Kry10 Technical Overview

Kry10 provides a secure operating system and other software for smart machines and other connected industrial systems. Built on a secure foundation which is mathematically proven to be functionally correct, the OS removes many classes of vulnerabilities that most common operating systems are susceptible to. While software backed by formal methods has been available for several decades, formal methods have not been adopted at scale because they weren't usable.

The Kry10 suite of developer tools and libraries has solved the usability problem. Developers can now code on a secure OS easily and efficiently. The Kry10 tools include familiar languages and development tools:

- Support common languages such as Rust and C
- Include build systems and libraries of reusable, attestable code.

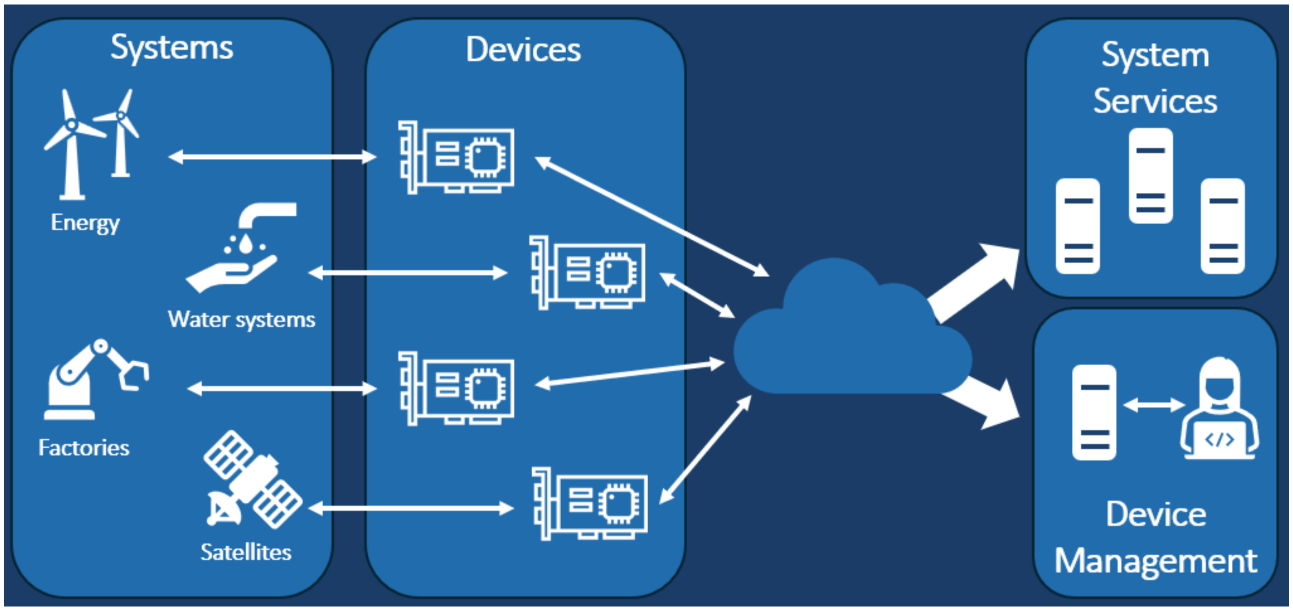
Built for Today's Connected Systems

Connected industrial systems consist of collections of smart machines and services that manage many types of industries such as factories, energy plants, construction sites, farms, and hospitals. The administrators of these systems often manage these systems remotely, including the following tasks:

- Automate and control system operation.
- Collect and analyze operational data.
- Monitor status of the system and operations.
- Perform system and smart machine maintenance.

The following diagram shows a typical connected industrial system, which consists of the following components communicating in the cloud:

- Devices that control or monitor the industrial process.
- A service to manage those devices.
- A collection of other external services that interact with the devices to control them, collect data from them, and process that data (for example, to perform analysis or the data).

