CHAPTER 4

IoT Software Security Building Blocks

Oleg Selajev from Oracle Labs is famous on Twitter for saying, "The '*S*' in the IoT stands for security."[1] Oleg does not spell poorly; instead, he was bemoaning the sad state of affairs in IoT security. Despite the truth in Oleg's statement, security does not have to be absent in IoT.

Chapter 3 took a comprehensive look at the hardware security offerings in the Intel Architecture. Putting these hardware features to use in an IoT platform requires software. This chapter looks at the software components used to secure IoT systems and how those software components make use of the underlying hardware security features described in Chapter 3.

In this chapter we define a software stack, building on top of the hardware all the way up to the IoT applications, and describe how to put the "*S*" back into IoT. As a way to guide our exploration of software security in IoT, the opening section introduces a generic architectural model that graphically depicts software components of a secure IoT device or gateway. A more detailed section is then dedicated to each component in our model, and we will define the necessary security

[1] www.cnet.com/news/iot-attacks-hacker-kaspersky-are-getting-worse-and-no-one-is-listening/

S. Cheruvu et al., *Demystifying Internet of Things Security*,
https://doi.org/10.1007/978-1-4842-2896-8_4

features as well as how those features contribute to the overall IoT device security. Our architectural model is a generalization of IoT devices, and no generalization is ever perfect; as Alexandre Dumas once said, "All generalizations are dangerous, even this one."[2] Therefore, in Chapter 6, we look at some actual Intel and open source software products and compare them with our generic model.

Due to the breadth of the software topic, this chapter is the longest in the book. For this reason, we have organized the sections so that they do not need to be consumed in a linear fashion, although they do build on one another. Figure 4-1 provides a map of the sections, and the topics covered in each one, including the security concerns discussed. The reader is encouraged to review the figure to find topics that are most relevant or interesting to them. Throughout the chapter, we provide forward and backward references to other sections that may contain additional relevant information, making navigation to the most interesting information a bit easier.

[2]Alexandre Dumas, quote, www.brainyquote.com/quotes/alexandre_dumas_136868

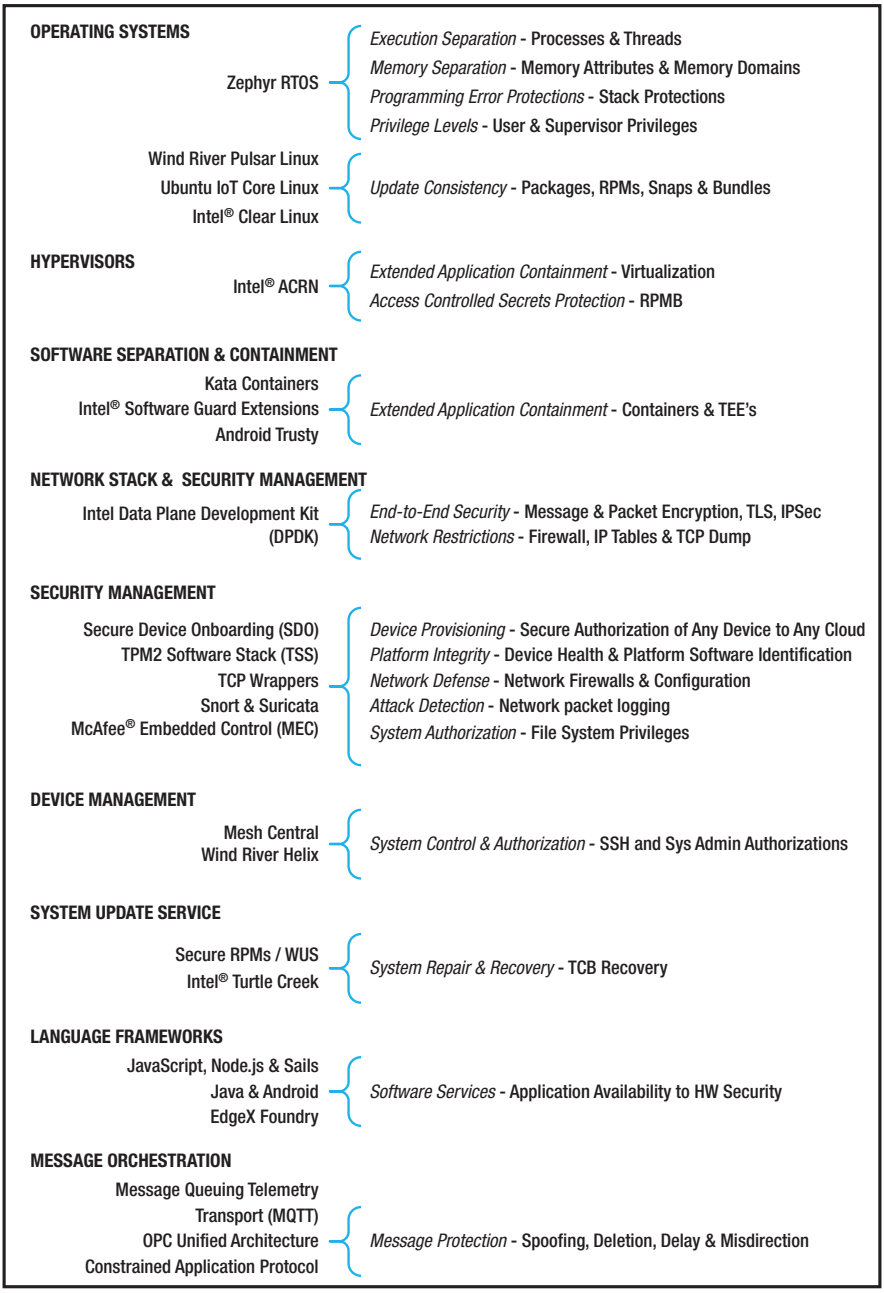

Figure 4-1. *Section outline and security topics*

Understanding the Fundamentals of Our Architectural Model

Before we explore the details of IoT software security building blocks, let us take a quick tour through our architectural model to establish a context for each of the building block components and how they fit together to create an IoT device. Our architectural model is shown in Figure 4-2 and is divided in four quadrants, where each quadrant contains software for a different purpose. Vertically, the figure is divided into platform software, which is the software that creates the *platform environment*, and application software, which is the software that creates the *platform behaviors* of the system. Horizontally, the figure is divided between the management plane, which handles management of the system, and the application/data plane, which is everything else not management related.

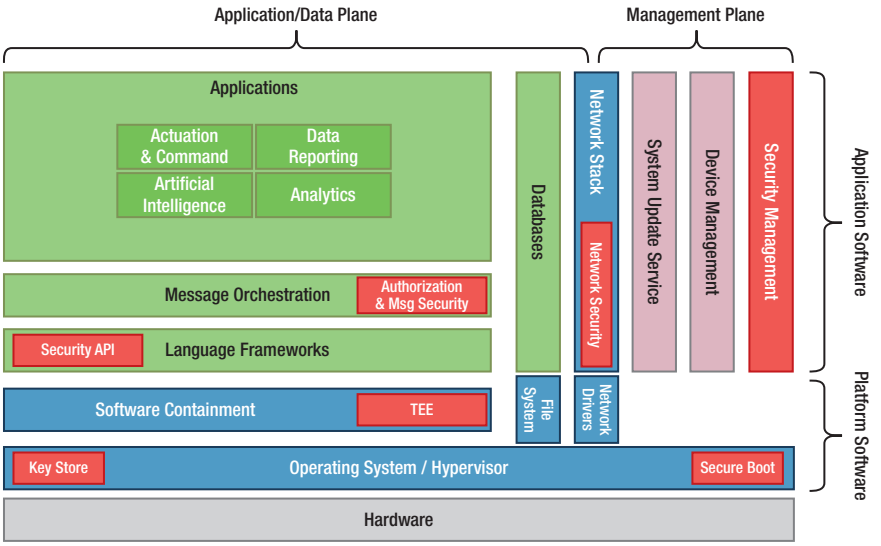

Figure 4-2. *Generic IoT stack diagram*

Beginning at the bottom is the *hardware* covered in Chapter 3. All hardware is implied by this element, including the processor, memory subsystems, flash and other storage, security coprocessors, wired and wireless communication hardware, or anything else physically connected to the processing unit and its motherboard. This chapter does not cover any of these elements, but refers back to the content in Chapter 3 where appropriate.

Directly above the *hardware* is the *operating system/hypervisor* element which is the system software in direct control of the hardware and may be a commercial or open source operating system, or it may be an hypervisor that creates one or more virtual hardware devices for the rest of the software to operate within.

The *software containment* element is optional, but if provided on the system includes technologies like containers and Trusted Execution Environments (TEE). This level of additional containment improves security by reducing privileges and controlling unintended interactions between applications. Both containers and virtualization with hypervisors provide containment. We devote a bit of time to discuss the differences and benefits of each.

Figure 4-2 also shows two components that are not covered individually, but will be interspersed among the other platform software components: the filesystem and the network drivers. These are shown in the diagram to aid in understanding the connection between the application part of the stack and the platform software.

Moving up from the *platform* software to the *application software*, we look at the *management plane*. The *management plane* software is made up of *security management*, *device management*, and the *system update service*. It also includes the *network stack*.

The *network stack* is most often included in the system software or part of the operating system. However, for our purposes, including it in the operating system obscures it and diminishes its importance to IoT systems. The network stack deserves its own separate treatment because it

actually enables a system to communicate with other devices, turning that system into an Internet of Things device. Additionally, the network stack is the entry point for the majority of attacks on IoT systems. It straddles both the application/data plane and the management plane, because it is used extensively by both. It includes communication protocols and network interfaces. The network stack subsection covers software elements needed to secure the network stack, like firewalls and intrusion detection systems (IDS). Chapter 6 is dedicated to covering the network protocols themselves.

Security management is the management software that performs security relevant management operations and used when exercising security management procedures and controls. The functionality in security management includes device identity and attestation, key distribution and certificate management, access control policy, logging rules, configuring and querying the system update service, and policy for network security, firewalls, virus scanners, and host intrusion detection software. Oftentimes these features are included as part of the actual software that performs device management. In our treatment, security management is separate from other management features to highlight adherence to the least privilege principle.[3] Security management features should require a higher level of privilege and additional authentication for an administrator to activate.

The *device management* element includes all the management features that are not part of security management. This includes querying and managing the state of the device, rebooting/restarting the platform, examining and downloading log files (but not deleting log files or stopping logs from being generated, as this is a security management function), starting and stopping and restarting applications, configuring applications, managing databases, and configuring message queues and software orchestration settings.

[3]Saltzer and Schroeder. *The Protection of Information in Computer Systems*. 1975. This paper defines several foundational security design principles which are referred to throughout this chapter.

The *system update service* is the last component of the management plane. While this element is controlled by the security management element (or the device management element in some platforms), it is typically composed of platform and operating system–specific elements in order to update more than just the application software and execution containers on the system. Updates to system and device firmware, boot loaders, and BIOS normally require special software and services to properly coordinate the version dependencies and be able to set the platform into the state where such components can be updated. The system privilege to update firmware and trusted software on the device must be strictly separated from everyday management functions.

The *application/data plane* contains the software that creates the actual behavior of the IoT device. This includes language frameworks, message orchestration, databases, and the applications themselves. Our discussion of these elements is limited, because we focus only on the parts of these elements that leverage hardware security features.

The *language frameworks* contain libraries and services used by application software. Examples of these include the Android framework in Java, Node.js libraries, and the Sails framework in JavaScript.

Message orchestration enables applications on the same platform to communicate, but more importantly enables machine-to-machine (M2M) communications over the network. Protocols like MQTT, message queue, and publisher-subscriber frameworks (pub-sub) like Kafka fall into the message orchestration bucket.

Databases are an important part of IoT systems, as they allow the data that is generated, manipulated, and consumed by IoT systems to be stored, collated, and massaged. There are multiple different types of database systems, including SQL and NoSQL. The types of operations possible on data and the security and privacy of that data are dependent on the database chosen.

The last element of Figure 4-2 is the *applications* themselves. This chapter is not able to cover all types of applications due to the broad diversity of IoT. However, in Chapter 6, several IoT use cases are explored, including a more detailed discussion of the security interactions and trade-offs between the platform and the software that is required to compose a working IoT system.

The next sections will look at each of these IoT software components in varying detail, and in each primary component section, we will introduce security topics relevant to that component.

Operating Systems

When considering software security in any platform, the first consideration should be the operating system. The operating system traditionally is the lowest, most base level of software on any system. It controls what hardware is activated and limits what other software can do. The operating system provides the baseline feature set for all the other software on the platform. If the operating system does not provide some basic feature, or does not allow other software to control or access some aspect of the system (hardware or software), then no other part of the platform can make up for that gap. If a particular security feature is missing from the operating system, then the rest of the software on the platform is likely exposed to significantly more threats. In this section, we take a look at some basic features of operating systems and discuss what security capabilities the operating system should be contributing to the security of the platform. The following is a basic list of security services that an operating system should provide:

- **Execution Separation:** Provides structures and mechanisms to separate different execution units of programs, so that their execution does not interfere with other executing programs; this separation includes processes, threads, interrupt service routines (ISRs), and critical sections.

- **Memory Separation:** Provides mechanisms to separate the different types of memory used by executing programs; this type of separation normally includes process memory, thread-only stacks, shared memory, and memory mapped I/O.
- **Privilege Levels:** Provide structures to separate executing programs into different privilege levels; this separation includes task identifiers for executing programs, user and group identities to own executing programs, and administrator vs. user privilege levels.
- **System authorization:** Provides structures and mechanisms to assign rights to objects and verify the privilege level of execution units against those rights; this includes setting the default privilege level assigned to programs and then enforcing those privileges when programs access system resources, by either permitting or restricting certain operations. This system authorization mechanism allows the implementation of the least privilege principle.[3] In systems with human users, this extends to authentication of users and assignment of privileges to programs under the user's control.
- **Programming Error Protections:** Provide structures and mechanisms to stop errors in executing programs from enabling attackers to manipulate those errors and take over the platform; these typically include stack overflow protection, detection and prevention of heap corruption, and restriction on control flow redirection. All these mistakes result in software attacks that allow a hacker to inject arbitrary code and take over a platform. Control flow protections include protection

from *Return-Oriented Programming* (ROP) and *Jump-Oriented Programming* (JOP) (see sidebar for detailed explanation[4, 5]).

- **Access-Controlled Secrets storage**: Provides mechanisms to store program secrets and prevent those secrets from being accessed by unauthorized users or programs, including the administrator; the system normally provides this through a hardware-backed secure storage.

WHAT IS ROP/JOP?

Return-Oriented Programming (ROP) and Jump-Oriented Programming (JOP) are two techniques used by attackers to create exploit code without having to download large binaries to the target platform. Buffer overruns have been used since the Morris Internet Worm to inject code onto a platform and cause that code to execute.

However, various countermeasures, including DEP (Data Execution Prevention) and ASLR (Address Space Layout Randomization), as well as network defenses that detect and prevent downloads of large binary data, have made such attacks more difficult. Instead of downloading new code, attackers use ROP and JOP techniques to reuse code already on the target platform, allowing attackers to construct their attack code on the fly. Since most software today includes shared libraries, the attacker leverages this to find *gadgets* in software and libraries already existing on the platform and strings the gadgets together into attack code.

[4]Jonathan Pincus and Brandon Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. Security & Privacy, IEEE, 2(4):20–27, 2004.

[5]N. Carlini and D. Wagner. ROP is still dangerous: Breaking modern defenses. In USENIX Security Symposium, 2014.

Gadgets are very small segments of code in existing libraries that perform meaningful subfunctions, like moving data into a register or setting up for a system call. Gadgets either end in a *return* statement or a *jump* statement, allowing the attacker to string multiple gadgets together to craft a new control flow, that, overall with many gadgets, accomplishes their evil task. ROP uses return statements, while JOP uses jump statements. Both are effectively the same attack.

Choosing an operating system for an IoT platform *is* primarily about choosing the one with the best services that also executes reliably on the chosen platform hardware. The capabilities provided by the underlying hardware often affect what the operating system is capable of providing. Some operating systems are designed for servers, or even specifically for cloud deployments, while others are designed to be used in the smallest IoT devices. Small devices typically do not have the computing power or hardware features necessary for an advance operating system to execute. Operating systems designed for low-power processors typically do not have a rich set of services, because the power and performance budget available on the processor just will not support it. CPUs in constrained devices might not have a full memory management unit (MMU) with advanced features like total memory encryption (TME) or memory integrity technology. These types of features are common in server CPUs. Without these hardware capabilities, the operating system is left to provide best-effort security services. In coming to a final decision on what operating system to use for an IoT system, it is also important to evaluate the threats to the operating system and what countermeasures the operating system provides to neutralize those threats. You can then determine if the hardware chosen for your device is powerful enough to resist the attacks the device is likely to encounter.

Threats to Operating Systems

Operating systems run at the highest privilege level, with access to nearly everything on a platform. A successful attack on an operating system can garner the attacker complete control of the platform, often with privileged access to other platforms on the same network. Table 4-1 shows the products (not just operating systems) with the most number of distinct reported vulnerabilities, with data accumulated from 1999 through 2018. As this table shows, there are a large number of different attacks on operating systems. In fact, operating systems make up more than half of the top 50 products with the most vulnerabilities. Although there are numerous types of attacks, it is possible to organize operating system threats into threat classes, all of which execute in similar patterns.

