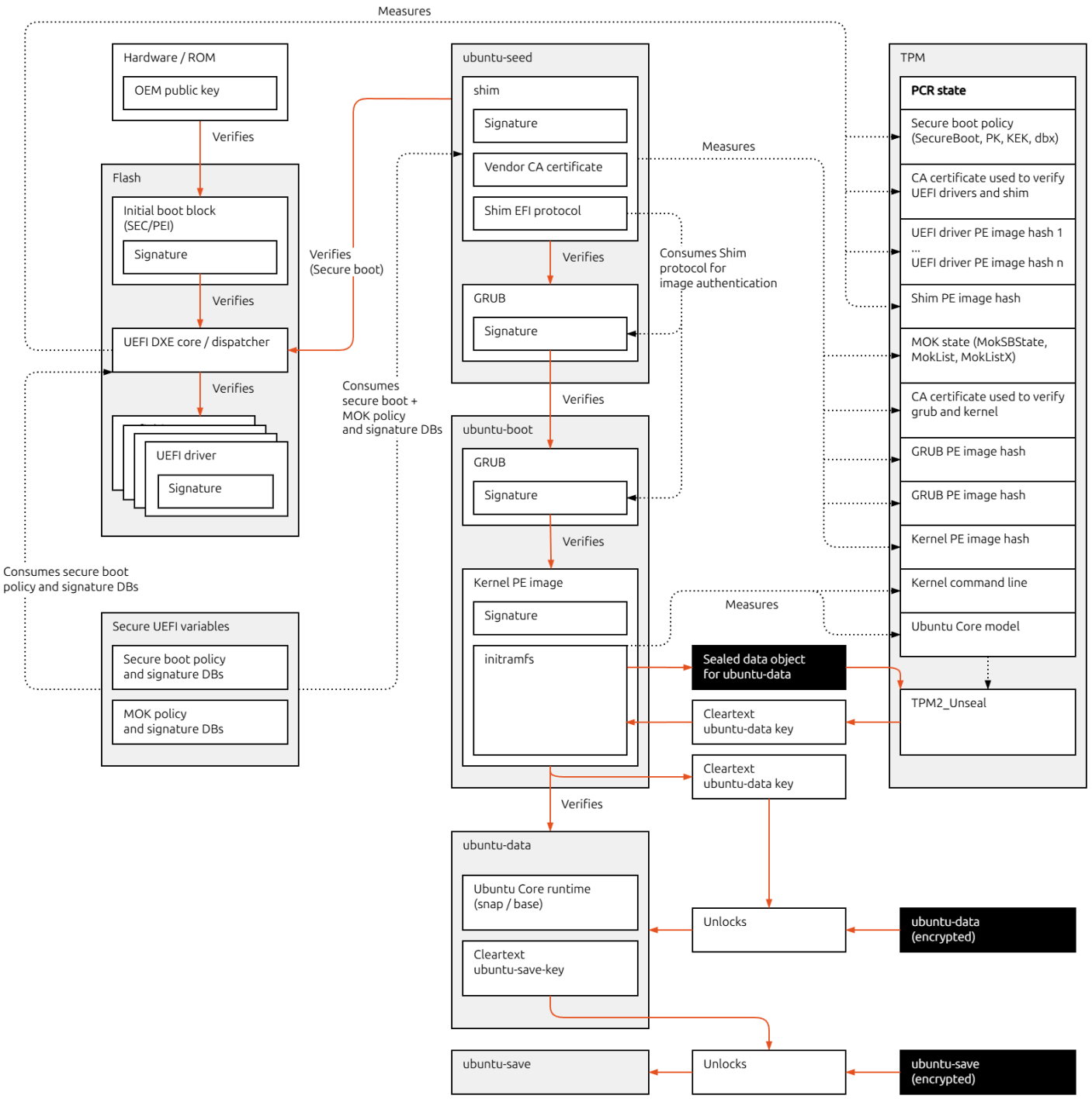
The initrd unseals the disk encryption key from the TPM and unlocks the encrypted disk; or prompts for a recovery key if the TPM authorisation policy does not allow the disk encryption key to be unsealed (for example, due to unexpected changes in TPM measurements etc).

Once the encrypted disk is unlocked, the init process launches console logins, system services, application services and other services, and snapd verifies the various snaps against their store assertions before launching their services.

Measured boot is used to cryptographically record the characteristics and components involved in the boot process back to the hardware root-of-trust. An auditable record is then available to attest to the trustworthiness of the system and this is then used to determine whether to reveal the disk encryption key to the software running on the device.

Normal TPM boot sequence

The following diagram depicts the high-level flow for a normal boot sequence for UEFI-based systems using a TPM 2.0 device.



Boot sequence for UEFI-based systems using a TPM 2.0

- The platform hardware verifies the signature of the firmware's initial boot block from Flash, using a public key burned into the chipset by the device manufacturer.
- The platform firmware measures the secure boot policy (including SecureBoot, PK, KEK, db and dbx variables) to the TPM.
- The platform firmware loads, verifies and starts UEFI drivers, measuring both their PE image hashes and the CA certificates used to verify the drivers to the TPM.
- The platform firmware loads shim from the ubuntu-seed partition and verifies that it has a valid signature, and that it is signed by an authority trusted by the platform manufacturer

(via the UEFI signature database).
- The platform firmware measures both the certificate used to verify shim and the PE image hash of shim to the TPM.
- The platform firmware starts shim.
- Shim measures MOK policy and the MOK signature databases to the TPM.
- Shim loads GRUB from the ubuntu-seed partition and verifies that it has a valid signature, and that it is signed by the authority whose certificate is built into shim, or an authority trusted by the platform manufacturer (via the UEFI signature database), or an authority trusted by the machine owner (via a MOK).
- Shim measures both the certificate used to verify GRUB and the PE image hash of GRUB to the TPM.
- Shim starts GRUB from the ubuntu-seed partition.
- GRUB chainloads another GRUB image from the ubuntu-boot partition, and uses shim's EFI protocol to verify that it has a valid signature, and that it is signed by the authority whose certificate is built into shim, or an authority trusted by the platform manufacturer (via the UEFI signature database), or an authority trusted by the machine owner (via a MOK).
- On GRUB's behalf, shim measures both the certificate used to verify GRUB and the PE Image hash of GRUB from the ubuntu-boot partition to the TPM.
- GRUB from the ubuntu-seed partition starts the chainloaded GRUB from the ubuntu-boot partition.
- GRUB loads the unified kernel and initramfs PE image and uses shim's EFI protocol to verify that it has a valid signature, and that it is signed by the authority whose certificate is built into shim, or an authority trusted by the platform manufacturer (via the UEFI signature database), or an authority trusted by the machine owner (via a MOK).
- On GRUB's behalf, shim measures both the certificate used to verify the kernel image and the PE image hash of the kernel image to the TPM.
- GRUB starts the kernel image EFI stub.
- The kernel image EFI stub measures the supplied kernel command line to the TPM.
- The kernel image EFI stub executes the kernel, supplying the built-in initramfs via the kernel's boot protocol.
- The initramfs measures some properties of the run mode model assertion to the TPM.
- The initramfs recovers the unlock key for ubuntu-data from the TPM by loading the required resources into it, starting an authorization policy session, executing a sequence of policy assertions, and requesting the TPM to reveal the unlock key. This is only successful if the properties that were measured into the TPM during the boot process match those encoded in the authorization policy for the key.
- The initramfs uses the recovered key to unlock the ubuntu-data container.
- The initramfs uses a key stored inside the unlocked ubuntu-data partition to unlock the ubuntu-save container.
- The initramfs verifies the integrity of the Ubuntu Core system from the unlocked ubuntu-data partition using the previously measured model assertion.
- The initramfs builds the root filesystem from the verified snaps and other persistent data in the unlocked ubuntu-data and ubuntu-save partitions.
- The initramfs pivots into the root filesystem to continue booting the Ubuntu Core system.