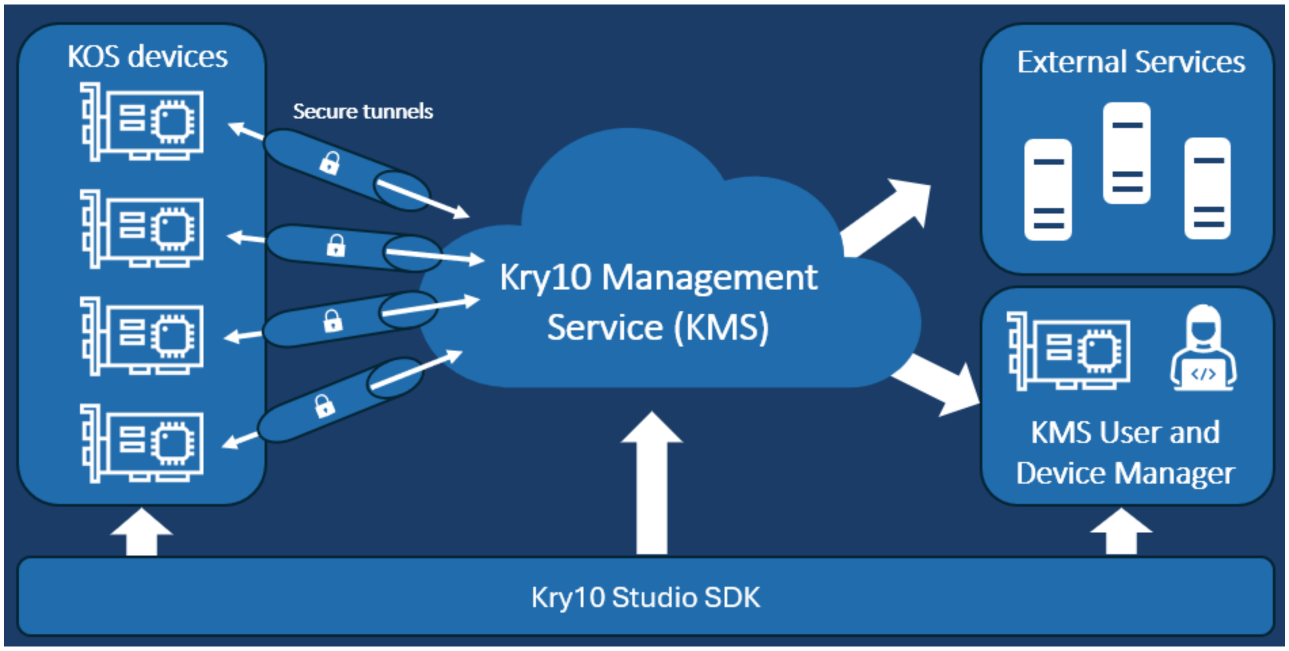


Connected industrial systems

Kry10 provides a secure operating system, services, and tools that support the lifecycle of a connected industrial system. While the focus for the Kry10 OS is the device and device management service, it also provides the software infrastructure needed to access the external services that the devices communicate with.

Kry10 Product Overview

Kry10 consists of three products. The following diagram shows how they work together to deliver on Kry10's trust and adaptability goals.



Kry10 operating system (KOS) – The base OS that runs on the device. It provides a platform for device-specific applications to run on the device and primarily supports the operation phase of a device deployment. The software implements services for the following:

- Boot and code loading.
- Device startup and configuration.
- Hardware access (through device drivers).
- Communication, user interface, software upgrade, and device management.

Kry10 Management service (KMS) – A device management service that can be run on development machines, on on-prem servers, or in the cloud. It supports the following functions:

- Deployment, operation, upgrade, decommission, and repurpose phases of device deployment.
- Tracks deployed devices, their properties, and their status.
- Real-time monitoring of devices through remote console access, collection of trace data, and remote device UI display.
- Allows administration of devices by monitoring and tracking the software running on the device and performing remote update of the device software.

Communication between the device and KMS is through a secure tunnel that is provided by KMS. Communication with external services also goes through the tunnel to KMS and is then forwarded to specific services by an KMS-managed proxy.

Kry10 Studio – Kry10 Studio is a software development kit (SDK) that provides tools to support Kry10-based systems throughout their lifecycle phases. For example, providing development tools and libraries, as well as tools to deploy, manage and update devices through KMS.

These product offerings are described in more detail in the following chapters.

Chapter 1: Kry10 OS (KOS)

KOS is an operating system for smart machines. It is a fundamental part of Kry10 and provides the lowest layer of software to allow applications to run on a device.

1.0 KOS Goals

The key goals of KOS are:

- **Secure**. Provide strong isolation between the components running on a device. Give components limited privileges and enforce these limits when accessing other components or resources. This approach reduces the attack surface of any component and limits the damage that could be done if any component is compromised.
- **Reliable**. Provide strong isolation and rapid error recovery at several system layers, reducing the impact of hardware and software failures.
- **Manageable**. Provide mechanisms, interfaces, and services that allow the system software and components to be easily managed. This focus includes configuring the system at development and deployment time, analyzing the system (both offline and online), controlling the system at runtime, and updating the system at runtime.

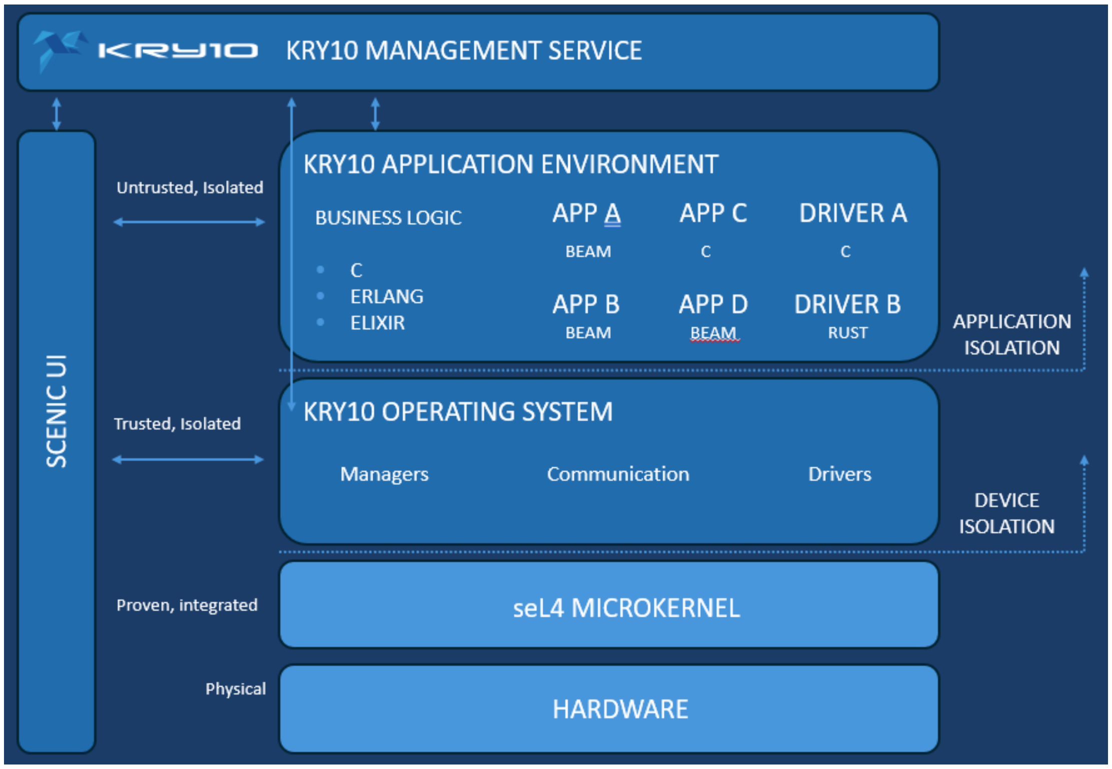
- **Performant**. Create minimal overhead for applications that run on KOS, making it suitable for resource constrained devices. Also provide transparent resource usage, so developers are aware of the exact resource usage and requirements of the systems running on KOS. This goal means developers have assurance that their software can successfully run on target devices and not run out of resources.