Table 4-1. *Products with Highest Reported Number of Vulnerabilities over a 20-Year Period*

	Product Name	Vendor Name	Product Type	Number of Vulnerabilities
1	Linux Kernel	Linux	os	2124
2	Mac Os X	Apple	os	2084
3	Android	Google	os	1925
4	Firefox	Mozilla	Application	1741
5	Debian Linux	Debian	os	1670
6	Chrome	Google	Application	1546
7	Iphone Os	Apple	os	1495
8	Ubuntu Linux	Canonical	os	1123
9	Windows Server 2008	Microsoft	os	1110
10	Flash Player	Adobe	Application	1060

[6] www.cvedetails.com/top-50-products.php. Retrieved 9 September 2018.

Attacks typically follow a common pattern, called a cyber kill chain®, shown in Figure 4-3, where an attacker executes a series of steps to compromise a target. The attacker begins by observing the target (Step 1), and then deciding how to attack the system, by fashioning some type of weaponized code (Step 2).

The weaponized software might be a program that runs from a web server or a crafted response packet in a protocol. The attacker delivers the attack in Step 3, which might entail a spear-fishing email, or hijacking a network connection, or injecting spoofed packets for a protocol. The actual attack occurs in Steps 4 and 5, and those steps can be iterative, where the attacker pivots from one compromised application or piece of software and uses that as a base to attack another piece of software or system service. Each pivot intends to increase the attacker's control of the platform or penetrate deeper into the network in order to gain complete control of the platform and the entire system.

With the background of the cyber kill chain in mind, we will review different classes of attacks on an operating system and describe how these attacks demonstrate an attacker pivoting progressively deeper into a system, as one attack builds on another. The following five items represent the common attack pattern used in Step 4, exploitation:

- **Fault Injection**: A fault injection creates or forces an execution fault in a process or thread; part of the responsibility for this threat rests on the applications themselves, but because fault injection is the first step to overcoming the operating system itself, the OS must take some responsibility to protect against the vulnerabilities that create this threat. The operating system uses containment to prevent these types of threats from growing into greater threats to the platform, but usually allows the fault to stop the execution of the attacked process or thread. From our

basic list of security services, the operating system uses **programming error protections**, including control flow protections and stack smashing protections, to mitigate this threat.

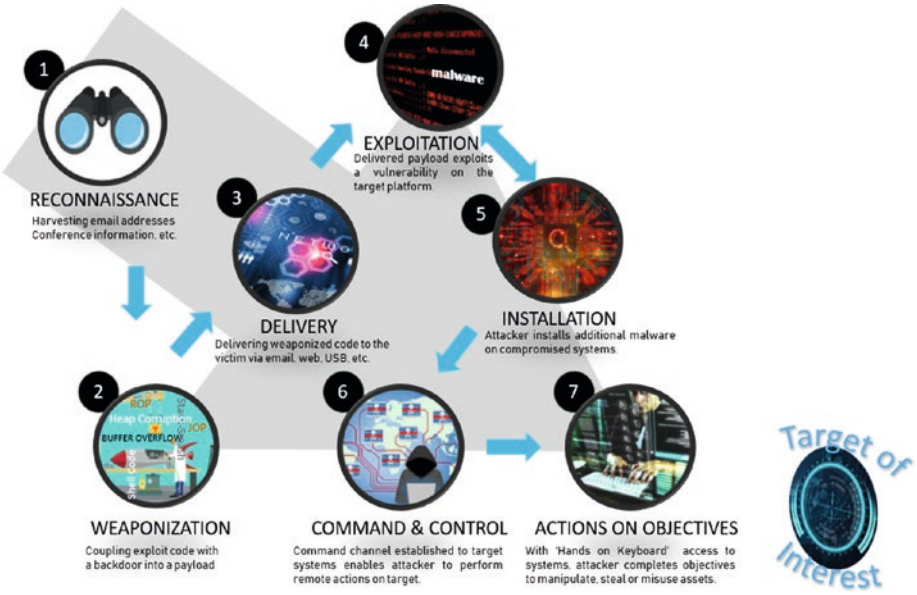

Figure 4-3. *Cyber kill chain*[7]

- **Arbitrary Code Execution**: Arbitrary code execution is the injection of an attacker's code into a process or thread on the platform, causing the injected code to run in place of the existing process or thread, effectively taking on that process or thread's identity

[7]Cyber Kill Chain Diagram, www.lockheedmartin.com/en-us/capabilities/cyber/cyber-kill-chain.html

and authorizations. Arbitrary code execution clearly violates **execution separation** by allowing unauthorized code to corrupt an execution unit, but also violates the **memory separation** guarantee of an operating system, by allowing what should be data to corrupt the code executed by the platform. If fault injection succeeds, either because the application mitigations were not effective or the operating system did not provide any protections against fault injection, then the typical escalation of a fault injection is arbitrary code execution. An attacker places code into the data used to trigger the fault and constructs the fault injection to force execution of, or redirection to, the injected code as part of the fault. Buffer overflows and heap corruption are common mechanisms used by attackers to create arbitrary code execution exploits.

- **Breech of Containment**: Breech of containment is code in one execution unit observing or interfering with the code or data in another execution unit. Once an attacker has achieved arbitrary code execution, the next step is to leverage that power to extract data or further corrupt other execution flows within the platform. Side-channel attacks are a common mechanism used by attackers to extract data and observe program execution. Side channels are so dangerous because they allow a lower-privileged execution unit to observe a higher-privileged execution unit, potentially extracting secrets like passwords and cryptographic keys from those other execution units. These attacks violate **memory**

separation by allowing one program to view or infer data from another program; oftentimes, the way a program breeches the memory separation is through attacks on the **execution separation**. A common example of this execution separation breech is speculative branch prediction, although there are other examples as well.

- **Escalation of Privilege**: Escalation of privilege is overcoming the operating system's authorization mechanisms or code that is able to assume a level of privilege in the operating system that should not have been allowed. After breeching containment and extracting secrets from other execution units, an attacker can leverage those secrets to assume a higher privilege level. In some cases, it is possible for the attacker to inject a fault into the operating system itself and force it to grant a privilege that should not have been given to the attacker's code unit. In both cases, the attacker has escalated the privileges that the operating system grants to the attacker's process. This escalation violates the expected behavior of the **system authorization** mechanisms.
- **Rootkit**: A rootkit is malware that penetrates into the operating system itself and subsumes some of its operations. Following arbitrary code injection, an attacker can chain subsequent arbitrary code injections, containment breeches, and/or escalation of privilege attacks to eventually inject the attacker's code into the operating system itself. In some cases, the attack is a simple one-two chain; in other cases, it may be a series of more complex actions. If the

> attacker can then modify the operating system code on disk or in flash, the attacker can remain permanently on the system. Once an attacker has achieved this level of penetration into the system, it is often extremely difficult to remove the attacker from the system without a complete rebuild of both the software and firmware on the device. With rootkit access, an adversary can normally overcome even the **access-controlled secrets protections** provided by the platform, making all secrets and execution units on the device manipulable by the attacker. A rootkit can actually change the behavior of the operating system, by modifying access control decisions, hiding execution units, and reducing or removing memory protections between different execution units through changes to page table allocations.

As this list illustrates, one of the most basic threats to a computing system is *code and data corruption*. The cyber kill chain outlines the attacker's steps to take over a system, which usually involve a chain of attacks escalating an attacker's position from injecting code into a single application, to interfering with another running application, to eventually changing the entire operating system's behavior. The importance of code and data corruption protections cannot be overstated. Extrapolating from Turing's theory of computation, given enough time, modifications to code can result in serious consequences, as has been demonstrated by various academic papers on ROP and JOP.[8]

[8]Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. On the expressiveness of return-into-libc attacks. In Recent Advances in Intrusion Detection, pages 121–141. Springer, 2011.

In the following sections, we examine several operating systems used in IoT systems and discuss the security features available in those products. Rather than repetitively inspect the same features on several operating systems, we select different security topics on each operating system to inspect in depth. However, for each operating system, we provide a summary to review their protections, the mitigations they have chosen, and their shortcomings.

Zephyr: Real-Time Operating System for Devices

The Zephyr operating system is an open source OS designed for constrained devices running on microcontroller units (MCUs) or in other minimalistic environments. Zephyr runs on many different chips and architectures, including Intel® x86, ARM® Cortex-M, Tensilica® Xtensa, and others. Many IoT devices at the edge utilize these small processors with limited memory. The Zephyr documentation can be found at http://docs.zephyrproject.org/.

In this section, we want to focus on the basic operating system responsibilities of containment and privilege. Since an RTOS is severely limited in what it can provide, these most basic features comprise almost all of what an RTOS can offer. Since Zephyr may not be familiar to most readers, it is an interesting OS to explore, and Zephyr's simplicity makes it easy to highlight the limits of these protections and where usages can go wrong.

Zephyr, like most real-time operating systems (RTOS), is built as a single monolithic binary image; this means that both the operating system and the applications are compiled into one binary that is run on the platform. But unlike most other RTOS systems that were designed purely for size and performance requirements, Zephyr's documentation states that during design, careful thought was put into the security of the operating system. Figure 4-4 shows the Zephyr operating system decomposed into application code, OS services, and the kernel. The next

few subsections will review how Zephyr operates and compare the security architecture[9] against the security properties that an operating system should exhibit. Zephyr version 1.12.0, which is the most current version as of this writing, is used for this review.

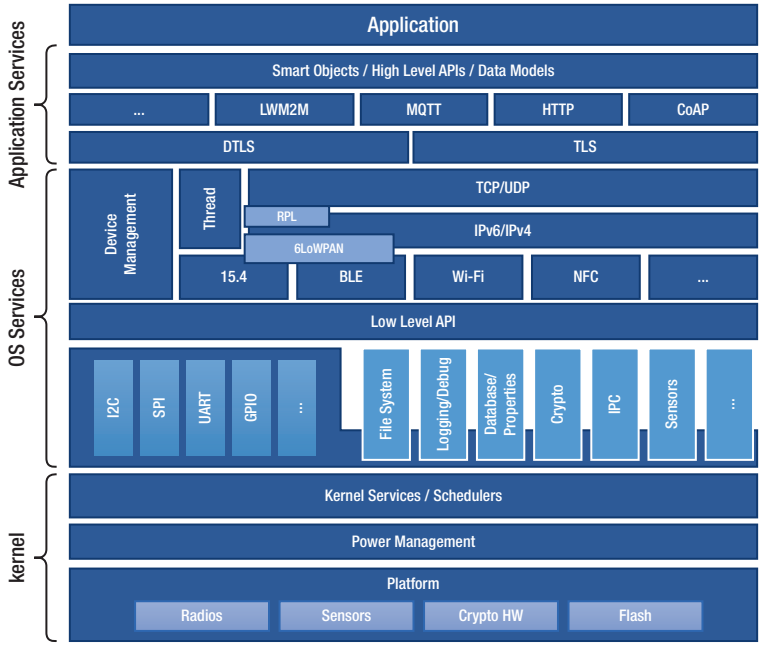

Figure 4-4. *Zephyr system architecture*[10]

Zephyr Execution Separation

Even though the Zephyr OS and the applications are built into a single binary, the OS still provides **execution separation**. In Zephyr, the primary execution unit is a thread. An application is composed of multiple threads that run forever in an endless loop. The application is defined and built

[9] http://docs.zephyrproject.org/security/security-overview.html

[10] Zephyr System Architecture Diagram, http://docs.zephyrproject.org/security/security-overview.html

at compile time using `CMake` and `make`; the system's threads are defined at compile time or can be created dynamically at Runtime. Each thread is separated from other threads in time and space.

The Zephyr OS separates threads in time through scheduling, and the OS saves and restores thread state automatically when threads are put to sleep. Scheduling of threads is organized through a hierarchy of priorities, allowing more important threads to preempt lower-priority threads, ensuring that the most important jobs are completed without interruption. Each thread is scheduled by the OS according to its priority. The highest-priority threads are *cooperative threads* whose priority is set to a negative number. Cooperative threads run until completion or until they voluntarily yield the processor using `k_yield()`. *Preemptive threads* have a positive priority value and are given a certain amount of time to run or are preempted when they perform an action that makes them not ready to run, like waiting on a semaphore or reading from a device or file. Cooperative threads must cooperate with the system and yield back to the OS so other things can run; if they misbehave, they can starve a system and force even higher-priority threads (threads with a numerically lower priority value) from running. Cooperative threads should only be used for high-priority tasks that cannot be interrupted. If a cooperative thread has a long operation to execute, it should break up the long operation into smaller pieces with a call to `k_yield()` at a convenient point. `k_yield()` returns back to the operating system, and the cooperative thread gets rescheduled if there is a higher-priority thread with something more important to do. If there is no higher-priority thread waiting, `k_yield()` just returns back to the thread and the long operation can continue.

Zephyr provides other refinements to the scheduling policy, including

- `k_sched_lock()` and `k_sched_unlock()` to define critical sections in preemptive threads, temporarily preventing them from being preempted.

- `k_busy_wait ()` which prevents a cooperative thread from being preempted when it performs some type of wait action that would make it unready and would normally cause it to be preempted.
- `CONFIG_METAIRQ_PRIORITIES` which is a configuration setting to define the numerically lowest cooperative thread priorities, making them act like IRQs and actually preempt other cooperative threads.
- Threads can change their thread priority, or another thread's priority, to a higher priority level (lower number numerically), even changing it from a preemptive thread to a cooperative or Meta-IRQ thread, if they are executing with privileges.

In addition to thread execution priorities used to enforce time separation of threads, Zephyr assigns a thread privilege to each thread. There are only two privileges, supervisory and user. By default, threads are assigned the supervisory privilege. This gives threads the ability to see all devices and access all of memory. A thread can drop its supervisory privilege and become a user-privileged thread by calling `k_thread_user_mode_enter()`, but once becoming a user-privileged thread, it cannot regain its supervisory privileges. Threads can temporarily perform an operation at the user privilege by spawning a new thread to perform the task and setting that new thread's privilege to the user privilege level.

Operating all or many threads at the supervisory privilege level is dangerous, since all of memory is exposed to those threads, even sensitive memory used by the kernel. User-privilege threads should be used as often as possible because Zephyr provides memory separation for user-privilege threads. Memory separation for user-privileged threads is discussed in the next section.

Since all of Zephyr's applications and libraries are enumerated at compile time, and there is no dynamic loading of applications or dynamic linking of libraries or other code, Zephyr reduces the attack surface created by interfering applications and library code conflicts.

Why does all this matter for security? Creating threads at the right privilege level is important for a system to remain stable in the face of an attack. If all threads are running at the supervisory privilege level, an attacker only has to find a single thread that it can attack via a buffer overflow and then gain control of the whole system. An attacker with control over a supervisory thread can see all memory, halt other threads, or modify stack values to create gadgets for ROP and JOP attacks, allowing the attacker to create their own programs with new, potentially destructive, functionality.

But even if user-privileged threads are enabled, if the right segmentation of memory partitions is not performed, user threads will be able to corrupt each other's memory partitions.

If user threads are enabled and restrictive memory partitioning is used, this will severely limit the types of attacks a remote adversary can perform. This is especially true if the threads that access the network and perform the bulk of the work on the system are user threads. But even if an attacker cannot gain access to an administrative thread, if they can take over a high enough privileged user thread, then by using `k_sched_lock()`, the attacker can starve out other threads. This situation can be mitigated by using the system's watchdog timer or even creating your own watchdog thread at the Meta-IRQ level to monitor and correct misbehaving threads. A detailed discussion of this is found later in the "Security Management" section.

Zephyr Memory Separation

In Zephyr, all threads have their own stack region, and their state is swapped out when they are removed from the running state. This provides basic (space) separation between threads. However, this protection does

nothing to stop a misbehaving supervisory thread which has access to all of memory; and recall that by default all threads are given supervisory privileges. This means that thoughtful, security-aware design is required to build a secure system with Zephyr.

Zephyr provides user threads to address this problem of too much privilege. Zephyr allows threads to be created as user-privilege threads, or allows threads to drop their supervisory privilege and become user threads. Memory access afforded to user-privilege threads is restricted. User-privilege threads are granted access to a specific set of memory locations by assigning a thread to a *memory domain*. A memory domain contains one or more *memory partitions*. A memory partition is a contiguous segment of memory with defined access rights (i.e., read, write, execute). Thus, a memory partition can be defined as read-only, and another memory partition can be defined as read-write. Both these memory partitions can be added to the same memory domain, and one or more user threads can be assigned to the memory domain. All threads assigned to a memory domain have the same access to that memory. A thread can belong to more than one memory domain. Memory domains can be created at compile time or created dynamically at Runtime.

For x86, the definitions for memory domain rights are found in the Zephyr source tree at `arch/x86/include/mmustructs.h`. x86 allows partitions to be defined as read-only, read-write, read-execute, and even the dangerous read-write-execute. And partitions can be defined to restrict access to user threads, but if a permission is granted to a user thread for a particular memory partition, then privileged threads also have the same access to that memory partition. It is important, then, to structure your applications with as few supervisory threads as possible. This follows the least privilege principle.

Zephyr Privilege Levels and System Authorization

As we already discussed, Zephyr defines two privilege levels: user and supervisor. The previous section discussed the impact privilege levels have on memory access. This section reviews how the privilege levels affect access to logical structures, devices, and files.

Zephyr allows for the construction of various logical structures, including FIFOs, LIFOs, mailboxes, and message queues. These logical structures allow different threads to communicate and share data. All these structures are mapped to memory addresses. This means that access to these structures can be restricted to only certain user threads, but any supervisory thread can access these structures as long as they know the address.

Physical devices, such as USB ports, SPI controllers, I2C interfaces, Ethernet ports, and GPIOs, are controlled by device drivers. Device drivers are accessible via APIs and are not restricted. Any thread merely links to the appropriate header file (i.e., `i2c.h`) and then can access the device. Zephyr does not implement any restrictions or authorization for device access.