Authorization policies

Authorization policies provide a mechanism to restrict access to a TPM resource unless a set of conditions are met. An authorization policy is represented by a single digest value. Despite this, authorization policies can be arbitrarily complex. A policy is executed by running a sequence of policy assertion commands that modify the digest value associated with a policy authorization session. When executing a command that requires authorization for a resource, and that authorization can be satisfied by a policy, the TPM will verify that the digest associated with the supplied authorization session matches the resource's authorization policy digest.[4]

[4] https://trustedcomputinggroup.org/wp-content/uploads/TCG_TPM2_r1p59_Part1_Architecture_pub.pdf (see section 19.7, "Authorizations and Acknowledgments - Enhanced Authorization")

Authorization policy digests are computed using the documentation provided for each policy assertion[5].

The authorization policy contains assertions for the properties defined in either the Secure Boot chain of trust, the Secure Boot chain of trust with the boot manager code strategy, or the combined driver and boot manager code strategy.

In addition to the above, the authorization policy also contains the following assertions:

- The kernel has been executed with a previously authorized command line.
- The Ubuntu Core system to be booted is one that has previously been established to be authorized to access the confidential data.
- The correct passphrase has been supplied (see Passphrase).
- The PCR policy has not been revoked.

This requires the following policy assertions:

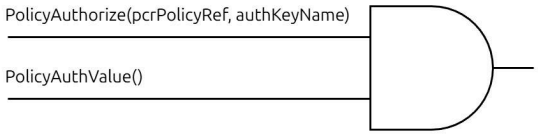
- TPM2_PolicyPCR[6] on PCR12, which is reserved for use by the Ubuntu Core kernel image (see Kernel image).
- TPM2_PolicyNV[7] for the PCR policy revocation check (see PCR policy revocation).
- TPM2_PolicyAuthValue[8] for the passphrase check (see Passphrase).

The authorization policy is split into 2 parts:

- A fixed part that is associated with the sealed key object and cannot be changed without creating a new object.
- An updateable PCR policy, authorized by an elliptic-curve key associated with the static part.

This permits updating of the PCR policy without having to reseal the disk unlock key, which would require knowledge of the passphrase if one has been set (see PCR policy updates).

The fixed part of the authorization policy is constructed as shown:



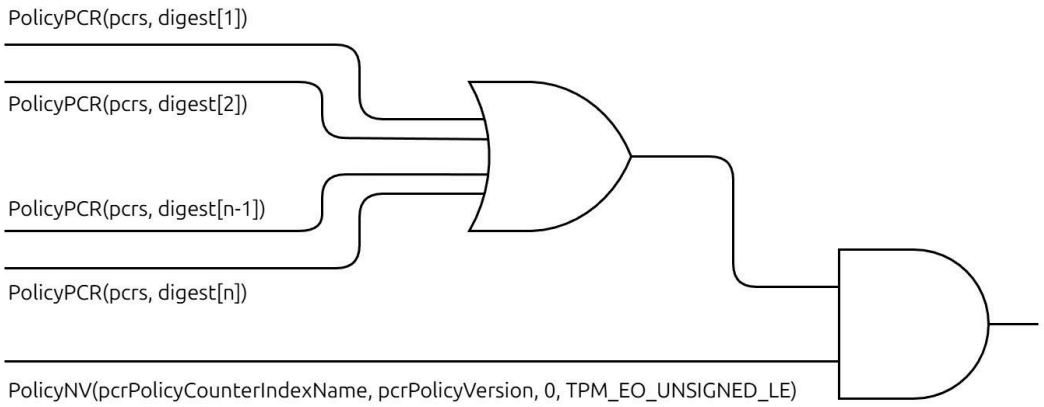
This makes use of the TPM2_PolicyAuthorize[9] assertion, which permits an authority to sign updated authorization policies. This command asserts that the current session digest matches a digest that has been approved by this authority.

[5] https://trustedcomputinggroup.org/wp-content/uploads/TCG_TPM2_r1p59_Part3_Commands_code_pub.pdf (see section 23, "Enhanced Authorization (EA) Commands")
[6] https://trustedcomputinggroup.org/wp-content/uploads/TCG_TPM2_r1p59_Part3_Commands_code_pub.pdf (see section 23.7, "Enhanced Authorization (EA) Commands - TPM2_PolicyPCR")
[7] https://trustedcomputinggroup.org/wp-content/uploads/TCG_TPM2_r1p59_Part3_Commands_code_pub.pdf (see section 23.9, "Enhanced Authorization (EA) Commands - TPM2_PolicyNV")
[8] https://trustedcomputinggroup.org/wp-content/uploads/TCG_TPM2_r1p59_Part3_Commands_code_pub.pdf (see section 23.17, "Enhanced Authorization (EA) Commands - TPM2_PolicyAuthValue")
[9] https://trustedcomputinggroup.org/wp-content/uploads/TCG_TPM2_r1p59_Part3_Commands_code_pub.pdf (see section 23.16, "Enhanced Authorization (EA) Commands - TPM2_PolicyAuthorize")

The labels have the following meanings:

- pcrPolicyRef - a reference for the PCR policy, computed as PCR policy revocation.
- authKeyName - the name[10] of the elliptic key authorized to sign new PCR policies (see PCR policy updates).

The updateable PCR authorization policy is constructed as follows:



This makes use of the TPM2_PolicyOR assertion to permit more than one set of PCR values so that a key can be sealed to more than one device configuration. The TPM has a limit of 8 conditions for TPM2_PolicyOR, so the design should make use of nested TPM2_PolicyOR commands in order to permit an arbitrary number of conditions.

The labels have the following meanings:

- pcrs – the PCR selection. This should use SHA-256 PCR banks, and the selected PCRs will be dependent on the chosen protection strategy (PCRs 7 and 12 for Secure boot chain of trust strategy, PCRs 4, 7 and 12 for Combined secure boot chain of trust and boot manager code strategy, or PCRs 2, 4 and 12 for Driver and boot manager code strategy).
- digest[x] – an authorized composite PCR digest, computed from the authorized PCR values associated with the selection (see Computing PCR values).
- pcrPolicyCounterIndexName – the name of the NV index used for PCR policy revocation (see PCR policy revocation).
- pcrPolicyVersion – the PCR policy version.

System security

The Ubuntu Core base system contains little more than a kernel, the init process, snapd itself, and the standard Linux/UNIX tools, and the libraries which support these tools. A few other non-network-facing services also exist to support normal system operation. These include init (systemd), systemd-journald, systemd-networkd, systemd-udevd, systemd-timesyncd, systemd-logind, systemd-resolved, dbus-daemon, and snapd.

The result is a reduced attack surface, limited to the kernel syscall interface, its modules, and /proc and /sys entries, alongside a limited number of optional services:

- a DHCP (dhclient) client running under a restrictive AppArmor profile.
- OpenSSH server configured with 'PermitRootLogin prohibit-password' to disable root logins

[10] https://trustedcomputinggroup.org/wp-content/uploads/TCG_TPM2_r1p59_Part1_Architecture_pub.pdf (see section 16, "Names")

when a password is set. /etc/ssh/sshd_config is writable, so it can be updated for site-specific needs.