1.1 Principles of KOS design and implementation

- **Isolation**: Isolate independent components from each other. This principle includes spatial isolation (that is, components cannot access another component's memory or resources) as well as temporal isolation (that is, components cannot affect each other's execution time). When components are isolated, it is then possible to consolidate independent functionality of the same or different criticality levels on a single device without increasing security and reliability risks.
- **Explicit access control**. Components can only access resources or communicate with other components if they have been explicitly given permission to do so. All of the possible privileges in the system are known so there are no unexpected accesses. It also limits the flow of information and control in a system.
- **Least privilege**: Components do not have more privileges than needed to perform their functions. For example, a component does not have access to a hardware device if it does not need to directly control that device, or a component cannot communicate with another component if that is not necessary for its function. This principle limits potential damage caused by misbehaving.
- **Specification**: The system components, resources, and their privileges are explicitly defined in a specification. The specification is used when configuring and running a system and can be analyzed to determine its (and the resulting system's) properties, such as resource usage, isolation, information flow, control flow, software versions, and more.
- **Supervision and Recovery**: System components are monitored at runtime and if they crash for any reason, a supervisor can detect and restart them and other components that depend on them. This principle allows components to take a 'crash first' approach to dealing with unexpected or unrecoverable errors. Restart policies must be configurable so that different components can use different recovery strategies.

1.2 KOS Architecture

A device running a KOS-based system consists of four key parts as shown in the figure below.



High-level structure of a KOS-based system

- **Hardware**: The physical device or machine that runs KOS. For the operating system, a device is its CPU or SoC and its attached devices such as sensors, actuators, gadgets, media peripherals and connectivity interfaces. The hardware is a critical building block to design a KOS-based system. Hardware options must be evaluated against design requirements in terms of what it provides in terms of the level of performance, the security primitives, the supported peripheral devices, the connectivity options, and the environment it will be deployed in, including whether physical device hardening is necessary.
- **seL4**: KOS leverages the seL4 microkernel to implement its operating system primitives. seL4 is a high-assurance, high-performance operating system microkernel, unique for its comprehensive formal mathematical verification. It is the key for providing guarantees about isolation and access control in a KOS-based system. seL4 is the only part of the system that runs in the processor's privileged kernel mode – all software in the layers above seL4 runs in unprivileged user mode.
- **Kry10 Operating System (KOS)**: KOS consists of a collection of sub systems and service components that instantiate and manage applications, provide runtime environments to application-specific components, manage critical functions such as inter-process communication and scheduling, manage hardware peripherals, and provide typical OS sub-systems such as a network stack and file system.
- **Applications**: Application-specific components, services, and drivers work together to implement a real-world function. For example, factory automation, medical equipment monitoring, autonomous driving, or environmental monitoring. These components are distinguished by the functionality they provide – for example, business logic vs hardware drivers – and their runtime environment – for example, BEAM, C, Rust, or VMs. A KOS device can run multiple independent or interdependent applications simultaneously, relying on the underlying isolation to prevent these applications from interfering with each other. Most system developers develop software that runs at this level.

1.2.1 seL4 microkernel

seL4 serves as the trustworthy foundation for building safe and security-critical systems. seL4 is a microkernel, which means that it is a minimal operating system kernel running on the hardware in the CPU's privileged mode. Running in privileged mode means that it has access to all the CPU functionality and can access all the system's memory and peripheral devices and other resources. seL4 is the only software that runs in this mode, with all layers above seL4 running in non-privileged user-mode where they have more limited access to the CPU functionality, system memory and other resources.

As a microkernel, seL4 is designed to have minimal and well-understood functionality that can be verified and optimized. Its minimal core is free from policy, forming a dependable base for flexibly building arbitrary systems. Its limited system call layer leaves a minimal attack surface, further hardening the kernel. Resources such as memory, threads, CPU time, as well as inter-process communication primitives are controlled by seL4 through fine grained capability-based access control mechanisms.

The assurances of a proven kernel

seL4 is unique in that the kernel has a formal (mathematical) proof of its functional correctness. This proof gives the following assurances:

- The kernel's implementation behaves exactly according to a formal specification of the kernel.
- The formal specification provides the necessary isolation properties - confidentiality and integrity - required to build secure and reliable systems.