Zephyr supports several different filesystems, including Newtron Flash File System (NFFS), FATFS support, and FCB (Flash Circular Buffer). The FATFS is an open source implementation of the well-known *File Allocation Table* (FAT) filesystem from the old PC DOS. The implementation supports creation of a filesystem in RAM, on MMC flash, or through a USB drive. No file permissions are supported on FAT, but read-only, hidden, and system file attributes are supported.

The Newtron Flash File System (NFFS) is a minimal filesystem for flash devices and provides no protections or attributes for files. The source code for Newtron can be found at `http://github.com/apache/mynewt-nffs`.

Since Zephyr does not implement any user or persistent thread identity, no authorization mechanisms are found in the logical structures, device drivers, or for the filesystem. This can represent a security problem if a thread is taken over by an attacker and manipulated to perform malicious actions, since the thread can be modified via arbitrary code injection to access resources it normally would not access, and the operating system enforces few limitations.

Zephyr Programming Error Protections

Zephyr does implement several safeguards to protect threads from being taken over by remote attackers. These safeguards include stack protections and memory protections. The previous sections have discussed the memory protections; this section reviews the stack protections.

Programming errors can create vulnerabilities in software that allow untrusted input to overrun or underrun buffers, writing this untrusted data into memory. Specially crafted inputs can result in buffer overruns or underruns that rewrite elements on the stack, or rewrite code pages in RAM, allowing an attacker to change a thread's flow or the code that it executes. Zephyr implements stack protection to detect overruns on the stack, and then halt a thread to prevent it from executing from a modified stack.

Other protections, like Intel's® Control-Flow Enforcement Technology that protects against ROP and JOP, are not yet implemented in Zephyr, but may be added in the future.

Zephyr's Other Security Features

While Zephyr does not directly provide secure storage, it does provide a few other security additions, including a cryptographic library and API for security modules and TEEs.

Zephyr includes an embedded cryptographic library written by Intel, called TinyCrypt. This can be found in the Zephyr source tree at `ext/lib/crypto`. TinyCrypt includes basic cryptographic functions including

- AES symmetric encryption using CBC, CTR, CMAC, and CCM modes[11]
- Elliptic curve asymmetric cryptography using Diffie-Hellman (DH) or the Digital Signature Standard (DSA)
- HMAC and direct use of the hash function, SHA2-256

Zephyr also includes the latest mbedTLS from ARM, which includes TLS v1.2 (Transport Layer Security) and many more cryptographic functions. Details on mbedTLS can be found on the web site `http://tls.mbed.org`.

Zephyr also includes an API to access a hardware random number generator, based on the processor on the particular board that is being used. This allows access to true hardware entropy if the hardware supports it. If there is no hardware entropy source, an interface to a pseudo entropy function is provided (see /ext/lib/crypto/mbedtls/library/entropy_poll.c).

Currently, the APIs for hardware crypto, Trusted Platform Modules (TPMs) and Trusted Execution Environments (TEEs), are very limited. Future versions of Zephyr are planning to implement APIs for these devices.

[11]CBC = Cipher Block Chaining, CTR = Counter mode, see `https://csrc.nist.gov/publications/detail/sp/800-38a/final`

CMAC = Cipher-based Message Authentication Code, see `https://csrc.nist.gov/publications/detail/sp/800-38b/final`

CCM = Counter with CBC for Message authentication, see `https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-38c.pdf`

Zephyr Summary

Table 4-2 includes a summary of Zephyr compared with our operating system security requirements.

Table 4-2. *Zephyr RTOS Security Summary*

Operating System Security Principles	Grade	Comments
Execution Separation	A	Zephyr provides all the standard separation capabilities of a standard operating system, with flexible application of those structures to address real-time concerns.
Memory Separation	C	Although some memory separation is provided, the ability of supervisory threads to see all of memory is a major weakness. Memory domains provide reasonable protections especially for the class of processors used by Zephyr.
Levels of Privilege	B	Two levels of privilege are common in systems today and even in popular operating systems, like Microsoft Windows, which has access to multiple different rings, but makes use of only two ring levels. There are however examples of extra protections – special supervisory modes and TEEs – that are currently lacking in Zephyr and thus warrant a slightly lower grade.
System authorization	D	Without any real system authorization, Zephyr leaves a significant gap for attacked threads to misbehave. While this is normal in MCUs, improvement is required.

(*continued*)

Table 4-2. (*continued*)

Operating System Security Principles	Grade	Comments
Protection from programming errors	C	Basic stack protection is the new normal. Control flow protection is the bar set by the industry today, which is lacking in Zephyr.
Access-Controlled Secrets storage	F	With the combination of no filesystem authorizations and no special secrets storage, Zephyr leaves a system vulnerable to any attacked thread. Systems with secrets should use a Secure Element or TPM to protect secrets, but this requires custom additions to Zephyr's device support.

While Zephyr provides some basic security features, like memory regions, and separate threads with stack protections, and user and privilege modes, Zephyr is limited in the services and protections that are available due to its focus as a minimalistic RTOS. But, even in other more powerful operating systems, similar process and thread structures are used, leading to similar attacks and pitfalls, so Zephyr is instructive to analyze. Our security lessons from Zephyr are applicable to all platforms and operating systems. Threads can be attacked and therefore should run at the lowest privilege possible. Privileges can be abused, maliciously or unintentionally, and therefore guards should be in place to check proper behavior of the system. Memory subsystems and filesystems can be exploited to leak or corrupt data; therefore, cryptographic protections such as encryption and integrity protection should be used. As we explore other software on our generalized IoT system, we will highlight how a defense in depth approach can work to minimize risk and reduce the impact of successful attacks.

Linux Operating Systems

Linux is a common operating system used for both cloud and IoT instances. It is feature rich and comes in many different distributions (distros) that enhance or embellish one capability or another. The security properties of Linux are well known, and there are complete tomes that do an excellent job of covering this topic,[12] so this section will not repeat that material here. Instead, this section looks at the concept of enhanced containment, but we do so from the perspective of an interesting IoT problem – updating the operating system and application software on a platform. The Linux distros covered here include Wind River Pulsar, Ubuntu IoT Core, and Clear Linux.

It is important to understand the update problem before progressing into the details of the distros. The update problem encountered in operating systems is one of both synchronization and access. Synchronization between different software elements of a system, and between the software and hardware of the platform, is required. An update to a system can destroy this synchronization. Access relates to the permissions and capability to update all parts of the system, including the operating system kernel, the boot software, and all types of firmware on the device.

A bad software update creates an incompatibility between two different software components on your device or an incompatibility between the software and the hardware of your device. An update problem is observed when two or more software components interfere with one another. The result of any of these conflicts can be a slowdown in operation, the failure of one or more services, a computer shutdown during operation (i.e., a crash), or even a failure to boot the device. It is not uncommon for some Linux updates to cause a failure to boot after a kernel update, which then requires a rebuild of the boot device in order to remedy the situation. A good software update requires synchronization between the hardware and all the software on the platform.

[12]Multiple Linux topic books by Apress, www.apress.com/us/open-source/linux

The word *all* introduces the other part of the update problem: *Access*. We defer the access issue until the section on secure updates, but it is important to understand the complexity of the update problem here and realize that the distributions we discuss now do not solve the whole problem. The access problem is caused by some updatable software on a device that resides in one or more difficult to reach hardware storage areas, normally referred to as firmware. The operating system itself may not be able to reach all these firmware locations. The device may need to be placed into a special operating mode, or an update must be submitted at a particular time during the boot process for the firmware update to be successful. This special access required to update firmware may be difficult or impossible to do without human intervention. If some part of the device's regular software is updated, and it depends on a newer version of firmware that is not present on the device, the instability of a bad software update may be the result.

If an operating system update causes an IoT platform to fail to reboot, or to crash so often that a new update cannot be pushed to the device, this requires a human being to go out to the device and repair or replace it. This physical maintenance drives up the cost for IoT deployments, resulting in an erosion or destruction[13] of the return on investment (ROI) for the IoT system. Driving operational costs down to preserve ROI requires the elimination of such physical interactions.

All three of the distributions covered in this section attempt to address the software update problem for IoT but do so in different ways. As we review these different solutions, we find the commonality is all about containment and finding ways to isolate the inconsistent dependencies.

[13]Destruction of the ROI can occur when many devices are impacted by a bad system update, either simultaneously or repeatedly over time. The cost of "rolling a truck" to repair devices can drive operational costs to completely consume any profit or efficiency gained by the IoT system.

Pulsar: Wind River Linux

Wind River provides various different operating systems for embedded sectors, including IoT. VxWorks[14] is a family of products representing their RTOS offerings. Pulsar[15] is Wind River's small, high-performance Linux distribution designed for manageability and IoT.

Pulsar is a binary distribution of Linux based on the Yocto Project. A primary focus of Pulsar is to provide a regular cadence of updates for the packages that are included in Pulsar, including the kernel. As shown in Figure 4-5, Pulsar is a container-based Linux, allowing the download of different features and functionality as containers. However, within the containers, updates are managed in a traditional manner using software packages.

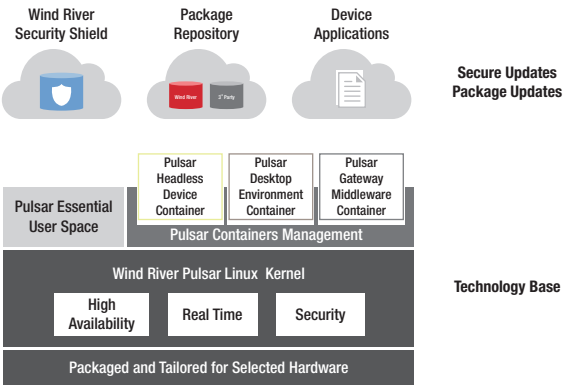

Figure 4-5. *Pulsar Linux architecture and service updates*[16]

[14]Wind River VxWorks, www.windriver.com/products/vxworks/

[15]Wind River Pulsar, www.windriver.com/products/operating-systems/pulsar/

[16]From www.windriver.com/products/product-overviews/Pulsar-Linux-Product-Overview/

Using containers as a separation capability reduces destructive interactions between applications and makes the whole platform more stable. Additionally, by using containers, there is greater security in the platform as a whole, since the containers have a reduced privilege on the platform, making an attack on an application in a container less likely to leak out and affect the whole device. Pulsar can update whole applications on the device seamlessly by just replacing a container.

Wind River addresses the issue of stable updates by providing an update service over a secure channel, where the updates themselves are comprised of RPMs (Red Hat Package Manager), a common Linux update mechanism. All RPMs are signed with a Wind River RSA[17] private key, ensuring the RPMs are genuine and not modified from what Wind River intended. All updates on Wind River's package repository have gone through extensive testing to ensure they are stable on the Pulsar-supported platforms. Constant reviews of the published Common Vulnerabilities and Exposures (CVE) databases, and the open source mailing lists, ensure the latest defects and issues are addressed in the quarterly updates.

Wind River Linux includes the following features, discussed elsewhere in the chapter:

- Wind River Helix Device management system
- Mosquitto MQTT
- OCF and IoTivity (See Chapter 2 Consumer IoT Framework Standards)
- UEFI or MOK Secure Boot (See Chapter 3, Device Boot Integrity - Trust But Verify)
- Support for Trusted Platform Module (TPM) (See Chapter 3, PTT/TPM)

[17]RSA (Rivest-Shamir-Adleman) is an asymmetric cryptographic algorithm that uses a private key to digitally sign data and a separate public key that anyone can use to verify the signature.

Pulsar includes the following other technologies that improve the security on the device:

- Virtual private network (VPN) provided by the open source StrongSwan IPSec/L2TP/PPTP project.
- STIG scripts: System lockdown scripts are included in Pulsar to configure the system for secure deployment, using the US government's Security Technical Implementation Guide (STIG)[18] scripts.

CONTAINERS

Containers are a type of software separation technology that allows one or more applications, and their dependent libraries, packages, and services, to run in an operating system created *namespace*.

In an operating system, certain resources are organized into namespaces. For example, all the users are in a namespace; this means you can have only one user named *root* and one user named *dave* (users are actually based on numeric identifiers, but the concept still holds). If there are two users both named *dave*, they would be the same user. Likewise, the same namespace concept exists with devices, file paths, and certain logical resources, like network ports and process identifiers.

Inside a container, the operating system gives the container its own namespace for certain types of resources. So one container can open port 443 for a web server to listen to incoming traffic, and a different container can also open port 443, and there would be no conflict. Outside the container, some type of mapping must be done to disambiguate the two network traffic flows (see the "Containers" section for details). In our example with the user identities, two containers can both have the user *dave*, and they would not

[18]STIG Home, https://iase.disa.mil/stigs/Pages/index.aspx

be associated to the same user; thus there would be no conflict between the containers and no privilege leakages or access overlap between the applications in the containers.

Containers also use another kernel feature called cgroups. Cgroups create a kernel structure that limits the amount of memory and CPU processing that is available to processes within a cgroup. This can be used to ensure the processes in a cgroup do not starve out other groups. This ensures that all containers get a fair amount of processing time, and one container cannot hog the CPU and prevent applications in other containers from executing.

Different containerization engines package these features in different ways to allow an environment to be created and managed that provides usable software separation for applications. These are all referred to generally as containers, but different containerization engines may have slightly different properties and controls.

Ubuntu IoT Core

Ubuntu is a popular Debian Linux distribution that includes desktop, server, and cloud versions. Ubuntu IoT Core is a new distribution that is headless, meaning that it does not include the elements an operating system normally provides for a screen, keyboard, and mouse – there is no user interface. Ubuntu IoT Core is intended to be used on devices that do not have buttons; they are intended to be turned on, and the device just does its thing, whatever that is.

Ubuntu IoT Core runs differently from the normal Ubuntu distributions. It uses a construct called a *snap*. Everything in Ubuntu Core is a snap, even the kernel. Developers create snaps that contain all the dependencies for their application or service. Users download snaps from the snap store and can add in (snap in) any snap they want to their system. Each snap is separated from the others in Ubuntu IoT Core, using

similar separation constructs as containers! One difference however is that snaps are transactional and can be rolled back easily if there is a problem. Thus, trying out a snap leaves no artifacts on the system, and a snap can be completely removed at any time. (reference for diagram in Figure 4-6. https://computingforgeeks.com/install-snapd-and-snap-applications-on-fedora/).

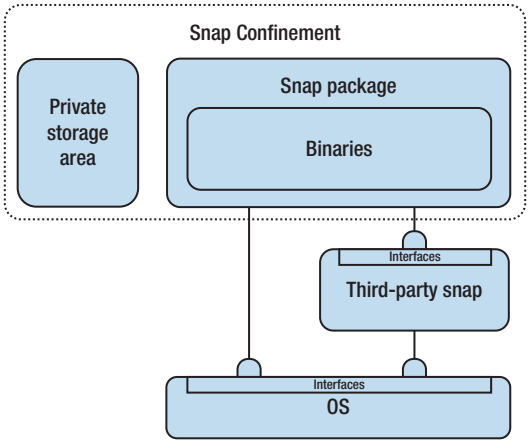

Figure 4-6. *Ubuntu IoT Core snap architecture*

A snap is actually a filesystem (the SquashFS filesystem) along with a YAML file that contains the snap's metadata. A snap is completely relocatable and does not depend on having specific libraries or configurations in a particular directory, like the `/etc` directory. The snap must carry all its dependent libraries with it in the SquashFS, kind of like a TAR or ZIP file with everything it needs packaged up inside it. The code for the snap in SquashFS filesystem is read-only, but once the snap is installed, a writeable section of the filesystem is created. When a snap is installed, it can be granted permissions to access things outside its filesystem, like the network or devices. If the system does not grant those permissions, then the install fails. In this way, a snap is similar to an app in the Android operating system.

Ubuntu IoT Core is claimed to be more reliable and more secure. Snaps are signed with cryptographic keys, just like Pulsar's RPMs, but snaps manage their own dependencies and are separated from other applications. Ubuntu IoT Core creates isolation between applications (snaps) using AppArmor and Seccomp.

AppArmor[19] is a security model built into the Linux kernel as part of the Linux Security Modules (LSM) framework. Other models supported by LSM include SELinux, Smack, TOMOYO Linux, and Yama. AppArmor allows the definition of security profiles that restrict the behavior of applications, and access to files (inodes), based upon a set of mandatory access control (MAC) policies. AppArmor comes installed with various preconfigured profiles to protect the system and applications, but these are modifiable by an administrator. Applications that do not have a policy defined execute in an unconfined manner (no special MAC restrictions). Policies reside in `/etc/apparmor/` and user-specific profiles are defined in `${HOME}/.apparmor/`.

Seccomp[20] is a Linux kernel mode used to limit the kernel system calls available to a process. Seccomp is short for secure computing and reduces the attack surface that the Linux kernel exposes through system calls. Seccomp was originally designed to expose only a certain set of kernel APIs available, but Seccomp 2 added filtering, allowing more flexible definitions of what kernel APIs are allowed to be used by a process. Seccomp is effective in restricting the actions an attacker can perform through injected code attacks, because a call to a restricted system call sends the SIGKILL to the process, terminating the offending program.

The combination of AppArmor and Seccomp allows Ubuntu to restrict the allowable actions of installed snaps. The inherent restrictions of a snap simplify the policy for these security tools, which

[19] https://gitlab.com/apparmor/apparmor/wikis/home/

[20] www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt

can be complex. Additionally, the filters that restrict the snap's actions actually document how the snap is supposed to behave, and what the app can and cannot do, which acts as a type of disclosure to the system administrator. In conclusion, the containerization of snaps includes a separate filesystem, special permissions with AppArmor and Seccomp, and documented interfaces to connect to other applications and services on the platform through the `snapd` service.[21] Using these strong security protections, and the ability to rollback misbehaving snaps, Ubuntu IoT Core provides a secure and stable operating system for IoT deployments.