Because the base system is built from the Ubuntu archive, it benefits from all of the OS and toolchain hardening features of classic Ubuntu.

Application snaps by default are disallowed access to all system services and have limited access to kernel interfaces. Ubuntu Core also provides restricted interfaces that the gadget snap developer or device administrator may assign (connect) to trusted snaps.

Users

Ubuntu Core disables password-based logins for root and all system users; only assertion-authorized users may gain administrative access via sudo.

As part of provisioning, one user account is set up using the Launchpad ID as the name of the user. By default this user has console access disabled and SSH access allowed via the ssh key stored in Launchpad for the user. This user is in the 'sudo' group and is thus capable of running commands as root (by default without password prompting but configurable via '/etc/sudoers.d/create-user-<username>'). Console access may be allowed by setting a password for the user.

Ubuntu Core supports a few specific, shared accounts that can be used to run services under a less-privileged user than 'root', in particular the daemon user with UID and GID of 584792.

Alternate user accounts can be set up using standard tools (eg, 'addgroup', 'adduser', etc).

Services and user commands

The Ubuntu Core system is designed to segregate the system from applications, and to segregate applications from each other, while providing controlled mechanisms for interacting with each other.

Snaps provide commands that may or may not run in the background. Commands that the user runs via console logins are installed in /snap/bin, which is included in the console user's PATH. Services are commands that run in the background and are long running processes that integrate with the systemd boot process to start automatically on boot. While the systemd unit specification is very rich, Ubuntu Core only exposes a small subset of the specification in the snap packaging, such as start-command, stop-command, post-stop-command, stop-timeout, daemon type. On system install, the systemd unit file is autogenerated based on these configuration options. This prevents snaps from interacting with systemd and the system in uncontrolled ways.

Certain Ubuntu Core interfaces provide DBus bus policy and security policy to allow a snap to run as a DBus service. While the DBus bus policy specification is also rich, the OS snap itself provides the bus policy instead of the snaps implementing the interface.

Importantly, to prevent unexpected interactions and maintain co-installability, whenever the system and snaps could potentially collide, the system will use naming conventions to avoid a collision. For example, snaps are installed to a snap-specific path, DBus connection names and interfaces are provided by the OS snap, and socket activation and connections must follow snap-specific naming schemes (discussed in detail, below).

Application security

The application security model relies on the sandbox to provide a least privilege execution environment and a combination of security and store policies that work together to extend the sandbox to accommodate application needs.

The security policies are modeled as ‘interfaces’, with each capturing a piece of system functionality. Interfaces have a slot side, implemented by the consumer or provider of the given functionality, and a plug side, listed by a consumer needing to access said functionality. An interface connection opens parts of the application sandbox, again keeping the exposed surface minimal.

An example of such an interface is hardware-observe, allowing access to the generic hardware identification information of the system. The slot is provided by the host OS, and the plug side is declared by a snap application. An example would be an application that needs to collect generic hardware diagnostic data and report it to the user or some network backend.

Snap Store

When using snaps from the Snap Store, if the snap publisher of an application releases a new version, the snap package is uploaded to the Snap Store and the snap undergoes automatic reviews. Part of this review process involves examining the snap’s requested interfaces (detailed further below).

If a snap passes the review, it can be made immediately available to users and devices. If it does not pass review, it is blocked and the uploader may request a manual review. The store’s upload policies and the security policies associated with an individual snap’s declared interfaces work together to ensure that users and devices are protected from malformed or malicious snaps.

The official public Snap Store is intended for anyone, so it therefore has very strict upload policies. Snaps implementing very sensitive Ubuntu Core interfaces must be pre-approved using store snap declaration assertions.

Dedicated stores (hosted via the Snap Store) are designed for individuals and companies to distribute software privately, with various controls that are appropriate for an organization’s internal policies. Dedicated stores are essentially owned by a particular brand (ie, individual/company) and may optionally specify different upload policies as the brand sees fit. For example, it is possible to restrict uploads to certain employees in the brand’s company. The company may also change their store upload policies or snap declaration assertions to allow app snaps implementing a sensitive security interface to pass automated review, provided they come from the owner of the store. Access to dedicated stores can be configured to be limited to either all devices owned by the brand, or specific models owned by the brand.

App snaps

App snaps are the primary method for making an Ubuntu Core system useful. App snaps are used to ship services that should start on boot or commands that the user or admin can run. All app snaps run under confinement, in a restrictive security sandbox that is configured in the snap packaging.

Because the Ubuntu Core base system is minimal, snaps explicitly bundle their dependencies, which ensures that developers can carefully control the application’s runtime environment. This technique adds predictability and reliability to the development process, but it also means that publishers are responsible for security issues found in the bundled parts.

Snapcraft also makes it easy to rebuild those snaps with updated external packages after they receive a security fix. Publishers can opt into receiving notifications for Ubuntu archive software

that has received a security update since the snap was last published.

The snap packaging store will also automatically notify snap publishers when an updated package with a new security fix is available for one of the dependencies bundled within their snap. There is no mechanism that provides such feedback for other source types. Application developers need to do their due diligence to ensure they address security issues in bundled, third-party software.

Gadget snaps

Gadget snaps are used to declare hardware capabilities to the system and optionally establish interface connections during first boot for snaps.

Robustness

The OS and installed applications are treated separately, which means their update processes differ. In general, the base system is delivered via three snaps that work together:

- OS[11]: provided by Canonical.
- Kernel: provided by either Canonical[12] , the vendor of a particular device, or the open source community.
- Gadget: provided by either Canonical, the vendor of a particular device, or the open source community.

The kernel snap provides the kernel and drivers, the OS snap provides the rest of the operating system and the gadget snap provides boot, kernel, OS, and application configuration. Each of these snaps may be updated independently of one another.

For Canonical-provided OS and kernel snaps, since the snaps are based on debs from the Ubuntu archive, the update process relies upon the classic Ubuntu archive distribution model (though the delivery to devices is of course as 'snaps' instead of 'debs').

The OS and kernel snap update process has an additional step over classic Ubuntu: the channel[13] which can be thought of as 'risk level'. When an Ubuntu Core system is initially provisioned, it will contain a core snap (eg, core22, built from Ubuntu 22.04 LTS[14]) and a kernel snap. These snaps will target a particular channel (for example, 'stable'). Updates will automatically flow from the Ubuntu archive to the Snap Store into the 'edge' channel for a given Ubuntu core snap and kernel snap. Updates to the 'stable' channel may originate from other channels (such as 'beta' or 'candidate') and undergo additional QA so that the stable channel is updated on a predictable cadence. High impact security updates or critical bug fixes may be pushed to the stable channel outside of this process and cadence as necessary.

For gadget and vendor kernel snaps, the origin of the snap is the publisher/vendor of the snap instead of the Ubuntu archive, and updates to these snaps are done via direct upload to the Snap store (or dedicated store). The store limits who can upload what via store accounts.

Ubuntu Core systems retain previous versions of snaps to support rollbacks and improve reliability[15] . For example, if the system fails to boot after a kernel or OS snap update, it automatically reboots using the previous known-good kernel or OS snap.