Isolation from a multi-server OS design

seL4's microkernel architecture and unique capability model fundamentally influences the design and implementations of the layers above it. A monolithic kernel, such as Linux, implements most of its operating system functionality within the kernel – all accessible in the same address space and operating in the CPU's privileged mode. Conversely, seL4 forces a significant amount of that functionality to operate at user-level, where operating systems built on top of it, such as KOS, adopt a multi-server OS design.

Adopting a multi-server OS design means the services that KOS provides are separated into separate inter-connected components that are strongly isolated via address space isolation, restricting access to system resources and other services. This approach leads to better security and fault recovery outcomes, as it prevents attacks and failures from spreading throughout the system.

1.2.2 Kry10 Operating System (KOS)

KOS is a collection of sub systems and service components that implement the operating system functionality. It comprises a set of core services that provide the base functionality of the OS and a set of optional services and components useful to build systems.

The runtime configuration of a KOS system is static, meaning that all components and resources are pre-defined ahead of time. Once a static configuration has been started no new applications and resources will be dynamically loaded. KOS can, however, perform system updates where a new static configuration can be loaded in place of the current system, essentially performing a reset of the system's state.

KOS Core Services

KOS core services are critical for the operation of a KOS system, implementing important operating system primitives and functions which support the software layers above it. In line with a multi-server design, each core service runs as its own server, strongly isolated using address space isolation. Also applying the principle of least privilege, each core service is only given access to the system resources it needs to carry out its task.

There are five core services, and all KOS-based systems are expected to run them:

1. **Root server**

   The root server is the first thread started by the seL4 kernel and serves the most critical function of bootstrapping the rest of the KOS system. Once started, the root server does the following:

   - Processes the system specification (called a manifest) and initializes all system resources according to that specification.
   - Loads and starts all other system components (including the rest of the KOS core components, non-core KOS components and application-specific components).
   - After starting the system, the root task acts as the system monitor and component supervisor - detecting when components have crashed and restarting relevant components according to a policy specified in the manifest.
   - Stops and destroys components, and loads and runs new components as necessary (for example, during a system update).

2. **Message server**

   The message server coordinates inter-process communication between components, including the following:

   - Enforces an access control policy (specified in the manifest) that determines how components can communicate with each other.
   - Enables allowed communication by creating communication channels using resources (memory and seL4 communication primitives) provided by the communicating parties.
   - Provides synchronous and asynchronous communication options.
   - Tears down and re-establishes communication channels when components are destroyed and restarted.

3. **Serial server**

   The serial server provides serial driver support for KOS's various supported platforms and acts as a default console for system logging and output.

4. **Clock server**

   The clock server implements a time server, providing driver support for reading real-world time from a reference hardware clock and providing an interface for other components to query the time source and create timeout events.

5. **Scheduler server**

   The scheduler server manages seL4's Mixed-Criticality Scheduler extensions. Only enabled for MCS configurations of the seL4 kernel, the server provides an interface to configure threads with an upper bound on CPU execution time.

KOS Optional Services

Besides the core services, KOS also provides a library of optional services and components, including:

- Runtime environments such as C, Rust, BEAM and VMs
- Device drivers for supported SoC and peripheral hardware
- Cryptographic key store and random number generator
- Secure Boringtun tunnel for communicating off-device
- Storage and communication stacks (for example, TCP/IP)
- Administrative component, responsible for connecting to KMS and performing requests initiated by KMS, including coordinating system updates
- Monitoring, logging and debugging functionality

1.2.3 Applications

The applications that run on KOS-based devices consist of a collection of interacting application-specific components. These components are responsible for implementing the application business-logic, collecting and analyzing data, communicating with external services, managing application-specific hardware resources, and providing a user interface, among other things.

Each application component executes in an *application runtime environment* that includes the programming language used to write the application through to the base OS interfaces and services needed to support loading and executing the application. The application runtime environment consists of a *system environment* and an *application environment*.

- **KOS System Environments**

  KOS provides two system environments:

  - KOS freestanding environment, a minimal environment that provides a limited subset of standard C library features that do not depend on underlying POSIX-like OS services.
  - KOS hosted environment, a more dynamic environment that makes available a complete C standard library along with support for several POSIX subsystems.

- **KOS Application Environments**

  KOS also provides four application environments:

  - **C** – Allows C applications to run on KOS and supports either a freestanding or hosted system environment
  - **Rust** – Allows Rust applications to run on KOS and requires an underlying hosted system environment
  - **BEAM** – Allows Erlang and Elixir applications to run on KOS and requires an underlying hosted system environment.
  - **Virtual machine hypervisor** – KOS also runs as a hypervisor, using hardware virtualization support to provide virtual machines (VMs) that can host virtualized guest operating systems such as Linux.

1.2.4 KOS Hardware Support

KOS currently runs on the following Arm hardware platforms:

- Beaglebone Black
- Nitrogen6_SoloX
- NXP i.MX 8M Mini Evaluation Kit
- Arduino Portenta X8
- Compulab IMX8PLUS Industrial Gateway
- QEMU virt arm

1.3 KOS Assurance

KOS provides strong guarantees about its security and reliability properties. To make strong claims about guarantees, Kry10 requires powerful assurance techniques. KOS guarantees are based on a combination of conventional testing and formal verification:

- **Formal verification** uses deductive logic to prove that certain properties are true for all possible executions of a high-fidelity mathematical model of the system.

- **Testing** checks properties for a sample of executions on real or simulated hardware.

KOS is designed to take advantage of the guarantees provided by the seL4 verification, using its fine-grained access control mechanisms to isolate each component from any faults or vulnerabilities in other components. Unlike conventional operating systems, seL4 can even protect the system from faults in device drivers.