Intel® Clear Linux

Clear Linux[22] addresses the operating system update problem by allowing frequent updates to the operating system, reducing the time a platform lacks the most recent updates, and preventing incompatible updates from being downloaded and installed on a system. Clear Linux is designed for a Linux distribution maintainer and provides tools allowing the maintainer to directly consume upstream projects, add them to their distribution, and maintain the distribution on an update server that keeps all the connected systems updated. It is easy to see the value of Clear Linux to an IoT deployment that is using a customized Linux kernel.

[21] https://tutorials.ubuntu.com/tutorial/advanced-snap-usage#1

[22] Clear Linux, https://clearlinux.org/

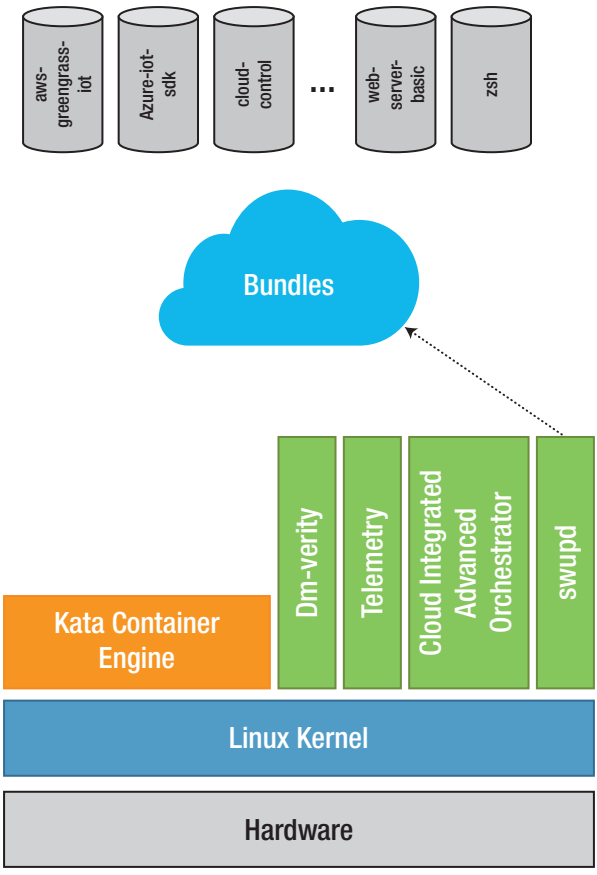

Figure 4-7. *Clear Linux deployment chain*

Clear Linux manages all the applications and software on the system using *bundles* instead of packages. Packages are hard to manage because of all the dependencies, and oftentimes different packages have dependencies on different versions of other packages. When two different packages are installed, and each requires different versions of another dependent package, installing both of those packages creates contention. Either one package will be able to use the newer (or older) version of the dependent package, or the application will break. Pulsar addresses this

contention by putting applications and services into containers, which separates the dependencies from each other; Ubuntu uses a similar approach with snaps. Clear Linux removes the contention with bundles, which is just a different containment mechanism. A bundle removes the outside dependencies and includes all the software needed for an application.

In Clear Linux, the operating system is completely made up of bundles. When one bundle is updated, it creates a completely new version of the OS. This new OS version is built and tested as a whole – there is no extra package to be added later. For the distributor, this makes updating simpler and guarantees that the OS update will work and will not brick the system. It is also the reason that updates need to be easier and happen more frequently.

Just making updates come faster is not really a solution. Updating an entire operating system every week could kill a system, not to mention bog down the network. Clear Linux solves this problem by including tools to allow updates to be smaller. Rather than an update requiring a full reinstall, the update can be a binary diff between versions. This is critical for IoT deployments, because sending down a new kernel that is multiple megabytes in size is just not practical over certain network connections.

Linux Summary

Linux supports strong security capabilities in both the kernel and the application space. Although we did not cover all of Linux's security features, Table 4-3 provides a summary of the operating system security features of Linux for comparison with Zephyr in our previous section.

Table 4-3. *Linux Security Summary*

Operating System Security Principles	Grade	Comments
Execution Separation	A	All Linux distributions discussed here support the standard separation capabilities (process, threads, ISRs) of operating systems.
Memory Separation	A	Linux utilizes the hardware memory management unit (MMU) to provide paged memory separation for all processes, with read-write-execute permissions. Unlike Zephyr, even a process running as root is restricted.
Levels of Privilege	A	Linux, like Microsoft Windows, has access to multiple privilege rings, but makes use of only two ring levels. Linux also supports other special supervisory modes and TEEs; for details see the section on containment.
System authorization	A	Linux provides authorization for structures using a common user-group-other identity structure with read-write-execute privilege bits. Extensions for other security models through the Linux Security Modules (LSM) and other frameworks, like AppArmor and Seccomp covered in the Ubuntu section, are readily available and integrated into the Linux kernel.

(*continued*)

Table 4-3. (*continued*)

Operating System Security Principles	Grade	Comments
Programming Error Protections	B	Basic stack protection is provided in the Linux kernel since version 3.14[23] and is turned on automatically in version 4.16[24] – strong stack protections are also an option. Control flow protection is not yet fully upstreamed in the kernel, but patches exist for 64-bit user applications.[25]
Access-Controlled Secrets Protection	C	Linux does not directly provide standard features for secrets storage, but support for the Trusted Platform Module (TPM), Secure Elements, and hardware security modules (HSM) are prevalent.

In this section, our discussion focused on the update features provided in different Linux distributions and how the distros are solving the problem of interfering applications and overly complex dependencies. These solutions used different forms of containment to solve the update problem. Clear Linux solves the problem by creating a new package format for updates called a bundle and then uses a series of tools to ensure the different bundles create a stable system. If an instability is found, a new update is easy to create by correcting a bundle. System updates are made less burdensome by incorporating special binary diff updates that take less time to download.

[23] https://lwn.net/Articles/584225/

[24] www.thomas-krenn.com/en/wiki/Linux_Kernel_Versions

[25] https://lwn.net/Articles/758245/

Pulsar and Ubuntu take a different approach and use advanced features of Linux to construct special containment for applications and even parts of the operating system itself (in the case of Ubuntu, anyway). These containment features are used to create Linux containers, which we look at in a bit more detail in a future section.

We also noted that even with these features, the problem of access required to update firmware on the platform is not solved by this approach, and additional capabilities are needed. We look at solutions to the access problem in the section on secure software updates.

Hypervisors and Virtualization

Virtualization is a generic term applied to several techniques that increase resource sharing and hardware utilization in a computer system. Modern operating systems like Linux provide *virtualized* memory, where more memory appears to be available than is actually physically present. Parts of memory used by idle processes are stored on disk, freeing more physical memory for the currently running process; short delays are incurred when the idle process becomes active and the operating system reloads physical memory with the contents from disk. Although some delays are incurred, they are outweighed by the benefit of having more physical memory available to the running process.

Platform virtualization works in much the same way, allowing multiple operating systems to run simultaneously on a single computer. Memory is virtualized, as well as the processor, storage, graphics, and other I/O devices on the platform. A small control program, called a hypervisor or Virtual Machine Manager (VMM), manages the virtualized hardware and mediates between the different virtual machines (VMs). Figure 4-8 shows a generic virtualized system. Each VM runs a guest operating system and application software that are logically separated from each other by hardware and software controls managed by the hypervisor.

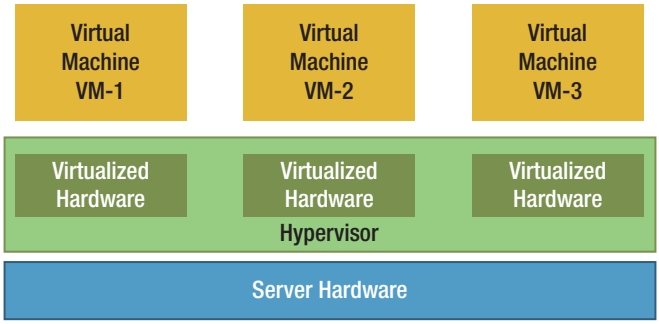

Figure 4-8. *Generic virtualization architecture*

There are actually two different types of hypervisors. Figure 4-8 depicts a Type 1 hypervisor or native hypervisor that runs directly on the hardware. Type 1 hypervisors are typically more performant and can utilize the hardware better, because they have complete control of the hardware. VMWare, Xen (for Linux), and Hyper-V (Microsoft) are examples of Type 1 hypervisors.

There are also Type 2 hypervisors, which run on top of an existing operating system. This allows a *regular* OS to run virtual machines too. VirtualBox by Oracle is a Type 2 hypervisor that runs on Linux. KVM is a Red Hat hypervisor that runs as part of the Linux kernel; some regard it is Type 2 hypervisor since other things can run on the Linux OS, but Red Hat claims it is a Type 1 hypervisor since it has direct control of the hardware through the kernel. Either way, it is a pretty good hypervisor. There is a question that frequently comes up relating hypervisors to containers. The question is: Which is better, containerization or virtualization? We discuss this later in the "*Software Separation and Containment*" section. For now, we focus on virtualization.

How does virtualization work? In Intel Architecture, virtualization is supported by the Virtual Machine Extensions (VMX) mode. This mode defines two privilege levels, one for the hypervisor, called *VMX root operations*, and one for the VMs, called *VMX non-root operations*.

As one might guess, the VMX root operations mode is more privileged. The hypervisor, operating in VMX root operations mode, initializes certain control registers in the processor to establish limits on the VMs. The hypervisor releases the VMs to execute by performing a *VM*-Enter instruction. The VMs are then executing in the restricted VMX non-root operations mode. When the VMs execute an instruction or perform an operation that is restricted by the hypervisor, a *VM exit* is performed by the processor, returning control to the hypervisor. The hypervisor can either perform the operation on behalf of the VM in a safe manner, or it can reject the operation and return some type of exception to the VM; in extreme cases, the hypervisor can even terminate the offending VM entirely.

The exact details of virtualized processor state are beyond the scope of this book. However, the curious may elect to read the *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3*. Chapters 23 through 33 cover VMX mode. These chapters discuss the virtual machine control structure (VMCS) that contains the state used by the processor to implement virtualization and discusses all the elements of the controlled state, including

- Virtual processor state, including control registers, debug registers, base registers, and segments
- Bit flags controlling what events cause a VM exit, for example, interrupts, use of IO ports, and so on
- Bit flags indicating how a VM's state is saved when a VM exit is performed
- Bit flags indicating how a VM's state is restored on VM entry
- Indicators for VMX aborts (the reason a VM abnormally exited into the hypervisor)
- Indicators for VMX exits (the reason for a normal return to the hypervisor)

There is also another distinction among hypervisors. In the discussion earlier, we described virtualization as though the operating systems in the virtual machines were no different than an operating system on a platform that is not virtualized. When the operating systems in the VMs are not aware they are being virtualized, this is called *full virtualization*. In some systems, or with some applications, it is very difficult to fully virtualize the system. This may be because there are complex devices that need to be shared between the virtual machines, or it may be that an application has very stringent performance requirements. In these cases, it is counterproductive to perform full virtualization – the cost to do so would outweigh the benefits. In these cases, the hypervisor implements a *para-virtualized* strategy, where the operating system, device drivers, and perhaps even the applications themselves are aware that they are being virtualized and are modified in order to behave better in the virtualized environment. Para-virtualization is accomplished by configuring VMX in a way that allows the VMs themselves to perform certain operations, for example, the ability to directly interface with certain IO ports. The VMCS allows the hypervisor to give some VMs more control than other VMs. However, the VMs must cooperate with the hypervisor and are trusted to cooperate in a trustworthy fashion. Intel's ACRN hypervisor is an example of a para-virtualized hypervisor, which we will discuss in more detail after we review the security threats to virtualization and hypervisors.

Threats to Hypervisors

Just like operating systems, the threats to hypervisors are numerous and dangerous. A successful attack on a hypervisor can lead to an attacker acquiring complete control over the platform and every virtual machine running on it. NIST-SP-800-125A Revision 1[26] outlines the baseline set of

[26] https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-125Ar1.pdf

security functions a hypervisor should perform. These hypervisor security functions have similarities to the security services that we defined for operating systems. But because this specification limits itself to virtualization of servers, and specifically does not address embedded systems, we use the NIST set of functions as a baseline. In our list that follows, the first five items are from NIST and are roughly equivalent to the security services we defined for operating system; protection from programming errors is a sixth security service we add to NIST's list. We then add three additional security services that are unique to IoT instances and not considered by NIST's analysis. The set of security services for IoT hypervisors are

- **VM Process Isolation (i.e., Execution Separation and Memory Separation)**: Each VM's execution should be separated from all other VMs' execution using multiple logical processor structures; a fault in one VM should not affect other VMs.
- **Device Mediation and Access Control (i.e., Levels of Privilege and Access-Controlled Secrets Storage)**: Hypervisors provide methods for VMs to share access to devices through various methods, including giving VMs direct access to hardware, para-virtualization of the device, or device emulation within the hypervisor. Access to the devices must be controlled to prevent effects from one VM leaking over to other VMs. This includes controlling direct memory access (DMA) devices to protect both memory read and write. If the platform offers secrets storage, the hypervisor should provide access to such storage in a manner that prevents other VMs from interfering with each other's usage, or from viewing, modifying, or using the secrets in that secure storage location (see Chapter 3, section on "*Intel Virtualization Technology (Intel VT)*").

- **Prevent Abuses by Guest VMs through their Direct Execution of Commands (i.e., System Authorization)**: As we stated earlier, para-virtualized systems allow VMs to cooperate with the hypervisor; the hypervisors in these systems execute commands sent from the VMs. The hypervisor's execution of commands from one VM should not affect another VM and should not compromise the security of the hypervisor or its data structures.
- **VM Lifecycle Management**: VM management includes creating, starting, stopping, and pausing VMs, as well as checkpointing (snapshotting) their state. This includes monitoring the state of VMs and various tools for migrating data or VM snapshots between physical machines. The management of VMs is typically performed through add-ons to the hypervisor or through a special management VM. These management services must not allow leakage of data or control across VMs.
- **Management of Hypervisor Platform**: The configuration of the hypervisor and the platform itself must be managed, including configuring devices, virtual networking, storage, and any VM policies. This management must include proper authentication of management requests and restriction of management actions to only authorized entities.
- **Protection from Programming Errors**: This is the leftover security service from our operating system list, but has a little different perspective when viewed from the virtualization perspective. The hypervisor must set appropriate VM aborts when a VM violates

restrictions on memory separation or corruption of VM control structures; stack smashing and heap smashing that occur from programming errors in one VM should not compromise the hypervisor or other VMs. These protections become much more difficult with a para-virtualized hypervisor because certain structures and interfaces on the hypervisor are accessible to the VMs.

- **Real-Time Guarantees**: In embedded systems and IoT, control loops require real-time guarantees and many devices operate with real-time restrictions on read/write operations that if violated result in data being lost. In a virtualized system, the hypervisor itself must provide these real-time guarantees in coordination with the VMs and their operating systems.
- **Deep Power Management**: In embedded systems and IoT, power usage is a critical parameter. Whether the power envelope is restricted due to battery life and energy harvesting limitations or the power/heat trade-off in an industrial environment limits equipment's power budget, management of energy usage is essential. Due to real-time guarantees and the management of physical devices or equipment, the management of power goes far beyond what is normally provided in a data center or server cloud instance. Power management cannot be left to the individual guest operating systems or VMs, because they do not have the platform view. The hypervisor, in conjunction with the VMs and guest OSes, must manage the platform constraints appropriately to prevent power spikes or violations of the equipment's defined heat envelope.

- **Protection from External Devices**: In embedded systems and IoT, virtualized systems are inevitably connected to other devices and sensors, usually in a very direct way. Because these devices can be attacked and PWNED[27] by an adversary, protection of the virtualized system from compromised devices is critical to protecting a virtualized IoT system. It should be noted that this goes beyond the normal protections a cloud server is required to enforce to protect REST APIs and network connections, which the IoT virtualized system must also do. The external IoT devices are normally connected to a low-level driver, an emulated or virtual bus implementation, or some other higher-privileged software component that must implement some type of intrusion and attack detection-prevention mechanism.

When examining the preceding list of necessary protections and the general operation of hypervisors described earlier, several threat vectors immediately come to the surface that are likely vulnerabilities in hypervisors:

- **Size and complexity of the hypervisor code**: The more complex and larger code size of a hypervisor, the more likely the hypervisor includes critical vulnerabilities, because adverse code interactions and defects are harder to find in larger code bases.

[27]PWNED is the Internet slang for "owning" a device or computer system; it comes from "mistyping" the "o" in the word "own" with a letter "p," ostensibly because hackers are bad typists perhaps. Its meaning goes beyond attacking and implies complete ownership of the attacked device such that the device is absconded to do whatever the attacker wishes – the device becomes part of the attacker's zombie or botnet army.

- **Attack surface of the guest VMs**: Since the VMs represent the manipulable interface to attackers, they represent the primary point of attack to virtualized systems. The more network services exposed by a VM, the more third-party code that is not written with a security mindset, and the larger the number of unprotected IoT devices connected to virtualized system, the higher the risk of vulnerabilities that can expose the hypervisor to attack.
- **Hypervisor add-ons that have vulnerabilities**: Some hypervisors have minimal services but allow add-ons or plugin modules that provide additional services, like management and configuration. These add-ons can include additional vulnerabilities.
- **Device driver virtualizations that have vulnerabilities**: Device drivers require special versions that provide virtualization features, which may react differently with different hypervisors or may operate differently on different hardware. These differences may create vulnerabilities an attacker can leverage.