Ubuntu developers who want to update the OS or reference kernel snap will:

- Update a Debian source package (targeting a particular classic Ubuntu release), sign it with the GPG key associated with the developer's account in Launchpad, then upload to Launchpad where it is verified, built, automatically tested, and published to the archive.

Canonical's automated processes will, per Ubuntu release and channel:

- Via 'apt', notice that packages in the Ubuntu archive are newer than what is in the current snap for this release/channel.

[11] Including the aforementioned 'snapd' snap
[12] Canonical provides reference kernels based on officially supported kernels in the Ubuntu archive but Canonical can also work with vendors to provide vendor kernels
[13] See http://snapcraft.io/docs/reference/channels for the full list of channels and how they are used
[14] Unlike classic Ubuntu, Ubuntu Core snaps are not released on a 6 month cadence and therefore the version number corresponds only to the year of the release (typically corresponding to the year of classic Ubuntu's Long Term Support release)
[15] Ubuntu Core 15.04 used system images with an a/b partitioning scheme instead of snaps to support rollbacks

- Rebuild the snaps using the contents of the newer versions of the debs.
- Publish the snap (and its associated deltas) as per the channel policies.
- Calculate snap deltas as per Snap Store policies.

Ubuntu Core systems will:

- Via https, notice that an update is available.
- Via https, calculate whether to use snap deltas or the full snap.
- Via https, download the updated snap or snap deltas.
- Verify the signature of the snap/applied snap deltas with the server's public key.
- Update the boot configuration to use the new OS and/or kernel snap.

Ubuntu Core systems are configured by default to automatically install updates and reboot[16] to improve security, but this can be changed as desired. The 'snap refresh <snap name>' command may also be used at any time to trigger an update, prompting a reboot if necessary. The store also provides a gating mechanism for publishers to coordinate updates of multiple snaps owned by the publisher, through the use of validation assertions or validation sets. Finally, a validation set can also be used to list specific snaps that are either required to be installed together or are permitted to be installed together on a device or system.

Operating system verification

The disk unlock keys are protected by a secure cryptographic element that will reveal the keys to software running inside of the initramfs only when the software executing on the device is authorized to access the confidential data. As the bootloader and software assets that execute to the point of disk unlocking may be shared between multiple Ubuntu Core models and brands, the chain of trust typically ends at this point. In order to extend the chain of trust beyond this point to the runtime OS, a model assertion[17] (or set of model assertions) is authorized by creating a cryptographic binding between the disk unlock key and the model assertion at installation time (see Key protection). This facilitates the ability to restrict access to confidential data to a runtime OS associated with a specific brand and model.

The steps for verifying the runtime OS and ensuring that it is authorized to access the confidential data are as follows:

1. Using the HMAC key recovered from the platform protected key data, compute the HMAC of the model properties from the model assertion associated with the runtime OS that will be executed (see Key protection).
2. Verify that the computed HMAC matches one of those stored with the protected key metadata (see JSON format).
3. Verify that the model assertion is valid.
4. For each of the snaps listed in the model assertion:
    a. Find and validate the corresponding snap-declaration[18] assertion.
    b. Using the snap-id from the snap-declaration assertion, and the version of the snap to be loaded, find and validate the corresponding snap-revision[19] assertion.
    c. Perform verification of the snap image by calculating a digest of the image and comparing it against the digest within the verified snap-revision assertion.

[16] The timing of updates can be influenced via OS snap configuration or Dedicated Store controls
[17] https://ubuntu.com/core/docs/reference/assertions/model
[18] https://ubuntu.com/core/docs/reference/assertions/snap-declaration
[19] https://ubuntu.com/core/docs/reference/assertions/snap-revision

Boot modes

The following boot modes are supported:

- run (default) – The normal boot sequence using a kernel from ubuntu-boot and an Ubuntu Core system from ubuntu-data. In this mode the ubuntu-save container is unlocked with its primary unlock key, which is stored inside of the ubuntu-data container.
- recover – Boot into a recovery system from ubuntu-seed. This mode may be used for interactive debugging and recovery tasks. In this mode both ubuntu-data and ubuntu-save containers are unlocked and available to the recovery system using either the primary disk unlock keys protected by the secure element, or using the fallback recovery key.
- install – Boot into a recovery system from ubuntu-seed in order to install the system as specified by the brand in the recovery system. This mode is used to create the ubuntu-data and ubuntu-save containers.

Key protection

Disk unlock keys are protected by a secure cryptographic element associated with the platform. This element will only reveal the key to code executing inside the initramfs if the software running on the device is authorized to access the confidential data.

Confidential data must be accessible only to an authorized runtime OS associated with a specific brand and model. This is particularly important in ecosystems where kernels or other bootloader assets may be shared across different brands. To enforce this restriction, a cryptographically secure binding is created between the encrypted container and the properties defined in the model assertion. A model assertion is a document signed by a brand that defines the fundamental properties of the system, including which essential snap packages the system is composed of.

The binding to the container is by way of a HMAC of the model assertion properties, generated with a 256-bit key that is derived from an authorization key protected in the same payload as the disk unlock key. This facilitates the ability to extend the chain of trust to the OS runtime and to ensure that it has not been modified (see Runtime OS verification). This binding uses a HMAC-SHA256, computed as follows:

$$M_1 = HMAC_{SHA256}\left(K_{HMAC}, signKeyHashAlg||signKeyHash||brandID\right)$$
$$M_2 = HMAC_{SHA256}\left(K_{HMAC}, M_1||model\right)$$
$$M = HMAC_{SHA256}\left(K_{HMAC}, M_2||series||grade\right)$$

where:

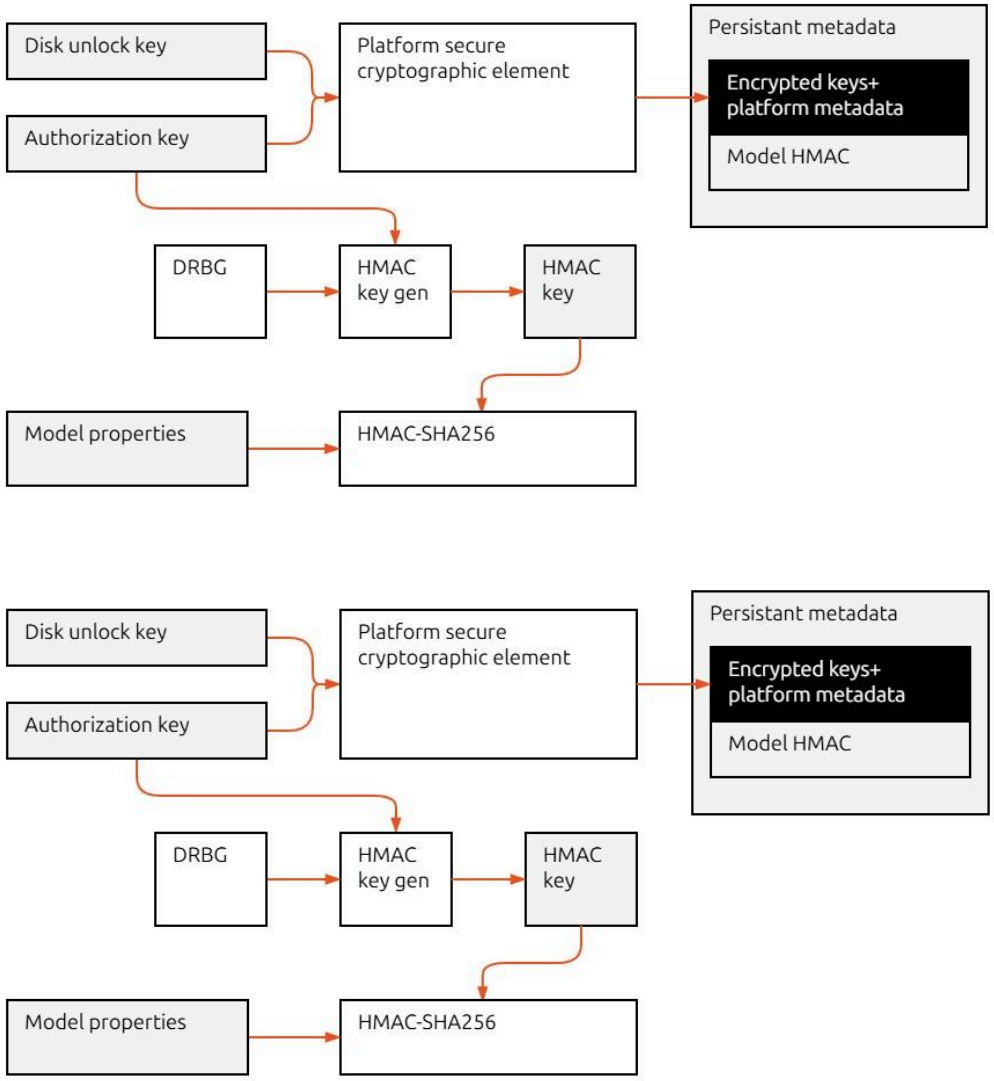
- signKeyHashAlg -- the OID of the algorithm used to compute signKeyHash.
- signKeyHash - digest of the public key that signed the model assertion, in binary form.
- brandID -- the brand-id field of the model assertion, without the NULL terminator.
- model -- the model field of the model assertion, without the NULL terminator.
- series -- the series field of the model assertion, without the NULL terminator.
- grade -- the 4-byte numeric representation of the grade defined by the model assertion, in little-endian format.

The HMAC key is derived from a 256-bit randomly generated authorization key that is protected in the same payload as the disk unlock key. The key is derived using a DRBG seeded with this value, using the block cipher based DRBG recommended by SP800-90A[20]. This facilitates the ability to derive other keys required for the maintenance of the protected disk unlock key as required by

[20] https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-90Ar1.pdf

specific platform implementations.

The following diagram details how a key is protected. The encrypted payload containing the disk unlock key and authorization key, platform-specific metadata, and one or more model HMACs are stored as described in Protected key storage.



All keys managed by the system use the same authorization key for computing and verifying model HMACs.

Fallback recovery key

A fallback recovery key is supported for the ubuntu-data container, which can be used to unlock it in the case that the primary unlock keys cannot be recovered normally. This enables the device owner to recover the device in the event of accidental destructive actions by an administrator or to facilitate data recovery on discrete storage devices outside of the original host platform. The recovery key is expected to be backed up by the device owner and is automatically generated in the form of a numeric string, allowing it to be input via any keyboard layout.

Since this key operates independently from the secure element, it has been designed to be secure against offline attacks, whilst still being feasible to be entered manually by a human operator. A 128-bit key is split into 8 x 16 little-endian numbers, each represented by a base-10 number, padded with leading zeros to 5 digits.

The Argon2i KDF is used for the recovery LUKS keyslot with the following parameters:

Target time	**2 seconds (-5%/+10%)**
Minimum memory cost	32KiB
Maximum memory cost	1GiB
Minimum iteration count	4
Max threads	4

Logging

Ubuntu Core shares the same logging scheme as classic Ubuntu. The only notable differences are that 'auditd' is not currently available on Ubuntu Core images, logging customization is limited and Ubuntu Core uses systemd's 'journalctl' command to obtain snap-specific logs. journald is configured for ephemeral logs with persistent logging via rsyslog. rsyslog can be disabled with 'snap set core service.rsyslog.disable=true'.

In addition to standard system logs, Ubuntu Core provides the 'snap changes' command to view snap installs, removals, and interface connects/disconnects, with more information being added in future OS snap updates.

Ubuntu Core has tools that can be used to view and monitor sandbox denials (e.g. the 'snappy-debug' snap). Please see the online documentation for details.

Confinement

Confinement for snaps is implemented with a security profile provided by the OS snap and is extended via Ubuntu Core interfaces. A snap's packaging may declare requested interfaces to consume via plugs and requested interfaces to provide via slots (if unspecified, the default policy is used with no additional interfaces). Upon snap install, snap-specific security policy is autogenerated with any auto-connectable interface security policy. Interfaces that provide privileged access to the system are not auto-connected by default and the administrator of the system must manually connect these via the 'snap connect' command (the gadget developer may auto-connect preinstalled snaps via the 'prepare-device' hook).

When a snap is uploaded to the store, part of the review process involves examining the snap's declared interfaces to determine if the snap meets the store's upload policies. Certain interface declarations will trigger a manual review in the public Snap Store, although the store may also grant snap declaration assertions for the snap that enable it to pass automated review. Brand stores may also issue snap declarations for snaps they control. As part of the install of a snap, the installation process will download any snap assertions and consider them for snap interface connections. For example, snap declarations may be used to allow a specific snap to use a sensitive interface or to specify that an interface may be auto-connected for a particular snap.

When this section refers to 'snap packaging' for defining security policy for snaps, it applies to both 'meta/snap.yaml' and 'snapcraft.yaml'[21].

[21] The details of working with Ubuntu Core security policy are discussed in depth in the online documentation

Default policy

The default security policy defines the bulk of Ubuntu Core's overall security policy. The default policy on Ubuntu Core enforces adherence to the Ubuntu Core FHS with enough access to the base system for the program to run. Specifically:

- read-only access to SNAP, which is set to the versioned install path
- write access to SNAP_DATA which is set to a versioned path in /var
- write access to SNAP_USER_DATA when the owner of the process and the owner of the file matches. SNAP_USER_DATA is set to a versioned path in /home
- write access to SNAP_COMMON which is set to a snap-specific path in /var
- write access to SNAP_USER_COMMON when the owner of the process and the owner of the file matches. SNAP_USER_COMMON is set to a snap-specific path in /home
- write access to XDG_RUNTIME_DIR which is set to a snap-specific path in /run/user/<uid>
- read-only access to install path and data directories of previous versions
- access to system libraries and a subset of executables in /bin, /sbin, /usr/bin and /usr/sbin
- write access to shared memory files (ie, /dev/shm/snap.SNAP_NAME.*)
- allow processes from the same snap to communicate with each other via abstract and anonymous sockets
- allow processes from the same snap to signal each other via signals
- various common accesses that are deemed safe

All commands within a snap share the same readable and writable areas, and share the same mount namespace (/tmp). Importantly, the security policy dictates that files in SNAP_USER_DATA and SNAP_USER_COMMON[22] must be owned by the uid of the process accessing the files, which means that a snap's daemons and commands running under sudo will not be able to access files in the non-root users' home directories (eg, under /home). In practice, this means that snaps need to consult HOME, SNAP_USER_DATA and SNAP_USER_COMMON for their per-user files. Shared files should be stored in SNAP_DATA or SNAP_COMMON under a directory with permissions appropriate for file sharing (eg, a directory with the sticky bit permission set).