This design relies on some core KOS components such as the root server and message server to correctly configure seL4. Currently, these components are assured using conventional testing techniques but are in the process of being formally verified over the next year.

1.4 Multicore Support is in Progress

The most important KOS feature coming in the product roadmap is multicore support.

seL4 currently supports a symmetric multiprocessing (SMP) model of multicore. That means that a single kernel image runs on multiple cores, manages concurrent access to shared kernel state, allows user level threads to run on the available cores, and supports managing which cores threads run on. SMP support in seL4 is not formally verified and KOS similarly does not support running on SMP seL4.

Kry10 is developing an alternative multicore model for seL4: the multikernel model. In the multikernel model, separate instances of the kernel run on the cores with no kernel state shared between the instances – each kernel instance manages its own state and its own kernel memory. User level threads run on a specific kernel instance and do not migrate across cores. User level threads running on different cores can communicate with each other using cross-core notifications and by mapping regions of memory shared between the cores. Verification of the multikernel model is simpler than SMP and is the pathway we are currently pursuing to extend support to many new important use cases requiring this functionality.

Chapter 2: Kry10 Management Service (KMS)

KMS is the Kry10 device management service. It provides tools for managing all aspects of Kry10 device deployments, including device registration and onboarding, device configuration, control and update, deployment management and deployment organization. These functions are provided through either a web-based graphical interface or a remotely accessible API. KMS can be run as a local service, on a dedicated server, or in a public or private cloud.

2.0 KMS Goals

The key goals of KMS are:

1. **Secure.** KMS must ensure that all device access is secure:
    - Mutual authentication and authorization for devices connecting to KMS, so only authorized devices and KMS instances may connect to and communicate with each other.
    - Secure, encrypted, communication channels between devices and KMS, so all communication from a device goes through a secure communication channel and is handled by KMS or forwarded on to other services.
    - All requests and commands are authorized, so only authorized requests and commands are accepted and executed either by devices or by KMS. Only authorized users may access and manage devices.
    - KMS itself must be secure, which means it must be built using secure development practices. It provides secure defaults and is deployed on a hardened platform.
2. **Reliable.** KMS must ensure that device access is reliable:
    - Devices can reliably connect to KMS.
    - Users can reliably manage those devices.
    - Requests and commands are reliably delivered and executed.
    - Support devices that are intermittently connected and are not always online (for example, low power devices that go into deep sleep modes when not used).
    - KMS itself must also be reliable, ensuring that it provides correct functionality and high availability.
3. **Scalable.** KMS must scale in every dimension, including size of deployments, number of deployments, number of projects, number of organizations, and number of users.:
    - Support both small developer instances with only a few devices, and large deployment instances including those with millions of devices in thousands of deployments.
    - Scale with regards to number of users and geographic distribution of the users and deployments.
4. **Usable.** Use of KMS should be intuitive, with supporting documentation providing instruction for more complex activities. This goal applies to the use of KMS through a graphical web interface or an API, as well as when building, deploying, and managing instances of KMS. KMS should also provide sufficient logging data such that problems can be diagnosed and fixed with relative ease.

2.1 Principles of KMS design and implementation

KMS's design and implementation is based on the following principles:

- **Visualize clear model.** Provide a single graphical interface to help users build a mental model of the organization of device deployments and device details. This model must be supplemented by clear instructions and meaningful error messages to ensure that the system is accessible to new, non-technical users.
- **Nudge users.** Guide users to make secure choices, and away from insecure choices. Do not restrict users from performing insecure operations if those are useful in some environments (for example, development and testing), but ensure that these are not the defaults. Users should be sufficiently and prominently warned when performing insecure operations, and that these settings are not missed when moving to production.
- **Synchronous and asynchronous execution.** Support a wide variety of command execution models. Synchronous commands require user interaction during their execution, asynchronous commands are started and executed without further user interaction, event-based commands execute whenever specified events occur, and timed events occur at specific times. Support devices that are not always connected and devices which connect intermittently.
- **Minimal Trusted Computing Base.** KMS should stay out of the deployed system's trusted computing base (TCB). This principle means that users do not need to trust KMS to deploy or manage devices. One example is using signed manifests for updates where KMS does not have access to the signing key. One internal standard KMS uses is whether an attacker can be given full control of KMS but not be able to gain access to the end devices that are connected to KMS.
- **Diagnostics.** Provide as much diagnostic support as possible to help users troubleshoot issues, including application bugs and network problems. Provide the ability to monitor diagnostics and receive alerts when specific events occur.
- **Testing.** Code must be thoroughly and automatically tested. Include unit or end-to-end tests for every feature, and automatically run static application security testing across the code base with every change.

2.2 KMS Functionality

KMS functionality can be split into three categories:

1. Device development and management
2. Deployment management
3. Organization management

2.2.1 Device Development and Management

Device development management involves controlling individual devices. KMS provides support for building a device software package, reasoning about its structure and interfaces through which individual devices can be interactively debugged and updated. This support includes functions for registering, communicating with, monitoring, configuring, and controlling devices.

Registering devices

KMS acts as a registry of devices, maintaining a database of devices that can connect to it. For each registered device, KMS keeps track of administrative information about the device that is used to manage the device, including device keys (used to connect to KMS and its secure tunnel) and the software that is running on the device. KMS provides an interface to register new devices with KMS and update the information stored for each registered device.