Like operating systems, hypervisors are susceptible to similar classes of attacks. A recent survey paper[28] looked at reported common vulnerabilities from a reputable CVE database for the top four hypervisors. Figure 4-9 shows the types of vulnerabilities and the number of such vulnerabilities by product. The purpose of this table here is to highlight the most common attacks on hypervisors and to highlight that all hypervisors have been successfully attacked. The data should not be interpreted numerically

[28]Litchfield, Alan., Shahzad, Abid. *A systematic Review of Vulnerabilities in Hypervisors and Their Detection.* 23rd Americas Conference on Information Systems. Boston. 2017.

to identify which hypervisor is more secure due to a lower number of attacks. The paper notes that although VMWare had the highest number of vulnerabilities over the study period (from 1999 to 2015), it was also the only established hypervisor product in the market for the first 8 years of the study period, making such rankings of hypervisor security inappropriate. The following list briefly reviews the most prevalent classes of attacks listed in Figure 4-9, describing the security principles violated:

- **Denial of Service**: A DoS attack causes a VM to halt or create such a serious VM abort that the hypervisor refuses to allow the VM to continue to operate. A more serious DoS could affect a device on the platform, preventing all VMs from accessing the device until the platform is rebooted, violating **Device Mediation**. A DoS attack on a virtualized hardware device represents a violation of **execution separation**. Another type of DoS attack consumes resources, like network socket handles, resulting in other VMs not being able to acquire the resource necessary to execute a function.

	VMWare	Xen	KVM	Hyper-V	Total
DoS	66	131	30	5	232
Code Execution	48	12	2	3	65
Stack overflow	30	28	4	1	63
Memory Corruption	8	10	2	1	21
Cross-Site Scripting	13				13
Directory Traversal	11				11
HTTP response splitting	1				1
Bypass something	5		3	1	9
Gain information	17	16	2		35
Gain privileges	54	24	7		85
Cross Site Request Forgery	3				3
Total	256	221	50	11	

Figure 4-9. *Most common attacks on hypervisors – 16-year period*

- **Stack Overflow and Arbitrary Code Execution**: Stack smashing, heap smashing, and use-after-free vulnerabilities allow an attacker to execute their own code on the platform. This type of attack can allow escalation of code's rights, allowing it to become a privileged user. In para-virtualized environments, this can cause the VM to misbehave and violate the trust the hypervisor places in the VM, causing an execution or memory separation violation (**Prevent Abuses from Direct Execution of Commands from Guest VMs**).
- **Gain Information**: An out-of-bounds read vulnerability allows a VM to access memory outside of its logical memory space. These vulnerabilities are common with virtualized drivers and VM tools. A gain information vulnerability represents a violation of **memory separation**.
- **Gain Privileges**: Gain privilege attacks are usually executed through add-ons, like tools and plugins. An example is the CVE-2017-4943 that allowed a *showlog* plugin to gain root-level privilege of the platform management VM that controls network settings, system updates, health monitoring, and device management. Becoming root on a para-virtualized system is tantamount to a compromise of the hypervisor itself, since root on a para-virtualized VM allows the attacker to easily violate the implicit para-virtualized cooperation agreement (**Management of Hypervisor Platform**).

Many of the attacks outlined are serious, but do not directly violate a fully virtualized system; the hypervisor can properly trap and stop attacks that directly violate the virtual machine's configuration. However, when

the hypervisor and VMs are operating in a para-virtualized manner, privilege escalations in the VM and process and memory violations even within the VM's logical memory space can escalate to a violation of the para-virtualization agreements. An attacker, operating as root within a para-virtualized VM, can disrupt device drivers and other critical parts of the VM's operating system that have direct access to the platform hardware as part of the para-virtualization contract. In the next section, we look at ACRN, a para-virtualized hypervisor, and explore some of the strengths and weaknesses of this approach.

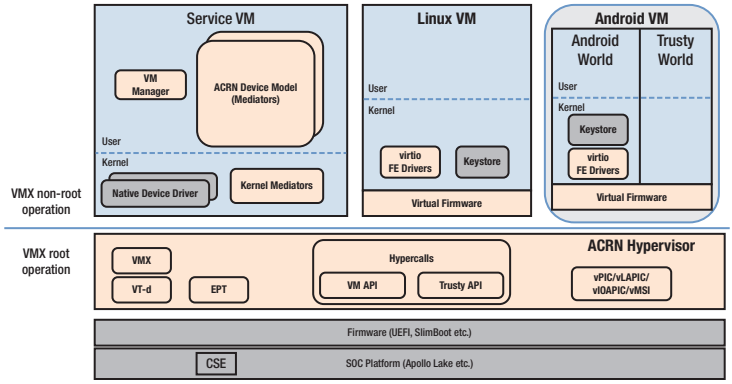

Figure 4-10. *ACRN architecture diagram*

Intel® ACRN

ACRN is a BSD open source hypervisor reference platform, built by Intel for the automotive industry, available at `https://projectacrn.github.io`. It is specifically designed to be a flexible and lightweight hypervisor and designed for real-time and safety-critical IoT deployments.

As shown in Figure 4-10, ACRN is a para-virtualized architecture where the guest operating systems must know they are being virtualized and cooperate with the hypervisor. A para-virtualized solution is required

in the automotive world due to the nature of some devices in the system. This model enables a more performant implementation and cleaner virtualization of these devices using virtio drivers. Notice the Service Virtual Machine (VM) in the top left of Figure 4-10. The Service VM performs some critical virtualization services for the hypervisor to avoid the performance penalty of full virtualization. However, support for device interrupts is provided directly in the hypervisor by virtualizing the PIC and APIC for each VM. The critical element of the Service VM is the set of ACRN Device Model (DM) applications that mediate between VMs and devices for certain operations. For example, USB and IOC (I/O Controller) devices are emulated in the Service VM due to their complexity, and the GPU is mediated by the Service VM since emulation will not provide the performance boost for which the GPU is often used. Because of these elevated privileges, the Service VM is a critical security element in the trusted computing base (TCB) of the ACRN platform. If not carefully limited, the Service VM can easily take on too much and become a security threat due to violation of the least privilege principle. As the number and complexity of the Device Models grow, the likelihood of implementation errors that can be leveraged by an attacker grows (see the list of common attack patterns discussed in the "Threats to Operating Systems" section). If an attacker is able to successfully attack a DM, the attacker is likely to inject other code inside the Service VM, having access to many other privileges than just the compromised DM. This is an architectural trade-off between necessary performance and security risk. The risk can be managed by ensuring every DM or other software component added to the Service VM is carefully verified and undergoes penetration testing to ensure there are no security weaknesses in those modules.

For security features, ACRN supports secure boot, a Trusted Execution Environment (TEE), and secure storage in a Replay Protected Memory Block (RPMB) in flash. Figure 4-11 shows the secure boot flow for ACRN when using the Slim Bootloader (SBL). The TEE and RPMB are shown in Figures 4-13 and 4-14, respectively.

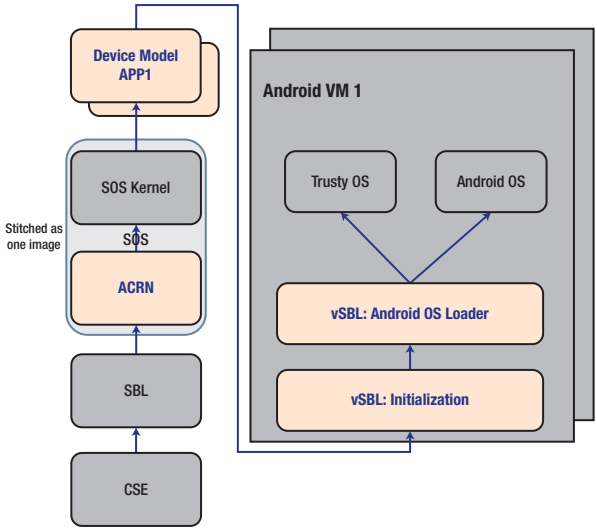

Figure 4-11. *ACRN secure boot flow*

Secure boot on Intel devices starts in the Converged Security Engine (CSE), which is the common root of trust for verification for Intel platforms (see Chapter 3, section "Intel CSE/CSME – DAL"). The CSE verifies a digital signature on the SBL; the digital signature is usually produced using the RSA algorithm and is commonly 2048-bits or 3072-bits in length. The public key is part of the SBL image, but this key is verified by the CSE using a hash of that public key kept in fuses. The fuses prevent the key from being modified in the image itself.

The SBL verifies the next stage of the platform, which includes the ACRN hypervisor and Service Operating System (SOS) kernel, which are included as a single image. The SOS kernel runs in the Service VM as VM's operating system. The SOS Kernel loads and verifies a Device Model application for each User VM that is loaded; this includes verifying

a virtual Slim Bootloader (vSBL) for each User VM. The SOS uses dm-verity[29] to check the validity of the DM App and the vSBL. The vSBL then is responsible to boot the User VM; in the case of Android, this uses the Android verified boot mechanism.

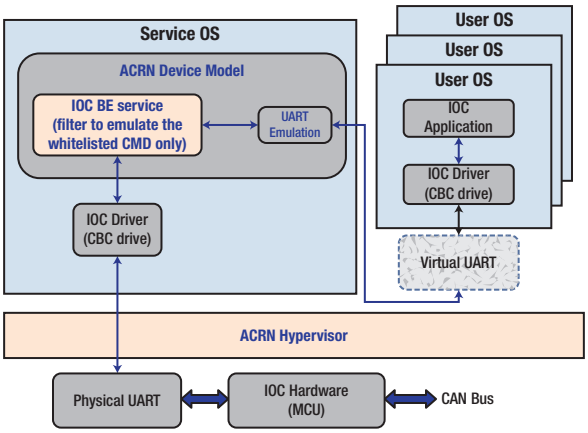

Figure 4-12. *ACRN connectivity to automotive CAN bus*

One of the key features in ACRN is support for real-time and automotive use cases. This creates extremely stringent requirements on the hypervisor and the VMs for real-time operations and connectivity. Because all VMs might require access to the CAN[30] bus, an I/O Controller is emulated in the Service OS that serializes data onto a physical serial

[29]DM-Verity, or Device Mapper Verity, was designed for Chrome OS and also used by Android. DM-Verity is built into the Linux kernel and uses the kernel cryptographic APIs to provide transparent integrity verification for block devices. See the Git Repository for more details at https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/device-mapper/verity.txt

[30]CAN bus, Controller Area Network, is a type of local bus system developed by Bosch for automotive systems to connect controllers and subsystems together. www.canbus.us/

bus connected to the vehicle CAN bus (Figure 4-12). In order to protect the vehicle, the Service OS implements a firewall in each VM's Device Model application. This filter restricts the type and content of messages that a particular VM can place on the vehicle's CAN bus. For example, the Android OS that implements the vehicle infotainment features is restricted from sending messages to critical ECU components for vehicle braking or engine control. Likewise, other VMs that render cockpit controls are restricted from receiving messages from USB ports in the cabin.

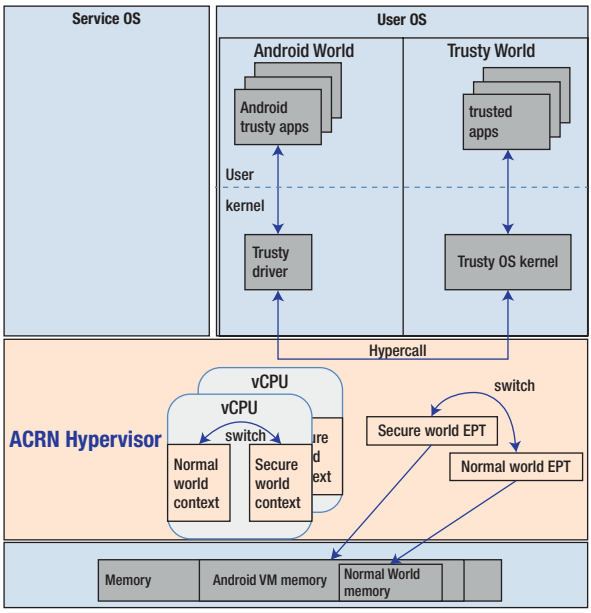

Figure 4-13. *ACRN trusted execution environment*

ACRN supports the ARM TrustZone TEE implemented in Trusty in the Android OS. As shown in Figure 4-13, the ACRN hypervisor implements the separation of unsecure memory (in the normal world or regular operating system) from the secure world purely in software through encrypted page tables (EPTs). The CPUs are also virtualized and maintain the NS (not-secure) bit used in ARM to switch between two different contexts in the vCPU. It should be noted that the secure world can see all

of memory, but the normal world is restricted to see only a subset. Just like we discussed in the Zephyr OS, the ability of privileged users to see all of memory makes processes and threads in the secure world potentially more dangerous. It should also be noted that the Service OS also acts like a privileged secure process with access to additional parts of memory in order to support the virtualized devices.

The last security feature in ACRN that we examine is the Replay Protected Memory Block (RPMB). RPMB is a feature of some flash devices that allows an encryption key to be used to protect data, using both confidentiality and integrity, in a reserved flash block. The data is also replay protected preventing rollback attacks where an old piece of encrypted data overwrites a newer piece of data.

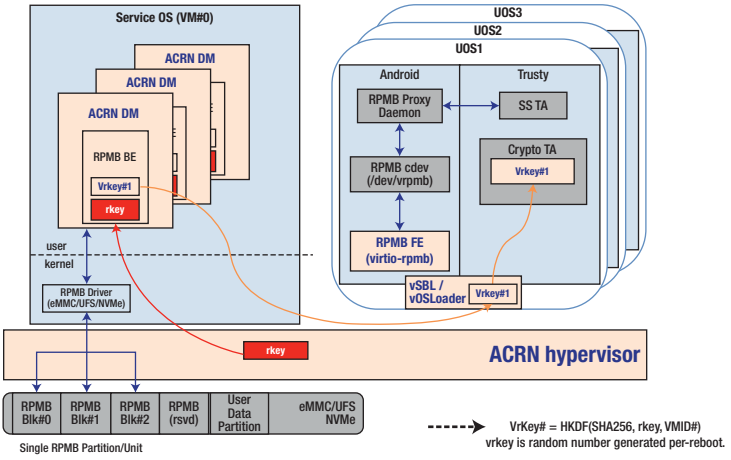

Figure 4-14. *ACRN secure storage support through RPMB*

The encryption key for the RPMB is held by a trusted entity in the platform. In Intel platforms, this trusted entity is the CSE, and the CSE shares this key with a single device driver on the platform and then locks access to the key so no other program can gain access to the key. If the key is overshared, then security of the platform diminishes. During the boot

process, the Slim Bootloader (SBL) reads the platform seed (pSEED) from the CSE and passes the pSEED on to the ACRN hypervisor. Since ACRN must support multiple virtual machines, and all these VMs must not be allowed to see the other VMs' data or be able to spoof another VM's data reads or writes, ACRN cannot directly share the pSEED with the VMs. ACRN uses a NIST-approved key derivation function (HKDF-256) to derive new secrets from the pSEED, called vSEEDs for virtual seeds, and passes a unique vSEED to each VM. Each VM then chooses which device driver or process will take ownership of the vSEED. For example, in Android, the vSEED is given to Trusty since it is the TEE for that VM. Figure 4-14 shows how the seeds are then used to implement RPMB. ACRN provides the real RPMB key to the DM applications in the Service OS. The derived keys are used by each of the User VMs to protect their RPMB data; the transactions for each of the User VMs do not go to the RPMB flash or the ACRN hypervisor, but instead are routed to the Service OS. The DM App in the SOS for the particular VM verifies and decrypts the data it received from its corresponding VM and then re-encrypts the data with actual RPMB key. Each VM has access to a small part of the RPMB and can only write to its own section. This separation is enforced by the RPMB driver in the SOS and the ACRN hypervisor.

It is clear from Figure 4-14 and the preceding description that the Service OS must be trusted, since it is possible for the DM Application to forge data or delete RPMB data as if they were the User VM. Careful review of the applications in the Service OS is required to ensure no security vulnerabilities are present, and only trusted applications are allowed to run in the SOS.

Real-Time and Power Management Guarantees in ACRN

In its current rendition, ACRN provides basic real-time and power management controls. ACRN maps a physical core into the guest OS for both real-time and power management. This means that the guest OS has direct control of the core and can reflect any of the operating system's

real-time characteristics to the applications in its VM. A real-time Linux kernel, for example, would run just as effectively in ACRN as on its own hardware. Since the physical cores are mapped into the VM, the hypervisor also allows the guest operating system in the VM control over that core's C-state, optimizing the core's power consumption during idle modes. The P-state, controlling the package voltage-frequency setting, is coordinated with the VM. ACRN manages the S-state, which is reflected from the User OS VMs, to the Service OS, and finally the hypervisor, in an ordered fashion. Future versions of ACRN are planning for further power and real-time management controls covering devices and real-time quality of service.

ACRN Summary

ACRN supports some strong security services, with RPMB secure storage and TrustZone TEE being two of the most significant. Many of the design and security trade-offs made in ACRN are a result of the performance requirements for automotive and IoT deployments and the need to interface with complex devices, such as the I/O Controller emulation in the Service OS for connection with the vehicle bus. Table 4-4 provides a summary of the hypervisor system security features for comparison with other systems.

Table 4-4. *ACRN Hypervisor Security Summary*

Operating System Security Principles	Grade	Comments
VM Process Isolation (Execution Separation)	B	Because ACRN is a para-virtualized hypervisor, and both the Service OS and parts of the hypervisor are accessible to guest VMs, the execution separation is not complete. This cannot be improved, however, due to the need for emulated busses and para-virtualization of certain devices.
VM Process Isolation (Memory Separation)	B	User OSs have access to both the Service OS and the ACRN hypervisor through some limited APIs. This necessarily means that some memory buffers and locations are shared, with some firewalling in place. Errors or defects in this sharing, especially if the uses of additional add-ons are integrated, can compromise the system.
Device Mediation (Levels of Privilege)	B	Device Mediation is done in the Service OS, per VM, using the Device Model application.