Devmode

To ease application development, Ubuntu Core supports installing applications in 'devmode'. In this mode, snap confinement policies are not enforced; instead, any policy violations are logged to aid development and debugging. This is often helpful for developers when first making a snap. For example:

```
$ sudo snap install hello-world --devmode
```

The packaging yaml also supports the 'confinement' directive to specify whether or not the snap is expected to work correctly when the specified interfaces are connected and the snap is confined. Specifying 'strict' in the packaging yaml indicates the snap works properly when confined and 'devmode' indicates it only works properly when unconfined. If 'confinement' is unspecified, the snap is assumed to work correctly when confined since developers are expected to develop their snaps for running in the sandbox. Uploads targeting the 'stable' channel in the store may not specify 'confinement: devmode' in the packaging yaml. Importantly, specifying '--devmode' is always required when installing a snap in devmode regardless of how 'confinement' is set in its packaging yaml.

Classic

The installation process also supports installing applications that specify 'confinement: classic' which disables all security mechanisms in the snap and allows it to run completely unconfined. This directive can be useful for some types of low-level tools or when first developing snaps as a step

[22] This restriction also typically applies when using the 'home' interface

towards devmode. Use of 'confinement: classic' is restricted by the store and snaps specifying this directive are not installable on Ubuntu Core[23].

Interfaces

Ubuntu Core implements many interfaces for snaps to request and use. The list of interfaces will grow throughout the lifetime of Ubuntu Core[24] and at any given time the list of available interfaces on a particular device (and the connection status) can be seen with 'snap connections'.

An important concept with interfaces is the connection: the OS snap or a snap service command implements a slot interface and a client may plug into (use) the slot when an Ubuntu Core interface connection is made. Interfaces form a contract between the slot provider and the plug consumer such that any snaps using a given interface will be able to interoperate with each other.

To illustrate the concept of slots and plugs, consider a bluez5 snap that implements the bluez slot via its bluetoothd command:

```
name: bluez5
...
apps:
  bluetoothd:
    command: bin/...
    slots: [bluez]
    plugs: [network]
```

When the bluez5 snap is installed, the security policy ID for bluetoothd will be 'snap.bluez5.bluetoothd' and it will have the default security policy, 'network' policy (since it is auto-connected), and the slot security policy needed for 'bluetoothd' to run. A snap that wants to connect to bluetoothd from bluez5 would use:

```
name: bluez-client
...
apps:
  cmd:
    command: bin/...
    plugs: [bluez]
```

When the bluez-client snap is installed, the security policy ID for 'cmd' will be 'snap.bluez-client.cmd' and it will get the default policy only (since the bluez interface is not auto-connected upon install). Using the 'snap connect' command (e.g. 'snap connect bluez-client:bluez bluez5:bluez') the administrator would connect 'cmd' to 'bluez5.bluetoothd' and the security policy would be regenerated to allow 'cmd' to communicate with 'bluez5.bluetoothd'.

Interfaces may either be declared per-command or per-snap. If declared per-snap (that is,. plugs and/or slots are declared in the top-level packaging yaml), all the commands within the snap have the interface security policy added to the command's security policy when the interface is connected. If declared per-command (i.e. plugs and/or slots are declared in the command's section of the packaging yaml, as in the above examples), only the commands within the snap that declare use of the interface have the interface security policy added to them. As a result, the 'snap connect' and 'snap disconnect' commands need only the snap name and not the command name.

The gadget snap may configure auto-connections for the device to avoid the 'snap connect'

[23] Classic snaps may only be used on traditional distributions like Ubuntu Server and Desktop
[24] See the snapcraft documentation for the most up to date list

command for pre-approved interface connections.

Ubuntu Core interfaces

The number of interfaces for Ubuntu Core is large and only a few representative interfaces are listed here to demonstrate what interfaces can do:

- firewall-control: can configure the firewall via netfilter and sysctl and grants the CAP_NET_ADMIN capability. Because this interface grants privileged access, this interface is not auto-connected on install and store policies may trigger a manual review for uploads of snaps specifying this interface.
- home: can access non-hidden files in user's $HOME. Because this interface grants privileged access, this interface is not auto-connected on install and store policies may trigger a manual review.
- log-observe: can read system logs from 'journald', /var/log and also adjust kernel printk rate limiting. Because this interface grants privileged access, this interface is not auto-connected on install and store policies may trigger a manual review for uploads of snaps specifying this interface.
- network: can access the network as a client.
- network-bind: can access the network as a server.
- network-control: can configure networking and network namespaces via sysctl, administrative commands (eg, ifconfig, route, ip, arp, etc) and grants the CAP_NET_ADMIN capability. Because this interface grants privileged access, this interface is not auto-connected on install and store policies may trigger a manual review.
- network-observe: can query network status information via sysctl, administrative commands (eg, route, netstat, etc). Because this interface grants privileged access, this interface is not auto-connected on install and store policies may trigger a manual review for uploads of snaps specifying this interface.
- snapd-control: can communicate with snapd over a UNIX socket to issue commands equivalent to those available through the snap command line tool. Because this interface grants privileged access, this interface is not auto-connected on install and store policies may trigger a manual review for uploads of snaps specifying this interface.
- system-observe: can query system status information via /proc and the ps command. Because this interface grants privileged access, this interface is not auto-connected on install and store policies may trigger a manual review for uploads of snaps specifying this interface.
- time-control: can set system time. Because this interface grants privileged access, this interface is not auto-connected on install and store policies may trigger a manual review for uploads of snaps specifying this interface.

Ubuntu Classic interfaces

Starting with Ubuntu 16.04, all Ubuntu flavors may install snaps from the Snap Store. This includes classic Ubuntu Desktop systems, which use the X window system or Wayland display protocol. On classic distributions, miscellaneous interfaces are provided for the various traditional desktop accesses required for a desktop application to run. Several of these interfaces do not provide full isolation of graphical applications running within a specific user's session (such as mediation for keyboard, mouse, screen grabs, clipboard, Xsettings, which are not supported). Snaps of graphical applications using X should therefore only be installed from trusted publishers[25]. Since Ubuntu 18.04 LTS, users can optionally run a graphical session using the Wayland protocol which is designed to address the security shortcomings of X.

Some interfaces are available only on classic Ubuntu systems (ie, not on Ubuntu Core). Some representative interfaces are:

- desktop: can use modern DBus APIs and resources required to run under modern desktop environments as a client. This interface is safe to use and is auto-connected.
- desktop-legacy: can use historic DBus APIs and resources required to run under various

[25] This only applies to classic Ubuntu flavors using X. Ubuntu Core is not affected.

historic desktop environments as a client. This interface grants privileged access to the user's session via DBus APIs. This interface is auto-connected.

- gsettings[26]: can use the user session's global gsettings database. This interface grants privileged access to the user's settings and is auto-connected.
- opengl: can use OpenGL hardware and libraries. This interface is auto-connected.
- wayland: can access a display server implementing the Wayland protocol. This interface is safe to use and is auto-connected.
- x11: Can use X as a client. This interface grants privileged access to the user's session via the shared X server. This interface is auto-connected.