Device communication

KMS enables and manages device communication. All network communication from a KOS device is mediated through KMS, so every KOS device must first authenticate with KMS before it can perform any other network communication.

When a device wants to communicate, it does the following:

1. **Establishes a secure encrypted tunnel with KMS** – The process of establishing the tunnel requires the device and KMS to mutually authenticate and authorize each other using a set of device-specific tunnel keys and KMS tunnel keys. The secure tunnel is provided by KMS and is based on Boringtun.
2. **Uses that secure tunnel for all further communication** – Once the tunnel is established, the device creates a command-and-control channel to KMS for all further KMS-specific communication involving device monitoring, update, control, and more. This channel is implemented as a persistent websocket connection to KMS through the tunnel. Once the websocket connection is established, the device authenticates itself to KMS with a separate set of device keys and is then ready to receive requests and commands from KMS.
3. **Uses the KMS proxy service to communicate with external services** – KMS also provides a service that proxies other network connections from the device to external services. When a device wishes to communicate with an external service, it establishes a connection over the secure tunnel to a service-specific port on the KMS server. If KMS has been configured to proxy that port, it ensures that any communication on that port is to a configured service and any replies are sent back to the device over the tunnel. If a proxy for a service has not been configured, then the device cannot connect to that service. An example of an external service that a device might use is an MQTT broker for collecting and processing MQTT messages from the device.

Monitoring devices

KMS allows connected devices to be monitored through the KMS interface. It provides five monitoring options:

- **Connection status** – Whether the device is currently connected or not.
- **Manifest** – A copy of the device’s current manifest to KMS.
- **Serial stream** – Mirrors the device’s serial console output.
- **Trace data** – Collected by the device and delivered to KMS.
- **Scenic UI** – A remote display and input for a Scenic UI implemented by the device.

Updating

KMS allows connected devices to be securely updated. During an update, KMS sends the device a signed *release*, consisting of a new manifest and binary images for the components that need to be updated. The update is signed using a signing key and KOS validates the signature before applying an update. The release is signed independently of KMS (that is, KMS is not responsible for signing the release and does not require access to signing keys). This approach gives additional protection to devices. Even if an KMS server is compromised, it cannot cause unauthorized updates, since it cannot create valid signed releases. KMS provides functions for uploading and registering release files with the service and for updating devices with specific releases.

Configuring

KMS allows connected devices to be remotely configured through KMS's update facility. When a device is updated, it can also be configured by adding a device overlay that specifies extensions and modifications to the update's release manifest. KMS provides functions for uploading and registering overlays and for activating overlays to be used during a device update.

Controlling

KMS allows connected devices to be remotely controlled. It provides four control options:

1. **Trace control** – Trace control turns trace collection on and off.
2. **Remote-IEx** – Remote-IEx provides an interactive Elixir console that connects to BEAM instances running on the device. This console enables interactive control of that BEAM instance. KOS devices have to be built with configuration options that enable remote-IEx, and the BEAM instances have to have network access enabled in the KOS device manifest in order to use remote-IEx.
3. **Remote Scenic UI** – Remote Scenic UI has been mentioned previously as a way to monitor a KOS device, however, since Scenic provides an interactive UI remote Scenic access can also be used to control aspects of the device.
4. **System restart** – System restarts sends a request to the device to restart itself. Note that all KMS commands sent to the device are authenticated and authorized through the secure tunnel. Only an authenticated and authorized KMS service can make use of the tunnel, and only authorized users can use the KMS service. There is currently no finer per-command authorization available.

2.2.2 Deployment Management

KMS manages collections of devices and groups them into *deployments*. A deployment is a group of homogeneous devices running the same software and with similar or related operational goals.

KMS provides interfaces through which you can manage deployments and devices in deployments. You can add and remove devices from a deployment, define releases for a deployment, apply bulk updates to the devices in a deployment, and manage proxies for device access to external services within a deployment.

2.2.3 Organization Management

KMS also lets you manage user access to deployments. You can group several deployments into *projects*, which own "releases" (software packages) that they can be shared between deployments. For example, to support "staging" and "production" deployments, or other phased rollouts.

Projects belong to *organizations*, and KMS can manage multiple organizations. Users are normally assigned to organizations, so this allows users to be granted different access rights to different deployments.