(*continued*)

Table 4-4. (*continued*)

Operating System Security Principles	Grade	Comments
Execution of Commands from Guest VMs	C	ACRN provides separation of commands, mostly through the Service OS and the Device Model. However, certain hypercalls go through the hypervisor itself as shown in Figures 4-10 and 4-13. Similar hypercalls are used for USB virtualization. This creates a disparity in where access controls need to be reviewed, and makes it harder to ensure all guest commands are properly mediated in every case; this represents a violation of the least common mechanism security design principle.[3]
VM Lifecycle **Management of Hypervisor Lifecycle**	C	ACRN provides a VM manager (Figure 4-10) in the Service OS; however the implementation is very slim. This is appropriate for the automotive space, but for generalized IoT, and especially for industrial usages which require sophisticated orchestration, the management features require significant add-ons. Because this is performed in the same VM as the mediation of the guest VMs, the likelihood of disastrous compromise is increased.

(*continued*)

Table 4-4. (*continued*)

Operating System Security Principles	Grade	Comments
Protection from programming errors	-	No specific controls; however the Linux OS and Android OS, for which the hypervisor was designed, provide these advanced controls. ACRN depends on these services in the guest OS.
Real-Time Guarantees and Deep Power Management	A	ACRN's entire design focuses on meeting real-time requirements for automotive, including providing optimized device drivers and virtualized access to power management controls using a virtualized PIC and APIC.
Protections from External Devices	A	ACRN provides a Service VM that mediates all external access points and utilizes VM-specific filters in the Device Model to individualize protection filters per VM instance.
Access-Controlled Secrets Storage	B	ACRN provides both a TEE and RPMB secure storage. The lower grading is a result of the implementations being primarily in software, not hardware.

In this section, our discussion focused on the unique features and architecture of para-virtualized hypervisors. We introduced the use of secure storage through the RPMB and additional containment through the use of a TEE. TEEs are discussed in more detail in the section "*Software Separation and Containment*." The design trade-offs for the hypervisor and

TEE led to potential vulnerability in the TEE due to lack of full memory separation – a similar problem was found in the Zephyr OS. These and other design trade-offs lead to some weaknesses in the system, but overall, the combination of hardware security features for VM separation and secure storage provides superior protection for the targeted IoT vertical.

Software Separation and Containment

Containment is a critical concept in security. Whether it is keeping the "bad guys" out, or protecting secrets, or just segregating high privilege operations from low privilege ones, separation and containment are paramount to safe operations. Even with the process and thread separation provided by the operating system, and the hardware-assisted virtual machine isolation, additional separation capabilities always seem to be useful to applications and IoT systems. In this section, we look at two different types of **extended application containment** capabilities: containers and Trusted Execution Environments (TEEs). We have touched on both of these topics already, but in this section, we unpack them to a deeper level.

Containment Security Principles

The principles that apply to **extended application containment** are the same principles we talked about for operating systems, which includes

- Execution Separation
- Memory Separation

The difference between applying these principles here and applying them to operating systems is the particular mechanisms used to provide the separation. The preference is for hardware separation as it is more secure. Containment through hardware separation might be provided

using a completely different processor (see the "Trusty TEE Security Summary" section), or a different mode of the current processor (see the section on Virtualization or SGX later). Memory separation might include a completely different cache of memory (as in SGX and Trusty), or merely using some extra virtualization controls (the approach used in hypervisors or containers). In both cases, there are trade-offs to be made, based on the threats that are being addressed.

Threats to Extended Application Containment

The threats to extended application containment typically come from privileged attackers. These attacks can come from a privileged user or might be from an unprivileged user that performs a privilege escalation attack to acquire higher privileges. In both cases, the application containment intends to remove the possibility, or reduce the efficacy, of attacks by privileged users (e.g., root or admin user accounts).

- **Memory Disclosure from Privileged User**: A privileged user leverages their access to all memory pages in order to read data from any application.
- **Memory Tampering from Privileged User**: A privileged user leverages their access to all memory pages in order to write, overwrite, or corrupt data for any application; they may also include making memory pages unavailable to an application.
- **Data Leakage through Side Channels**: A privileged user leverages their access to data caches to perform timing attacks allowing them to determine contents of application memory.

- **Execution Interference from Privileged User**: A privileged user leverages their ability to schedule tasks or run tasks that have higher priority and starve or interrupt other applications during a critical operation.
- **Execution Leakage through Side Channels**: A privileged user leverages their ability to schedule tasks and uses speculative execution or timing operations to determine code branches executed during operation.[31]

Application containment techniques provide defenses against these attacks to varying degrees. Full separation[32] is the only complete solution, but this increases costs and adds complexity to management and control of sensitive applications. The use of different containment techniques is a trade-off between absolute security and ease of use and utility of the solution. In each containment example discussed later, we highlight the different levels of hardware usage that improve the solution's security level.

Containers

Containers are a software mechanism to increase the separation between applications. In the "Linux Section", we discuss how Wind River Pulsar uses container to improve the stability of their operating system updates; because services and components execute within containers with enhanced separation between applications, the applications are less

[31]For a discussion of the L1 Terminal Fault (L1TF) speculative execution attack and its specific effect on ACRN, see https://projectacrn.github.io/latest/developer-guides/l1tf.html

[32]Full separation means using a completely different processor with completely different memory and devices for sensitive operations.

likely to interfere with each other, increasing system stability. Ubuntu IoT Core uses a similar construct to containers, which they call snaps. Containers and snaps utilize software techniques in the operating system for separation. The long-standing debate is which approach is better – containerization or virtualization?

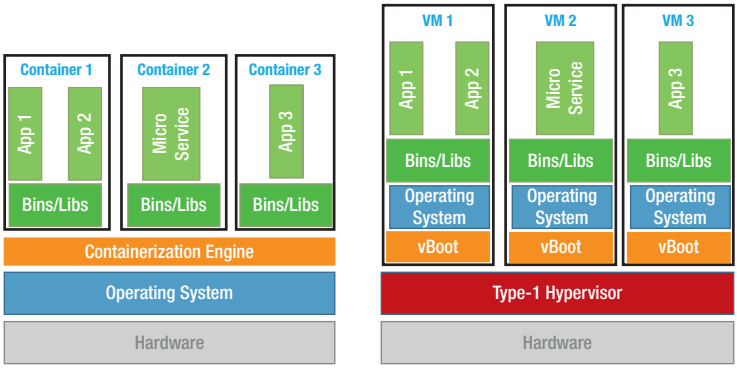

Figure 4-15. *Containers and hypervisor comparison*

In Figure 4-15, we show the relationship between a container software stack and a virtual machine (VM) software stack. What is evident from the diagram is the VM stack contains more layers of software due to the operating system in each VM.

The strength of a VM solution is the hardware separation between the different VMs; however, setting up the VMs and getting the operating systems booted in each VM takes more time. The strength of the container solution is faster startup time for each container and lower overhead of execution; the weakness in containerization is the reliance on software separation in the container engine and the underlying operating system. A best-in-class solution would be a hybrid that provides the hardware security of virtual machines with the speed of deployment and startup for containers. Kata Containers provides such a solution.

Kata Containers

Kata Containers is an open source project managed by the OpenStack Foundation, which includes technology from an Intel open source project called Clear Containers. Clear Containers is related to the Intel Clear Linux project discussed earlier in the chapter. In actuality, Kata Containers are *really* lightweight virtual machines designed to be managed like containers. The benefit of Kata Containers over regular containers is the increased security from the hardware-enforced separation provided by the hypervisor. This discussion of Kata Containers is based on the 1.2.0 release.[33]

Kata Containers uses the KVM hypervisor and works seamlessly with Kubernetes, Docker, and OpenStack. Other hypervisor support is being built and may even be available as you are reading this. Kata Containers is comprised of six different components, as shown in Figure 4-16.

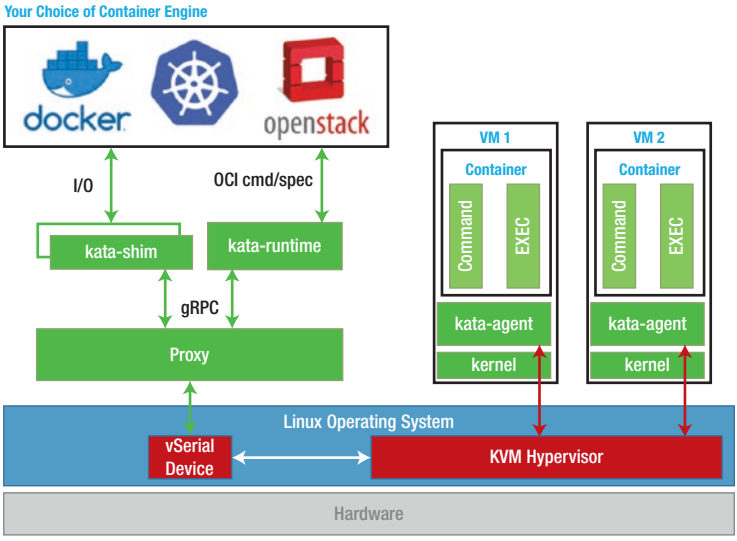

Figure 4-16. *Kata Containers architecture*

[33]Kata Containers, https://katacontainers.io and https://github.com/kata-containers/documentation

- The **Runtime**, called the `kata-runtime`, handles all the Open Container Initiative (OCI) commands used to create and configure a container. It also starts the Shim instances. The `kata-runtime` utilizes the `virtcontainers`[34] project to perform the heavy lifting in a platform agnostic way. Whenever an OCI command is run on a container, the Runtime creates a new Shim to connect the container engine to the container.
- The **Shim**, called the `kata-shim`, is an interface between the container engine (like Docker, Kubernetes, or OpenStack) and the created container inside the virtual machine. The container engine has a process (called the Reaper) that monitors the container, manages the container, and *reaps* the container when it dies or must be killed. Because Kata Containers are inside a virtual machine, the Reaper cannot actually access the container, due to the hardware separation in place by the VM. The Shim pretends to be the container, so the container engine can connect to it for management, and the Shim communicates with the actual container using an agent. The Shim links the standard input and output flows and any Linux signals from the container back to the container engine, so the container engine can receive them and process them appropriately.
- The **Agent** (`kata-agent`) is part of the minimal Clear Linux OS image and runs inside the VM; it provides communication outside the VM to the `kata-runtime` and `kata-shim`. The Agent creates a container sandbox

[34] https://github.com/containers/virtcontainers

based on a set of namespaces for a container to run inside. The namespaces include UTC, IPC, network, and PID[35] namespaces. The Agent can support multiple containers running inside a VM (called a *pod*); however using Docker, only one container per pod is supported.

- The **Hypervisor** provides virtualization and is a combination of KVM with QEMU. As shown in Figure 4-17, QEMU is the Virtual Machine Manager (VMM) and creates the virtual machine for the container to run in, populates it with the virtualized kernel, and emulates virtualized devices for the VM. KVM is used to control the VM, and all VM exits return directly back to KVM. The hypervisor provides a virtual socket (VSOCK) or a serial port to communicate with the Shim or Runtime. The serial port is the default, but for Linux kernels beyond v4.8, VSOCK is available. If a serial port is used, gRPC runs over Yamux on the serial port.

[35]See http://man7.org/linux/man-pages/man7/namespaces.7.html, IPC = interprocess communication and message queues namespace, PID = process identifier namespace, UTS = hostname and Network Information Service (NIS) domain name service.

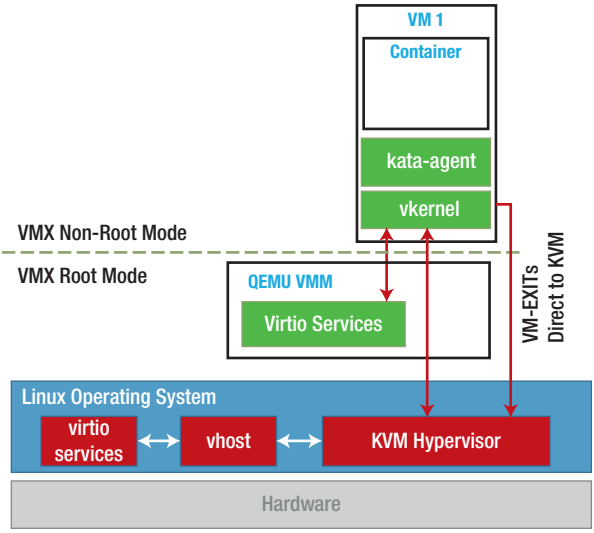

Figure 4-17. *Kata Containers hypervisor architecture*[36]

- The **Proxy** is a multiplexer for the hypervisor if a serial port is used to connect between the Runtime or Shim and the hypervisor. Multiple connections are required because the `kata-agent` can communicate with multiple different `kata-runtime` instances and `kata-shims`; each instance opens its own remote procedure call to the Agent using gRPC, and the Agent connects these to the appropriate container process in the VM. The Proxy is not needed if a VSOCK is used to connect between the Runtime and the hypervisor, since gRPC can run directly over a virtual socket and then gRPC directly handles the multiple different

[36] https://github.com/kata-containers/documentation/blob/master/architecture.md

communication streams. In this case, the gRPC connections from the `kata-runtime` feed directly to the hypervisor over a VSOCK, and the Proxy disappears from the architecture diagram in Figure 4-16.

- The ***kernel*** is the operating system that runs the container inside the virtual machine. The kernel is a highly optimized kernel from Clear Linux with a minimal memory footprint and includes only the services needed to run the container workload. QEMU virtualizes or emulates everything else. The smaller Linux kernel reduces the attack surface presented to the container, further increasing security.

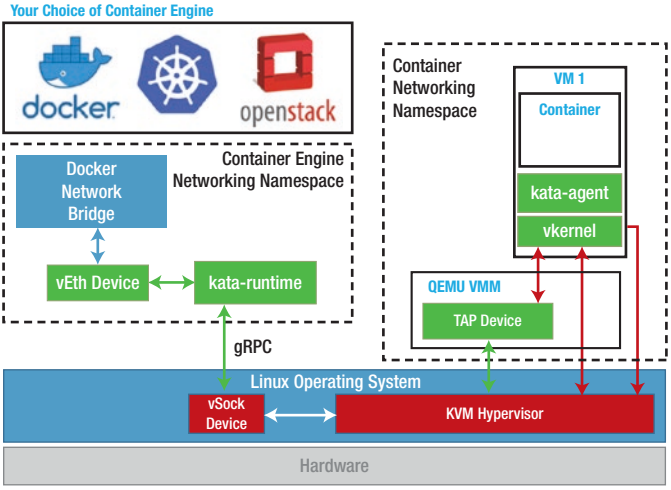

Figure 4-18. *Networking with Kata Containers*

WHAT IS QEMU?

Software Engineering can solve any problem using another layer of abstraction…except for the problem of having too many abstraction layers. QEMU is a special abstraction layer that solves several difficulties with virtualization.

Recall that KVM is a Type 2 hypervisor and part of the Linux kernel. But, KVM uses all the hardware features of VMX, so it is as fast and secure as a Type 1 hypervisor. Kata Containers combines QEMU with KVM for further speed improvements.

QEMU is a virtual machine monitor (VMM) that runs on top of an operating system host, like Linux. QEMU is also an emulator that does binary translation and can even run programs compiled for different CPUs or OSs on that host. So, QEMU is really good at emulation.

In Kata Containers, QEMU quickly boots virtual machines (VMs) for KVM by using emulation. A special version of QEMU provides highly optimized emulators to speed boot time and reduce interpretation of ACPI interfaces. Other emulators provide the root filesystem as a persistent memory device. QEMU also provides *hot-plugging* devices, during the launch process, allowing devices and virtual CPUs to be added to the VM only when needed.

All this speeds the construction of VMs and makes Kata Containers execute really fast.

Connecting containers to a network is accomplished with a virtual network created by the container engine on the host. The container engine connects this virtual network to the real network, using appropriate filters, including a network address filter (NAT). Docker connects the containers

to this network using a virtual Ethernet (veth) device. However, virtual machines normally use a TAP[37] device. The problem in Kata Containers is that all devices are emulated through QEMU, and QEMU does not support veth interfaces. The solution implemented in Kata Containers requires the `kata-runtime` to bridge between the TAP device in QEMU and the host virtual network created by the container engine. Figure 4-18 shows this configuration graphically, with the traffic from the Docker virtual network running through the TAP device emulated by QEMU and then into the container in the VM.

Figure 4-19 shows the series of interactions between the Kata Containers components to create a container in a virtual machine. The `virtcontainers` library as part of the `kata-runtime` essentially does all the work to create the VM, start the Proxy, create the container sandbox that the container will run in, and then create the container, and finally start the `kata-shim` to communicate with the container.

Once the container is created, it can be started and used with the *start* message. In Kata Containers, the *start* message does not create the container as it does with most container engines. The container was already created with the *create* command, as shown in Figure 4-19. Instead, *start* just forwards the *start* message to the `kata-agent`, and the `kata-agent` starts the container's primary process.

[37]A TAP device copies all traffic from the network into the device, just like a "tap" on a phone line.