System interfaces

In addition to these Ubuntu Core and Ubuntu Classic system interfaces, other interfaces exist on Ubuntu Core to facilitate snaps connecting to each other. For example, a snap that provides a D-Bus service declares a slot interface, which other snaps can consume by declaring a corresponding plug. The OS snap ships the interfaces and associated security policies for both the slot side (server) and the plug side (client), and the server snap implements the service within the added slot side security policy. Some representative additional interfaces available when a snap implementing the interface is installed are:

- bluez: slot policy allows access for the bluez bluetooth service and provides DBus bus policy. Plug policy allows access to the services implementing the corresponding slot. Both sides require privileged access. The plugs side is not auto-connected on install. Store policies may trigger a manual review for uploads of snaps specifying this interface.
- mir: slot policy allows access for the Mir display server. Plug policy allows access to the services implementing the corresponding slot. Slot side provides privileged access and store policies may trigger a manual review for uploads of snaps specifying this slot interface. The plug side will be auto-connected.
- network-manager: slot policy allows the NetworkManager service to configure networking on the device and provides the required D-Bus bus policy. The plug policy allows access to the services exposed by the corresponding slot. Both sides require privileged access, and plugs are not auto-connected at install time. Snaps that declare this interface may require manual review during store submission.
- pulseaudio: slot policy allows access for the pulseaudio service to playback and record[27] audio. Plug policy allows access to the services implementing the corresponding slot. Slot side provides privileged access and store policies may trigger a manual review for uploads of snaps specifying this slot interface. The plug side will be auto-connected[28].
- docker: slot policy allows access for docker to manage containers. Plug policy allows access to the service implementing the corresponding slot via the docker admin socket. Both sides require privileged access and the plugs side is not auto-connected on install. Store policies may trigger a manual review for uploads of snaps specifying this interface.
- lxd: slot policy allows access for LXD to manage containers. Plug policy allows access to the service implementing the corresponding slot via the LXD admin socket. Both sides require privileged access and the plugs side is not auto-connected on install. Store policies may trigger a manual review for uploads of snaps specifying this interface.
- location-observe: slot policy allows access for the location service to use the GPS. Plug policy allows access to the services implementing the slot. Both sides require privileged access and the plugs side is not auto-connected on install. Store policies may trigger a manual review for uploads of snaps specifying this interface.
- dbus: slot policy allows access for binding to a well-known DBus name on the session or system bus. Plug policy allows access to the services implementing the slot. The slot side requires a snap declaration assertion from the store to grant use of the well-known name. Plug side is manually connected.

[26] Safe snap-specific access to gsettings is planned. When implemented, most applications that use gsettings will not require this privileged global gsettings interface.
[27] Separate audio-playback and audio-record interfaces is planned
[28] When audio-playback and audio-record are implemented, pulseaudio will no longer be auto-connected

Continuous updates

The update process for application, base, gadget, and vendor kernel snaps is different from the system update process because publishers simply upload directly to the store. The publisher will:

- Develop the snap (and for app snaps, bundling any necessary libraries, data and programs to properly run).
- Create an account on Launchpad (if not done already).
- Create a project for this snap (one time only) in the Snap Store tied to the Launchpad account.
- Optionally sign the snap with the developer signature.[29]
- Upload the snap to the store under the appropriate Launchpad account and store. project via https, targeting particular channels[30] and whether or not to automatically publish to a channel if the snap passes review.

The store will then:

- Perform automated checks on the uploaded snap. If errors are found, they are reported to the uploader and the snap is not published.
- If the snap passes automated checks, the store GPG-signs the snap with a store signing key and publishes the snap to the store either automatically or when the uploader triggers it at a later time.

Ubuntu Core systems will do the following:

- As part of the factory install process[31], a unique identifier for the device (eg, a serial number) will be signed by the vendor. The vendor's public key will be given to Canonical and optionally associated with a Dedicated store for the vendor
- On first launch, the device will register with the store by presenting the signed unique identifier to the store and optionally which Brand store to access. The store will verify the signature and if specified, register the device to the Brand store. A device may only be registered with one store.
- Query the public Snap Store or a Brand store via https for any updates.
- If updates are available, download the updated snaps via https.
- Verify the signature of the snap with the store's public key.
- If verification succeeds, the device will review the validation set and will install the snap.

Like with OS and kernel snaps, Ubuntu Core systems are configured by default to automatically install updates for applications and all other snap types, but they can also be scheduled. The 'snap refresh <snap name>' command may also be used at any time to trigger an update. Package rollbacks are supported via the 'snap revert <snap name>' command.

Snap packaging example

To illustrate how snap packaging, security policy, and the runtime environment work together, consider the following complete example. In this scenario, the snap below is uploaded to the Snap Store and is assigned revision 7.

name: foo
```
version: 1.0
```

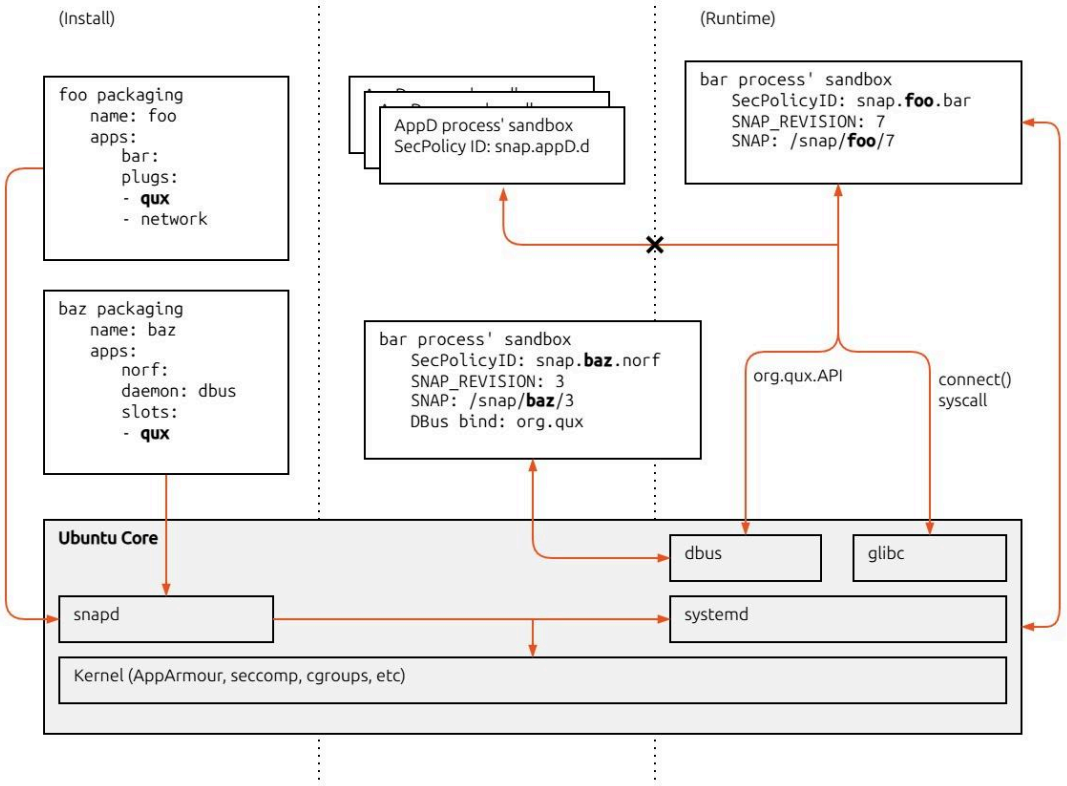
[29] The snap publisher may optionally configure the store to verify the uploaded snap against a developer signature
[30] The publisher may also request the use of store 'tracks' which can, for example, be used to group different upstream releases and channels (eg, an 'lts' track and a 'current' track, each with their own stable, candidate, beta and edge channels)
[31] This describes the store interactions for Ubuntu Core devices. The old Ubuntu Core 15.04 factory process differed substantially

```
description: foo does stuff
apps:
  bar:
    command: bin/bar-service
    daemon: simple
    plugs:
    - qux
    - network
  ctl:
    command: bin/control
```

With the above, the:

- Security policy ID for ‘bar’ is snap.foo.bar. It has the default security policy with the ‘network’ interface auto-connected.
- Security policy ID for ‘ctl’ is snap.foo.ctl. It has the default security policy only.