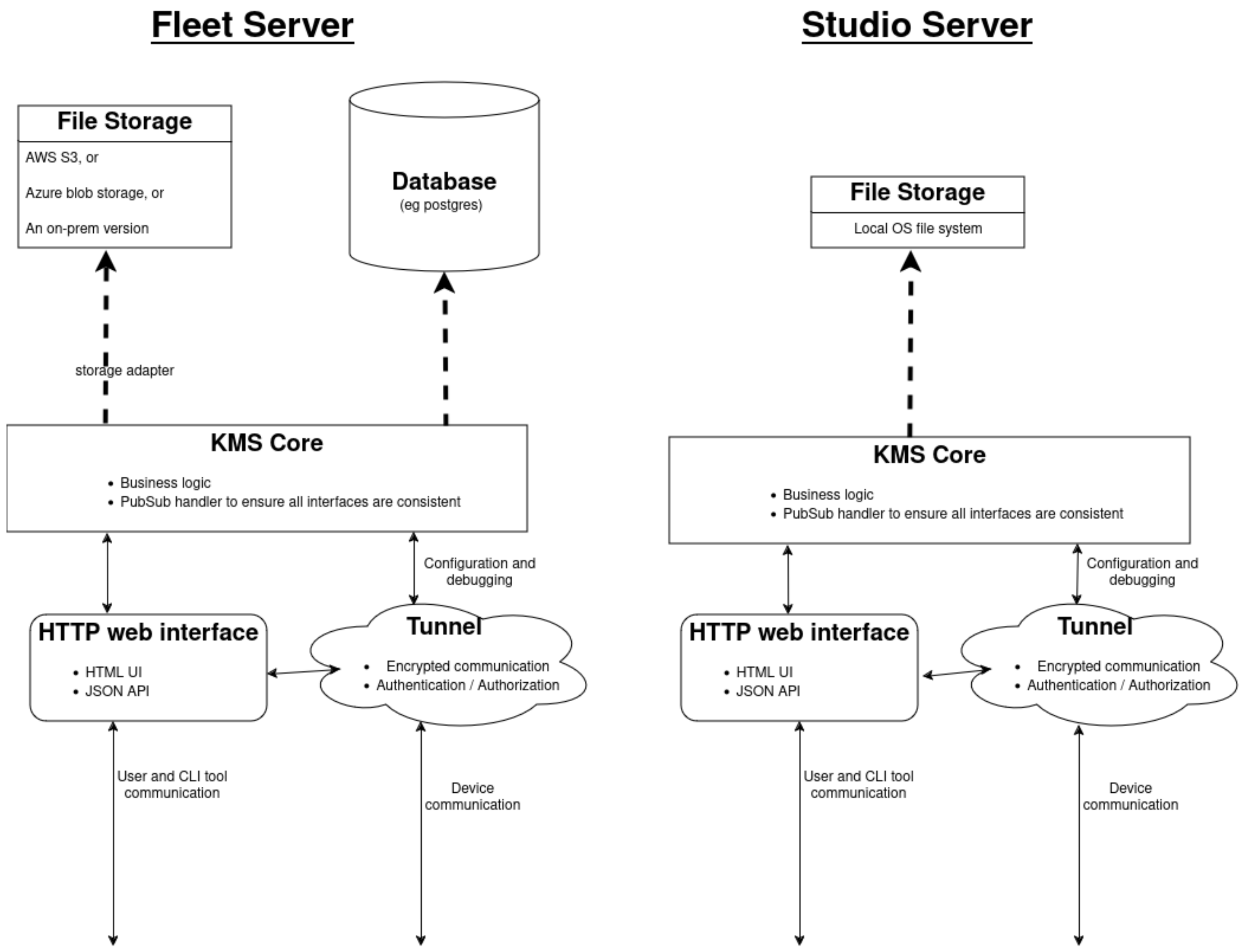
KMS provides three user roles:

1. **Administrators** – Administrators are unique in that they are not tied to organizations, instead having universal access to an KMS instance. The administrator role is used for managing the KMS instance and KMS provides a separate interface for this. Administrator accounts are not intended to be involved in using KMS (for example, to manage devices and deployments) but should be used to control the other users' access levels, monitor KMS status, and configure KMS's tunnel and interaction with the wider network.
2. **Organization owners** – An organization owner has full rights to make any change to their organization, but cannot configure KMS-wide settings or access and modify any other organization. They are intended to be the trusted users involved with day-to-day tasks. Examples of actions an organization owner can take include adding and removing users in their organization, creating projects and deployments, registering releases and devices, deploying releases, and interacting with devices.
3. **Organization users** – An organization user can only view the current state of the organization and is unable to make any changes. This account level is intended to be used for auditors or other people who should only check device state but not directly influence devices. Examples of actions a user of an organization can perform include viewing the releases that have been created, viewing the state of a deployment, and interacting with a Scenic UI.

2.3 KMS Architecture

There two versions of the KMS with a similar architecture, shown in the following diagram:

- **Fleet Server** – Primarily aimed to support managing large deployments in the real world.
- **Studio Server** – For use on a local developer's machine while they build KOS systems.



KMS architecture

The following sections describe these components in more detail.

2.3.1 KMS Core

The KMS Core contains the main business logic of KMS which manages several parts of KMS:

- Manages devices and other KMS entities such as users, organizations, projects, deployments, devices, releases (for the Fleet Server), and projects (for the Studio Server).
- Manages the setting up and maintenance of secure tunnels for devices.
- Maintains network proxies.
- Manages the remote IEx interface (Studio Server only)
- Processes and analyzes device manifests.
- Fleet Server's KMS Core also implements access control, determining which operations users are authorized to perform and which they are not.

2.3.2 KMS Web interface

You can access KMS interactively through a website, or through a web-based API. The web interface is implemented as a Phoenix web server that receives and serves HTTP requests and runs KMS Core functionality as necessary. The KMS UI is built using the tailwind CSS framework. The web interface also implements the websocket control and command channels for communication with devices.

2.3.3 Secure Tunnel

KMS also provides a secure tunnel that is used for communication with devices. The tunnel is based on Boringtun and has been modified for KMS's use. Communication to and from devices goes through the tunnel to the KMS Web interface, which notifies KMS Core as necessary. When devices are added to KMS Core with a public tunnel key, KMS Core automatically assigns a private IP address and configures the tunnel to accept connections from the new device.

2.3.4 Database and File storage

The Fleet Server stores its persistent state in a Postgres database accessed using the Elixir Ecto API. State includes user accounts, registered devices, deployments, projects, organizations, and more. It also stores some release and update-related data, particularly the release files (manifest and binary images) and device overlay files. Due to these files' sizes, this is stored in storage servers rather than in the database. KMS Core accesses these using a generic interface, backed by specific implementations for AWS and Azure depending on how KMS is deployed.