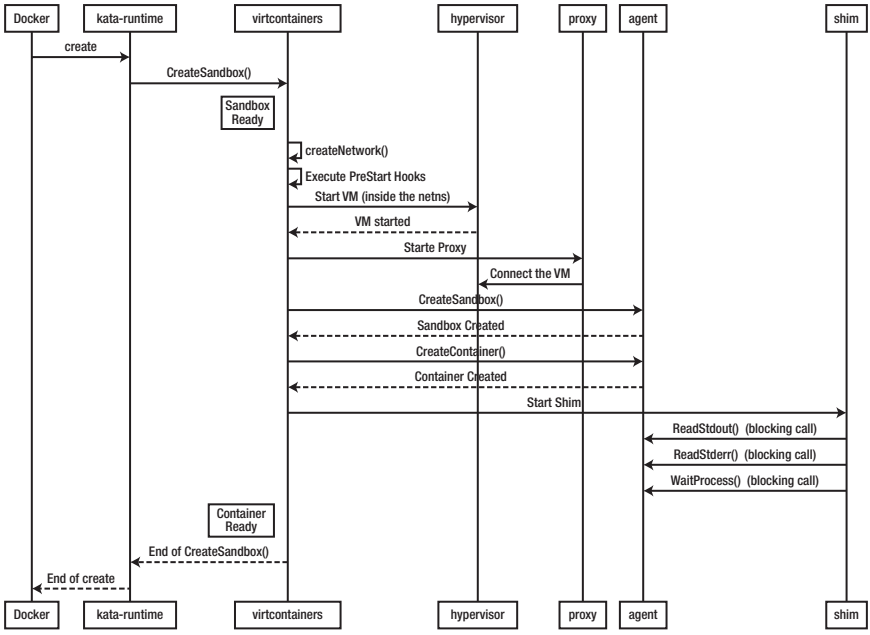

Figure 4-19. *Kata Containers create command*

Kata Containers Summary

With Kata Containers, you can have security and performance. Virtual machines utilize hardware separation to provide increased containment between containers while creatively using different software abstractions to maintain the same software APIs to start up and control the containers themselves. As we saw with ACRN, there are areas of attack that could breech the system, including the QEMU virtualized drivers, and the Kata Shim and Runtime. Table 4-5 outlines the analysis of the Kata Containers against our containment security requirements.

Table 4-5. *Kata Containers Security Summary*

Operating System Security Principles	Grade	Comments
Memory Disclosure by Privileged User	B	Most privileged attackers are restricted from viewing or tampering the VM memory. QEMU and the KVM, including their virtualized devices, remain as potential privileged attackers on the VM memory. Platforms with multi-key total memory encryption (MKTME) can provide protection but should include integrity as well as encryption. See the following article for attacks on encrypted memory: https://arxiv.org/ftp/arxiv/papers/1712/1712.05090.pdf
Memory Tampering by Privileged User	B	
Execution Interference by Privileged User	C	Most privileged attackers are restricted from viewing or tampering with workloads in the VM. QEMU and the KVM, including their virtualized devices, remain as potential privileged attackers on the VM execution, though this is expected and cannot be avoided. However, the Kata Shim and Runtime provide targets for privileged attackers to subvert workloads running in the VMs, and these processes are not protected.

(*continued*)

Table 4-5. (*continued*)

Operating System Security Principles	Grade	Comments
Data Leakage via Side Channels	C	Side channels are most concerning for VMs and container systems, as there may be applications of different trust levels running inside different containers or VMs. Updates to the Linux kernel and microcode patches for the serious side-channel CVEs are available. There continue to be security patches for KVM and QEMU, as late as October 30, 2018. For additional information, see www.qemu.org/2018/01/04/spectre/ and www.redhat.com/archives/rhsa-announce/2018-October/thread.html for Red Hat kvm-qemu patches.
Execution Information Leakage via Side Channels	C	Execution leakage is similar to memory leakage and requires multi-key total memory encryption (MKTME) for protection but must also include integrity protection as well as encryption. Inference of execution is still possible under MKTME if page loads and misses are observable by the attacker. Just as we see in SGX, the operating system kernel remains as a potential attacker here.

(*continued*)

Table 4-5. (*continued*)

Operating System Security Principles	Grade	Comments
Trusted I/O	-	Trusted I/O is not supported in Kata Containers.
Application Flexibility	A	Any application can build into Kata Containers, and the support of many devices through virtual device drivers improves the level of support and flexibility of the Kata Containers solution.

Trusted Execution Environments

Even with the protections afforded to applications by containers and virtual machines, some applications are so sensitive that they require even greater separation protections. Examples of such applications include payment applications that deal with credit card transactions or authentication applications that deal with fingerprints or other biometrics. Trusted Execution Environments (TEEs) are application execution containers that are separate from the operating system and other applications on the platform and provide enhanced memory and execution separation characteristics. TEEs provide containment guarantees that prevent even the administrator or root from interfering with or peeking at the secrets of an application. This section looks at two such TEEs, Intel's Software Guard Extensions (SGX) and Android Trusty.

Software Guard Extensions

Software Guard Extensions (SGX) is a ring 3 TEE, meaning that SGX is directly accessible to applications, and applications running in SGX have the same type of privileges (ring 3) as other applications on the

platform. SGX creates a TEE from a special memory cache and a secure mode of the CPU, removing the entire operating system from the trusted computing base (TCB); this means that unlike other TEEs, SGX does not even depend on secure boot to instantiate a trusted environment. For applications to use SGX however, the operating system must support access to the SGX instructions and the SGX memory cache. Support for SGX is available for Microsoft Windows and many Linux distributions, including Ubuntu Desktop. SGX has not been ported to Ubuntu IoT Core.

An application running SGX is called an *enclave*, and an SGX enclave is actually part of an application. An enclave is built like a dynamically loadable library (DLL) or shared object library (SO), to use Linux terminology. The enclave is loaded by an application and, from the operating system perspective, the enclave is an extension of the process space of the application that loaded it. There are three primary differences between a regular application and an enclave:

- The way the enclave memory is treated
- The way the enclave memory is loaded
- The way the enclave is executed

Memory for an enclave comes from a special pool of memory called the Enclave Page Cache (EPC). EPC memory is encrypted by the processor and is only accessible in SGX mode. Regular applications, or even the operating system, that try to access EPC memory see only encrypted junk. Only when an enclave is executing can the CPU provide the decryption key so the memory page contents can be viewed. Likewise writes to the EPC pages are also restricted. These guarantees are part of what makes SGX mode a TEE. Platforms must allocate memory into the EPC, thus making that memory unavailable for regular applications; a new feature of SGX is the dynamic allocation of pages to the EPC, but this is not supported on all processors yet.

The second thing that makes SGX mode a TEE is the special way that code and data are loaded into an enclave. When a regular application asks the operating system for an enclave to be created, it provides the DLL (or SO) that contains the enclave's code. That code must be signed. We will discuss how the code is signed and with what key in a moment. SGX includes a special loader that verifies the signature on the enclave as it is loading its code and data into EPC memory. If the signature indicates the enclave code is authentic, then the loader activates the enclave and the application can use the enclave functions. If the signature indicates the enclave has been tampered with or is not signed with an authorized key, then the load of the enclave fails. All code and initial data pages loaded into an enclave are verified as authentic, which indicates that the authorized party that signed that code also trusts that code. This makes the code running inside SGX trustworthy and another attribute of SGX as a TEE.

The final thing that makes SGX mode a TEE is the fact that it is a special mode of the CPU, and this creates execution separation between regular applications and enclaves. The execution of enclave code within SGX is separate from execution inside the operating system and the execution in applications. There have been side-channel attacks on SGX, just as there have been on other execution modes. This is a result of some shared micro-architectural state and shared cache state; there is also a dependency from SGX on the operating system to load and manage memory pages which creates another type of side channel. Changes to the CPU microcode have addressed the attacks that are known, and further changes are being made to hyperthreading mode to address additional issues. We talk more about this in the section on threats later.

Creating code that can be run as an enclave requires a special key. This is because the enclave code must be verified by the SGX launcher when the enclave is loaded. The SGX launcher uses a key set by the BIOS during boot to verify enclave programs. If an enclave is signed using a key which itself is signed by the SGX launcher key in BIOS, then the enclave is trusted. By default, the BIOS includes an Intel key. Intel will sign an enclave developer's key after they submit a formal request and fill out some paperwork. This means that any developer with such a key could run enclave code on any platform with an Intel processor. Intel realizes this should be a bit more controlled, so they allow the owner of the platform to change the key in the BIOS to a key of their own. This means that the platform owner can change the behavior of the SGX launcher to approve only enclaves that they themselves trust; this is done by changing that BIOS key.

SGX is a powerful mode on Intel processors that provides a trusted execution environment to applications. This gives applications the ability to put their most sensitive code inside a trusted execution container and keep the operation of that code, and any secrets that the code uses, away from other applications and even the operating system.

SGX Security Summary

Table 4-6 provides a summary of the SGX system security features for comparison with other systems.

Table 4-6. *SGX TEE Security Summary*

TEE Security Principles	Grade	Comments
Memory Disclosure by Privileged User	A	SGX uses a separate memory cache that is encrypted by the CPU and is separated from the operating system and other applications.
Memory Tampering by Privileged User	A	SGX mode prevents access to memory pages in the EPC unless an SGX application is executing, which locks out other applications and the operating system from tampering with the memory. Page attributes are set and locked at page set up time when the enclave is loaded.
Execution Interference by Privileged User	B	The operating system still controls the page tables, including allocation of pages and page eviction; a misbehaving OS can perform a DOS on an enclave and perform some side-channel attacks using the enclave's usage of pages. Protection of secrets within an SGX enclave still requires the use of constant-time programming constructs and careful use of pages and cache to avoid such side-channel attacks.

(*continued*)

Table 4-6. (*continued*)

TEE Security Principles	Grade	Comments
Data Leakage via Side Channels	C	Research on SGX side channels, including L1 Terminal Fault, have been reported.
Execution Information Leakage via Side Channels	B	These are a result of microarchitectural side channels in the CPU itself. CPU patches are effective in mitigating most of these attacks, other than hyperthreading-based attacks.[38] Other options including forcing CPU core scheduling are potential solutions. (https://www.usenix.org/system/files/conference/atc18/atc18-oleksenko.pdf)
Trusted I/O	-	Trusted I/O is not supported in SGX.
Application Flexibility	A	Any application can contain enclave code which can be loaded into SGX. Commercial development of enclaves requires a key from Intel, or the platform must be set up with a special SGX launcher key.

Android Trusty

The Trusty TEE[39] is an offering from Google that includes an operating system, a set of drivers for Android to communicate with Trusty, and APIs for applications to use applications running in Trusty. Trusty is an interesting TEE that has some very different properties as compared with Intel SGX.

[38] https://software.intel.com/security-software-guidance/api-app/sites/default/files/336996-Speculative-Execution-Side-Channel-Mitigations.pdf

[39] Google. "Trusty TEE." https://source.android.com/security/trusty

The first primary difference is Trusty is designed to operate on a completely separate processor from the main processor running the untrusted operating system. Trusty uses its own memory management unit (MMU) and provides virtualized memory for all the trusted apps running in Trusty. All the applications must be single threaded, though multithreaded applications may be provided in a future update.

The next significant difference with Trusty is that it can have access to devices, platform keys, and other resources and give access to those resources to Trusty applications. Since SGX runs in ring 3, it does not have privileged access to devices and does not currently have a trusted I/O mechanism.

The last difference in Trusty is that trusted applications are compiled into Trusty and run as an event-driven server. Applications cannot be added dynamically into Trusty – they must be designed and built into the Trusty kernel. And each trusted application running in Trusty is accessible to any application in the untrusted operating system; Trusty applications are not bound to a particular process in the untrusted processor.

Trusty TEE Security Summary

Trusty is an interesting TEE that provides significant trust for platform developers, but it does not expose the capability for end users to create their own trusted applications.

Table 4-7 provides a summary of the Trusty TEE security features for comparison with other systems.

Table 4-7. *Trusty TEE Security Summary*

TEE Security Principles	Grade	Comments
Memory Disclosure by Privileged User	A	Because Trusty uses a physically separate memory from the untrusted operating system and its own MMU, disclosure and tampering are avoided. The drawback is the additional HW cost for this separation.
Memory Tampering by Privileged User	A	
Data Leakage via Side Channels	A	
Execution Interference by Privileged User	A	Because Trusty uses a physically separate processor (or physical core) from the untrusted operating system and its own MMU, disclosure and tampering are avoided. The drawback is the additional HW cost for this separation.
Execution Information Leakage via Side Channels	A	
Trusted I/O	B	Trusted devices are built into the system and allocated when Trusty software is compiled.
Application Flexibility	D	Only the applications built into the Trusty software are available – no dynamic loading of software applications or services is possible.

Containment Summary

In this section, we reviewed different types of application containment, ranging from software-only containment using containers, hardware-assisted containment with virtual machines, hardware TEE with encrypted memory and special processor state with SGX, and full hardware separation TEE with Trusty. The more hardware used in the containment

solution, the greater the level of security provided by the solution. However, there is a balance to be had, as we saw with Trusty, because software is more flexible than hardware. A full hardware solution, while more secure, creates limitations to what can be accomplished and what usage models applications can execute.

Network Stack and Security Management

This section signals the shift in our chapter from platform software to the management plane. Networking and connectivity are vital to an IoT system, and therefore the entirety of Chapter 5 is devoted to this subject. We leave the discussion of the network technologies and protocol stacks, including the threats, to that future chapter. However, before we leave the networking topic completely, we want to cover an important software library for network packet processing, the DPDK, as well as a few software packages that make security management easier.

Intel Data Plane Development Kit

The Data Plane Development Kit (DPDK) is a set of software libraries and device drivers that make constructing software networking stacks with advanced features very easy and very performant. We talk about the DPDK because it is a useful component to speed the development of **end-to-end security** features and in the implementation of network security policies to enforce **network restrictions**. This library exposes the features and capabilities of network cards into ring 3, enabling better performance when processing packets at high speed. This is important for edge devices implementing industrial control loops, because the DPDK allows software to reliably receive and send packets within a

minimum number of CPU cycles. The Linux Foundation[40] provides a downloadable version of the DPDK, and Intel contributes specialized features and drivers that directly leverage Intel silicon performance. The DPDK boosts packet processing throughput and provides multicore support, facilitates processing of packets in user space (ring 3) to avoid costly transitions between user and kernel space, and enables direct access to devices for high-speed IO.

The latest DPDK version is 18.05 and supports the following features:

- Support for multiple NIC cards, including virtualized drivers
- Support for cryptographic operations in cryptodev library
- Support for event handling in the eventdev library
- Baseband wireless in the bbdev library
- Data compression support in the compressdev library (new in DPDK 18.05)

Figure 4-20 shows the architecture of the DPDK library.

[40] https://www.dpdk.org/ and documentation is available online at http://fast.dpdk.org/doc/pdf-guides/

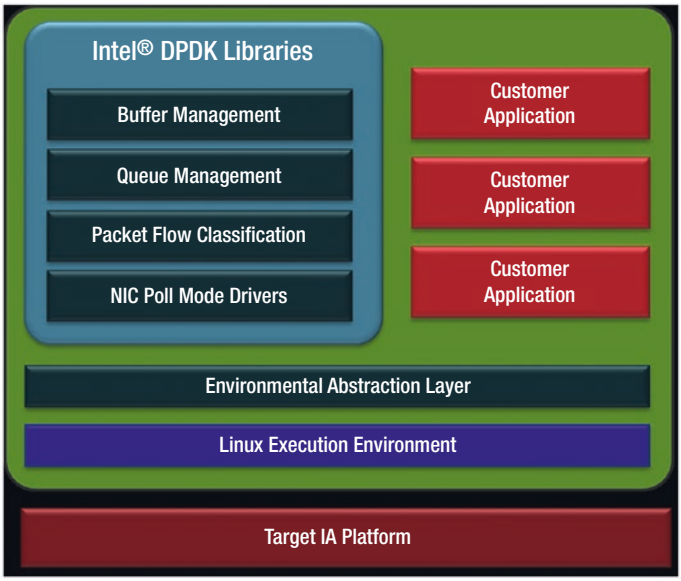

Figure 4-20. *The Intel DPDK library structure*

The DPDK is very comprehensive and supports multiple hardware capabilities across Intel, AMD, ARM, NXP, and other hardware manufacturers. In keeping with our theme in this chapter, let us review the security capabilities and the Intel-specific hardware features that are supported through the DPDK.

The DPDK supports standard modes for the Advanced Encryption Algorithm (AES), including Cipher Block Chaining (CBC) mode, Electronic Code Book (ECB) mode, Counter (CTR) mode,[11] and a special mode used primarily for block storage devices, XTS[41] mode. All modes are supported

[41]XTS is actually considered a tweak cipher, a modification of the underlying cipher using parameters. XTS stands for XEX-based tweaked-codebook mode with ciphertext stealing. XEX is a tweak cipher mode, which stands for XOR-Encrypt-XOR, which was designed by Phillip Rogaway, 2004, "Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC," http://web.cs.ucdavis.edu/~rogaway/papers/offsets.pdf

with the standard key sizes, 128-bits, 192-bits, and 256-bits. DPDK also supported DOCSIS encryption and DES and 3DES.[42]

In addition to encryption, the DPDK supports hashing algorithms using the SHA2[43] algorithms, SHA2-256, SHA2-384, and SHA2-512. The older SHA1 algorithm is also supported, but should only be used for interoperability reasons; the use of SHA2-256 should be the minimum requirement for IoT systems.

The DPDK supports the Intel SHA-NI and AES-NI instructions (see Chapter 3, section "*CPU hosted Crypto implementations"),* providing access to hardware acceleration of these algorithms. In addition, AES-GCM, the Galois Counter Mode of AES, is further enhanced by combining the AES-NI instruction with carryless multiplication instructions to speed performance of the Galois integrity tag calculation.[44]

The DPDK provides compatibility with other software and hardware implementations of cryptography, even providing a full software implementation using the OpenSSL open source cryptographic library. Using these different plugins for the DPDK cryptodev library, a fully portable application can be built that makes use of the best hardware features the platform has to offer. The use of the DPDK allows applications

[42]DES, Data Encryption Standard, and 3DES, Triple Data Encryption Standard, are older modes included only for interoperability. It is strongly recommended to avoid use of these modes unless they are required for interoperability. In IoT systems, there is no good reason to use such modes.