For both of the above commands, the runtime environment variables are set to SNAP_REVISION=7, SNAP=/snap/foo/7, SNAP_DATA=/var/snap/foo/7, and SNAP_USER_DATA=$HOME/snap/foo/7. Note that the store revision is used in the snap-specific directories instead of the version number declared in the snap packaging. How snap packaging, interfaces and the runtime sandbox work together can be seen in the following diagram:



Snap packaging, interfaces and the runtime sandbox

The ‘foo’ snap plugs into the ‘qux’ interface and ‘baz’s ‘norf’ service implements the ‘qux’ slot. After connecting the snaps with ‘snap connect foo:qux baz:qux’, the security policies are configured to allow (by seccomp filter) ‘bar’ to connect() to ‘norf’s DBus service at org.qux (by AppArmor policy).

Update procedures

Updates or modifications to some components or properties of the system may lead to a change in the digests that are measured to PCRs, and therefore require an update to the PCR policies (see Authorization policies) for the disk unlock keys associated with the system.

The general process for performing updates that require PCR policy changes is as follows:

- Compute and authorize a new PCR policy that includes composite PCR digests for the currently supported boot configurations, plus composite PCR digests associated with the updated component (see PCR policy updates and Computing PCR values).
- Replace the key data with one containing the updated PCR policy.
- Commit the update to the system.
- Compute and authorize a new PCR policy that removes support for boot configurations that have been made obsolete by the applied update.
- Replace the key data with one containing the updated PCR policy.
- Revoke previous PCR policies (see PCR policy revocation).

Kernel updates

Kernel updates for the run mode system are performed in a way that facilitates rollback to the last known good kernel if a kernel update fails. The process for performing such an update is as follows:

- Compute and authorize a new PCR policy that includes composite PCR digests for the currently supported boot configurations, plus composite PCR digests associated with the new kernel (see PCR policy updates and Computing PCR values).
- Replace the key data with one containing the updated PCR policy.
- Configure the system to try the updated kernel on the next boot.
- On next boot, verify that the system can be booted successfully with the updated kernel and that the encrypted containers can be unlocked.
- Fully commit the kernel update by configuring the system to always use the updated kernel on subsequent boots.
- Compute and authorize a new PCR policy that removes support for boot configurations that include the old kernel.
- Replace the key data with one containing the updated PCR policy.
- Revoke previous PCR policies (see PCR policy revocation).

PCR policy update triggers

PCR policies are updated when any of the following components are modified:

Component	What changes?	Why?
Shim	PCR4	Shim digest changes
	PCR7	CA certificate used to verify shim changes
		CA certificate used to verify GRUB and the kernel changes because of a change to shim's vendor certificate
GRUB	PCR4	GRUB digest changes
	PCR7	CA certificate used to verify GRUB changes
Kernel	PCR4	Kernel digest changes
	PCR7	CA certificate used to verify the kernel changes
Kernel command line	PCR12	Kernel's EFI stub loader measures the command line
UEFI authorized and forbidden signature database content	PCR7	These are measured by the platform firmware
Model assertion	PCR12	This is measured by the initramfs

Conclusion

Ubuntu Core provides a modern, application-centric operating system model for embedded and IoT devices. By separating the base system from applications, using an immutable image-based design, and delivering everything as snaps, it improves predictability and reduces the operational risk associated with traditional package-based upgrades.

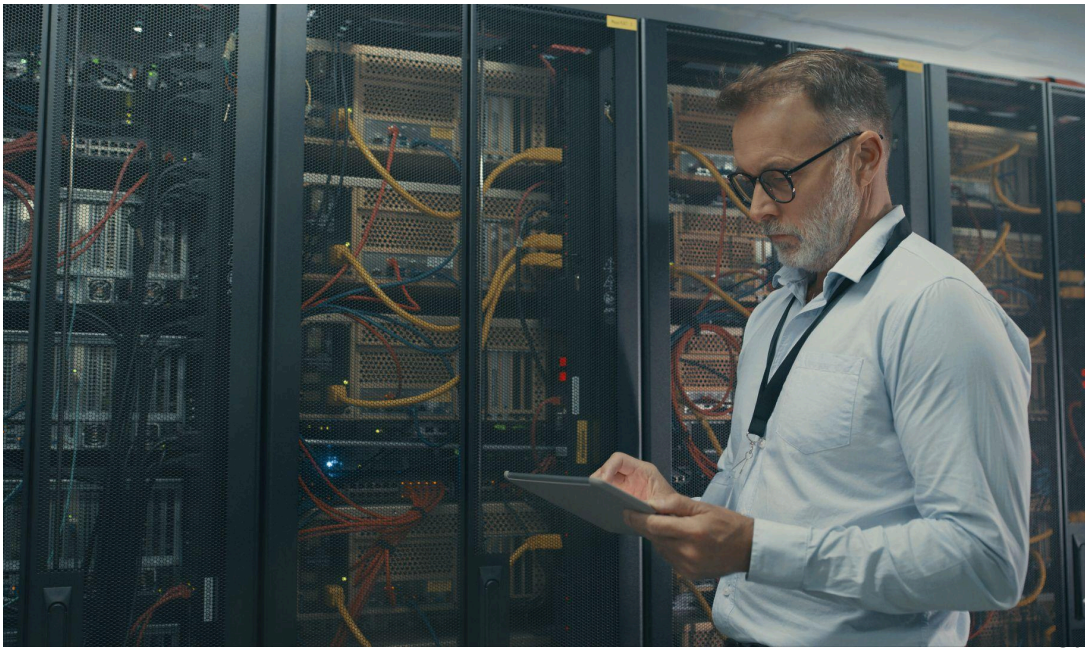
The platform's update and rollback mechanisms support unattended operation while preserving reliability. Kernel, OS, gadget, and application snaps can be updated independently, and the system retains prior revisions to enable automatic fallback when an update prevents successful boot. This helps vendors keep fleets current without turning routine maintenance into bespoke engineering work.

Security in Ubuntu Core is implemented as a set of mutually reinforcing layers: verified boot, measured boot, and TPM-backed full disk encryption protect confidentiality and integrity at startup; model assertions and key-binding extend trust decisions to the runtime OS; and confinement (AppArmor, seccomp, namespaces, and cgroups) limits application behavior at runtime. Store review processes and interface policies add an additional control plane, enabling publishers and device owners to manage privileged access explicitly through well-defined, auditable connections.

Together, these mechanisms make Ubuntu Core well-suited for products that require controlled software distribution, resilient lifecycle management, and strong security boundaries—while still enabling developer velocity through reproducible packaging and well-scoped interfaces. Canonical's pre-built images, kernels, and OS snaps provide a supported foundation, and vendors can partner with Canonical for tailored kernels, gadget snaps, and store branding to meet device-specific requirements without compromising the system's core guarantees.

Further resources

Main site: https://ubuntu.com/core/
Documentation: https://documentation.ubuntu.com/core/



Canonical