The Studio Server stores its persistent state in the developer's local file system, so that developers do not need to manage a database. This state includes registered devices, device overlays, and paths to locally created projects.

2.4 Running KMS

You can run KMS locally on a developer machine, hosted as an on-prem instance, or hosted in the cloud. You can run KMS independently or access it as a managed service run by Kry10. Separate KMS instances are independent of each other and do not share any data, including user accounts, organizations, deployments, or devices.

KMS instances can be single or multi-tenant. Single-tenant instances host only a single organization and its data. Access to a single-tenant instance is restricted to authorized users who must login and provide valid credentials to access the KMS instance. Multi-tenant instances host multiple organizations with their projects, deployments, and devices. Access to organizations and their data is restricted based on user login.

You can also run KMS as a developer-only instance, in the form of Studio Server. In this case, its functionality is restricted to system development and debugging tools. User accounts, organizations, projects, and deployments do not exist in this version. It is meant to be run on a development machine and used to assist with device development, so it should not be deployed in a way where it is publicly accessible over the Internet.

2.5 KMS Assurance

KMS is not intended to be a critical component in a device's security (that is, it should not be part of a system's trusted security base). Because of this, Kry10 does not assure it using formal methods. However, as a networked service storing and processing data, KMS must still be generally secure and reliable. This goal is achieved through comprehensive testing, including unit tests, end-to-end tests, static application security tests and a web application firewall (for the Fleet Server). Kry10 has plans in progress to provide addition assurance in the future through dynamic application security testing, penetration testing and an external audit.

Chapter 3 - Kry10 Studio

Kry10 Studio provides tools for developers to productively build and manage secure KOS devices with a great developer experience.

3.0 Kry10 Studio Goals

The key goals of Kry10 Studio are:

1. **Devices off desks** – Allow developers to work with devices that are not directly connected to development machines (for example, through a USB or serial connection). Devices can run independent of a development machine but can still be connected to, managed, and monitored over a network connection.
2. **Trustworthy builds** – Builds must be deterministic, reproducible, and free from unwanted interference, providing software supply chain integrity and confidentiality. When a developer deploys a system to a device, they know exactly what software (for example, its versions and origin) will run on it. They have the assurance of integrity; that is, that the software (including dependencies) has not been modified in unknown or unwanted ways during any step of the development and deployment process. They also have assurance of confidentiality; that is, that the information about the software supply chain (that is, exactly what software runs on a device) will not be available to unauthorized parties at any part of the process.
3. **User friendly** – Tools and interfaces are designed with the end user's goals and context in mind. Developers can get tasks completed in the most straightforward ways, without requiring conceptually unnecessary steps or inputs.
4. **Enabling Stability without stagnation** – Provide stable mechanisms and interfaces that allow for incremental and sustained skill mastery, while still allowing for innovation and change. Users are expected to become more proficient with their tools over time as they gain experience with them. Tools must be stable, and their functionality and interfaces do not change significantly over time. It is important to retain this stability while also allowing opportunities for improvement and change when necessary.
5. **Rapid response times** – Provide fast results or fast feedback when results may take longer to produce so that users are not left waiting for results, not knowing if or when to expect them.
6. **Minimal working set** – Writing code is hard and debugging is harder still, so keep the tools simple to free developer's mental load, allowing them to concentrate on what matters. Developers should not need to remember complex sequences of commands, nor need to maintain an understanding of complex models or theory to productively use the tools.

3.2 Kry10 Studio Functionality

The key high-level tasks of Kry10 Studio are:

1. **Develop on Kry10** – Create a *development environment* suitable to developing on Kry10, Kry10 Studio does the following:
    - Downloads a release of Kry10 and installs it locally on a developer's host.
    - Downloads, builds, and installs any required third-party dependencies.
    - Builds and installs relevant KOS components and libraries.
    - Prepares a set of commands for interacting with Kry10 Studio.
    - Starts a shell configured to provide access to all the above so that developers do not have to perform such configurations themselves.
2. **Support KOS system development** – Provide tools, toolchains, libraries, components, and configurations needed to develop KOS-based systems. Kry10 Studio ensures the following:
    - All necessary third party build and configuration tools as well as libraries and other components are available.
    - All relevant KOS libraries and components have been built, configured, and made available.
    - Provides a build system that is responsible for building and composing a KOS system.
    - Provides a set of its own commands and tools to simplify the process of preparing, building, testing, and debugging a KOS system.
3. **Support KOS device deployment** – Provide the tools and commands needed to deploy a KOS system to a device (whether physical or simulated) and interact with that device.
4. **Support KOS device and deployment management** – Provides the tools and commands to manage deployed devices and device deployments. These tools are typically command line interfaces to KMS functionality (for example, configuring and performing system updates and registering devices). Kry10 Studio also provides commands to start and manage a local developer instance of KMS. Besides KMS-based device management, Kry10 Studio also includes some development tools that interact with and manage devices without requiring KMS, such as accessing remote IEx sessions outside of KMS.

Chapter 4 – Working with Kry10

After 5 years of development, Kry10 has a working 1.0 version of these products, including extensive developer documentation. Kry10 remains in technical stealth working largely with national cyber agencies in the US, UK, Germany, and Australia/New Zealand to build evidence for the industry and work on certification. Kry10 software and developer documentation is available to early partners based on an NDA. Please reach out for a meeting if you want to partner with us on building a resilient and secure future.