[43]SHA, Secure Hash Algorithm, are algorithms defined in the NIST Secure Hash Standard (SHS) for cryptographic hash algorithms. The older version SHA1 is deprecated for most uses today. The SHA2 family of algorithms, https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf, covers multiple hash digest bit lengths.

[44]https://software.intel.com/en-us/articles/intel-carry-less-multiplication-instruction-and-its-usage-for-computing-the-gcm-mode and https://software.intel.com/en-us/articles/aes-gcm-encryption-performance-on-intel-xeon-e5-v3-processors

direct access to the best cryptographic acceleration hardware of the Intel platform, and compatibility to other platform's cryptographic accelerators as well.

Security Management

Security management is the combination of active processes and executed procedures during installation, configuration, operation, and decommissioning of systems that preserves the confidentiality, integrity, and availability of those system and network resources for the approved mission(s) of the organization. This chapter focuses on software, not processes and procedures. However, there are software tools and agents that directly aid the security management process. We look at a few of those here, just for completeness in our discussion.

Secure Device Onboarding

The very first issue requiring a solution in an IoT system is **device provisioning** or how to provision devices so they can connect to the correct back-end cloud system or device management system. A common solution is to preprovision devices during manufacturing to connect to a specific cloud agent. Microsoft Azure Sphere uses this approach. While it works, that solution locks the device into a specific cloud, and the approach can have impacts on high-speed manufacturing. A better approach is to provide flexible and secure onboarding for any device to any cloud system. Intel's Secure Device Onboard (SDO[45]) provides this security capability using an EPID[46] device identity key. Figure 4-21 shows the provisioning lifecycle of a device, from manufacturing to installation. This can be any device from a gateway or server down to a smart sensor.

[45] https://software.intel.com/en-us/secure-device-onboard

[46] See the discussion of Intel's Enhanced Privacy ID (EPID) in Chapter 3.

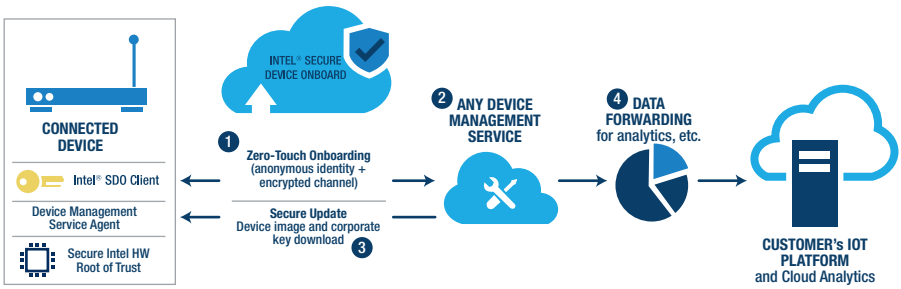

Figure 4-21. *Intel's Secure Device Onboard preserves device privacy and provisions "Any Device to Any Cloud"*

SDO utilizes a few hardware security features to construct this high-level service, including

- The platform's root of trust containing an identity key; an EPID group signature key is the preferred identity key, since it provides privacy for the device installation, but an RSA or ECDSA key may also be used.
- The Intel SDO Client firmware executing inside a TEE; SDO currently uses the CSME discussed in Chapter 3 for its TEE, but SGX or Trusty are alternative TEEs for SDO.
- Secure storage on the device to hold the manufacturer's public key, a GUID, and an ownership credential.

During manufacturing, a digital record of the device is created, which is referred to as the ownership credential. The ownership credential includes the device's unique identifier (GUID) and the owner's public key; the owner credential is signed by a private key belonging to the manufacturer and includes an integrity checksum to prevent modification or forgery of the ownership credential. The manufacturer endorses the ownership credential by digitally signing it with the manufacturer's private key when the device is sold. This endorsement can be repeated in the

supply chain, allowing a deferred binding between the credentials stored in the device and those of the device management service (e.g., running within a particular AWS account) who will control the device in operation.

When the device is installed (Step 1 in Figure 4-21), the device contacts Intel's Secure Device Onboard Rendezvous server and is connected with the device management service which was specified by the device's owner. As a precursor to the device install step, the preferred device management server must have been registered with the SDO Rendezvous service by the device owner using the ownership credentials. The SDO protocol between the device and Rendezvous server validates the ownership credential, as well as the authenticity of the device and the Rendezvous server to each other. At the end of the SDO protocol, the device is forwarded to the proper device management service to complete provisioning (Step 2 in Figure 4-21), allowing the device management service to install a management agent on the device. SDO prevents unauthorized entities from taking control of the device and gives the end customer flexibility to provision the device to any management service or cloud back end. The device management service can then update the device with new software and link the device to the preferred back-end cloud system (Steps 3 and 4 in Figure 4-21). Intel SDO can also be reactivated by the device owner at any time, allowing the device to be reprovisioned or for device resale.

Intel Secure Device Onboarding solves the first problem an IoT device encounters – how to securely connect to the right back-end service for management and operations. Using hardware security elements inherent in the platform, SDO provides a low-cost and flexible solution with high security.

Platform Integrity

Once a device is provisioned, maintaining the integrity of the platform software is vital to keeping an IoT system operating. **Platform integrity** means ensuring that a device has booted the platform software intended

by the system and that the platform firmware, boot loader, and operating system have not been corrupted. Device management software can query the platform's integrity state and determine if something needs to be updated or remediated. But, some software element must reside on the device to calculate the platform integrity and then communicate it up to the device management software in a meaningful way.

In Chapter 3, we discuss protected boot technologies included in Intel platforms, including PTT[47] and TPMs. These hardware elements use software in the operating system, boot loader, and BIOS to measure the platform during boot. These measurements are stored in hardware-protected storage in PTT or the TPM. The software to access these measurements is included in the trusted services stack (TSS) that was written according to the Trusted Computing Group's (TCG) specification for TPM2. As shown in Figure 4-22, this software stack is comprehensive and, besides platform integrity measurement, includes features for other TPM operations including encryption, key generation, secure storage, and attestation. The application-level APIs are all provided in the System API (SAPI)[48] or the Enhanced SAPI (ESAPI)[49] and are defined by the TCG; the FAPI is still under development. The Feature API (FAPI) would be the easiest to use and abstracts many details of the TPM from the application, while the SAPI provides near-transparent use of the TPM commands and responses.

[47]Platform Trust Technology (PTT) and Trusted Platform Module (TPM)

[48]https://trustedcomputinggroup.org/wp-content/uploads/TSS_SAPI_Version-1.1_Revision-22_review_030918.pdf

[49]https://trustedcomputinggroup.org/wp-content/uploads/TSS_TSS-2.0-Enhanced-System-API_V0.9_R03_Public-Review-1.pdf

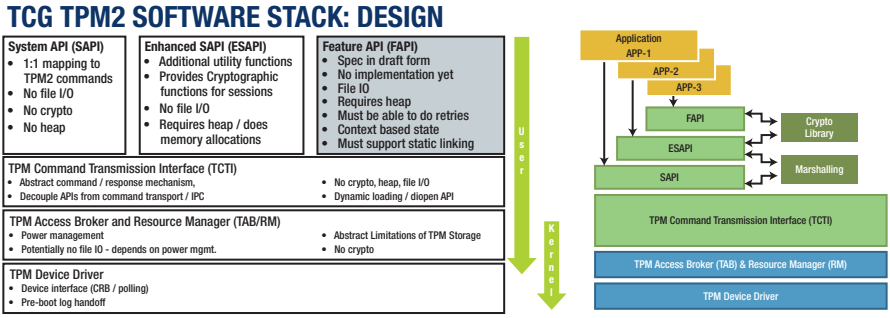

Figure 4-22. *Intel TPM2 software stack (TSS)*[50]

Network Defense

IoT systems are all about communication, and without some type of defensive measures, these IoT devices would be easy targets for network attackers. Common network defense capabilities including firewalls and intrusion defense software are important to add to any IoT device. Some devices are so small and so resource constrained that no attempt is even made to add any network protections. However, there are tools that can provide some reasonable protections and should be considered.

The first step of **network defense**, of course, is to limit the applications and services that open ports to listen for connections. In fact, if your IoT device is so resource constrained that you are considering putting no network defenses on the device, then there should be no listening services either – only outgoing connections. But because firewalling is the most basic defense, a program that intercepts the incoming network traffic to check for anomalies is important and should be considered.

On Linux distributions, the recommended program for network defense is TCP Wrappers. This program can be called from inetd or configured into the hosts.allow and hosts.deny configurations. TCP Wrappers allows the system to be configured to allow or deny connections based on the network

[50] https://software.intel.com/en-us/blogs/2018/08/29/tpm2-software-stack-open-source

address and protocol. Additionally, other commands can be executed when rules in the TCP Wrappers configuration are triggered, such as sending an alert email or adding an entry to the syslog. Configuration of the TCP Wrappers file can provide extensive filtering and can be set up so that normal traffic and operations easily get through without any overhead. Other options for firewalling include directly using the kernel netfilter or configuring the netfilter through ipchains. Significant material is available both on the Web and in Linux books, so that information will not be repeated here.

Finally, good logging for what is happening on the network and on a device is vital to reconstructing an attack or understanding an attempted intrusion. There are numerous programs for **attack detection** that operate on both Linux and Windows and can be compiled for other operating systems as well. TCPdump and snort[51] are common programs for detecting network intrusions or malformed packets on a device. Snort can be turned into a full-scale network intrusion detection system where devices capture traffic and send dangerous looking packets to a central server for deeper analysis. Suricata is a similar robust open source solution for intrusion detection. These types of intrusion detection system are very useful for IoT system for early detection of attacks and fast response to prevent such attacks from bringing down the IoT system.

Platform Monitoring

Security management includes monitoring a device and its workload for anomalies, in the event that a network attacker is able to circumvent the network defenses in place on the device. The monitoring functions are tied into the device management agent on the platform, allowing problems to be reported back to the management servers.

In the section on Zephyr, we discussed watchdog timers used to monitor for long running privileged threads. Remember the problem in Zephyr was a privileged thread that does not yield back to the operating

[51] https://www.snort.org/

system which can then starve out the execution of other processes on the system. The operating system can prevent this by using a hardware timer started before releasing control to the high-privileged thread; if the thread does not yield back in a certain amount of time, the hardware timer causes a non-maskable interrupt (NMI) that stops execution of everything else and returns control back to an interrupt service routine (ISR) in the operating system. When the operating system regains control, it can terminate the offending thread and report the situation back to the management service. Sometimes this doesn't work. It often fails because the attacked thread had enough privileges on the system, allowing the attacker to disable or continually reset the timer, effectively disabling the watchdog.

There are other unique options for performing platform monitoring that can identify side-channel attacks or threads that have potentially been corrupted by network attackers. Several techniques are published[52, 53] that utilize hardware performance counters in the CPU microarchitecture to characterize and monitor software and detect when attacks are likely present. This information can be used to shut down the attacking threads or reboot the system into a known good state.

McAfee Embedded Control

There is one last software capability that deserves mention in security management that provides some unique **system authorization** capabilities. McAfee Embedded Control (MEC)[54] is a software program

[52]A Survey of Cyber Security Countermeasures Using Hardware Performance Counters, https://arxiv.org/pdf/1807.10868.pdf

[53]Cache-Based Side-Channel Attacks Detection through Intel Cache Monitoring Technology and Hardware Performance Counters, https://hal.inria.fr/hal-01762803/document

[54]https://www.mcafee.com/enterprise/en-us/products/embedded-control.html

that provides extended access control and integrity to IoT platforms. MEC protects both executable as well as data files on a platform, ensuring that those files are not accidentally or maliciously modified, even by a user with administrative rights. MEC creates a new privileged user on the platform that is only accessible through the MEC admin interface and manages a database of integrity checksums over directories and files specified by the MEC admin. MEC includes an augmented launcher that is integrated with the Windows or Linux operating system, allowing MEC to check the integrity of executable files before launch. The access control database allows MEC to also specify what services and devices an executable can access, providing even stricter control on running applications. This means that even if a program were maliciously corrupted at Runtime, the attacker would not be able to use unauthorized system resources, and ROP or JOP attacks would only be able to modify the use of authorized system resources, not fundamentally change the resources to which the program has access.

MEC creates a very powerful protection for IoT devices, and this system works extremely well when the platform's software and configuration does not change regularly. MEC can be integrated easily in McAfee ePO device management suite as well (see the discussion in the *"Device Management"* section). In some versions of MEC, dynamic protection of memory is also provided, limiting the effect of buffer overflows. A limited version of MEC is included in Intel's IoT Gateway Software Suite,[55] and McAfee continues to add improvements and support for other operating systems in MEC. Upgrading to the fully featured McAfee Embedded Control Pro from the basic MEC version included in Intel's IoT Gateway is a smooth transition, fully supported by the MEC admin interface.

[55] https://shopiotmarketplace.com/iot/index.html#/details?pix=58

Network Stack and Security Summary

In this section we looked at various software components that can provide effective **network defense** and **attack detection**, and even be used to build comprehensive **end-to-end security** using the cryptographic library in the DPDK. Common IoT problems like **platform integrity**, **device provisioning,** and **system authorization** can be solved using specialized packages like the TSS, Intel SDO, and McAfee Embedded Control. While these problems cannot be solved for free, the cost in additional compute resources and Runtime RAM may likely provide the difference between a platform that is regularly being attacked and draining the maintenance and remediation budget and a platform with adequate tools and packages that is resilient to attack.

Device Management

IoT systems are composed of thousands of devices, and with so many devices, manual management is prohibitive. In other cases, IoT devices are physically located in remote or difficult-to-reach locations, increasing the cost of sending out a repair person in a "truck roll." Autonomous device management using a cloud-based management solution is essential to preserving an IoT system's return on investment (ROI).

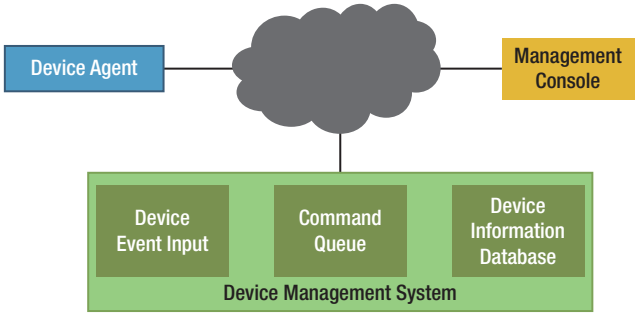

Figure 4-23. *Notional cloud-based device management system*

Cloud-based device management systems include a few common elements, as shown in Figure 4-23. On the device, a management agent performs the actions requested by the management system and also provides data to the management system on the device's health. How such an agent is installed on a device can sometimes be an issue; however **device provisioning** solutions like SDO covered in a prior section provide convenient solutions to this issue. A management console, normally implemented as a browser-based web application, retrieves data from the device management system and presents usable information to system administrators, allowing admins to schedule maintenance, perform actions on groups of devices, or even dive into details of a specific device to troubleshoot problems or investigate trouble tickets. The actual device management system in the Cloud is what separates different systems. Generally, each management system must have three elements:

- A Device Event Input queue allowing devices to provide status and report problems
- A Command Queue allowing administrators to push out commands to devices
- A Device Information Database containing information on each device in the system

The security services that device management system must provide include

- **Authentication**: Ensures that both devices and administrators on the device management system are who they claim to be. Cryptographic credentials issued to these parties are essential to maintaining proof of identity for all entities on the system.

- **Authorization**: Commands to devices can be disruptive to the services provided by the IoT system, or even potentially destructive to the device itself. A reboot command to several devices might cause a temporary denial of service, but a forced operating system update with corrupt software could bring down a system for days or even months.
- **Confidentiality and Integrity**: Although data sent via device management systems do not typically include personally identifiable information (PII), the commands and device health data can contain sensitive information. Integrity of this data is vital to prevent tampering or accidental corruption of the data in transit, but confidentiality may also be warranted depending on the information contained in commands and data updates from devices.
- **Nonrepudiation**: Guaranteed proof of source attached to health data or even the collection of other environmental data around devices could be crucial to the IoT system. Guaranteeing data originated from a particular device is part of data provenance.
- **Defense in Depth**: Is a layering of defenses to protect system elements from hacking. This includes attacks on the devices, gateways, and on the cloud systems and management consoles. Because the device management system represents the most significant network attack surface, and many of the software elements attached to the device management agents require elevated privileges to perform their operations, the device management system itself

must be constructed to prevent attackers from gaining control over the IoT systems. Careful attention to the construction of both the device agent and the interfaces and APIs presented by the cloud system is necessary to prevent successful attacks.

This section reviews two different device management systems, one designed for small to medium deployments (Mesh Central) and one designed for large deployments (Helix Device Cloud).

Mesh Central

Mesh Central is a device management solution appropriate for small- to medium-sized IoT deployments. Mesh is an agent technology, which means that each managed device must be running the Mesh Agent software component. Mesh allows a Mesh Administrator to gain remote access and control of their devices through a variety of means, including direct shell access, dashboards, and connection via custom web applets. Mesh also provides peer-to-peer (i.e., Machine-to-Machine [M2M]) interactions, allowing devices to communicate directly to each other, without a human administrator being involved; this enables the IoT M2M type actions for true IoT automation.

Mesh Central is an Intel open source project and has a wide array of services targeted for remote monitoring and management of computers and devices. Users can manage all their devices from a single web site, no matter the device location or the device position behind routers and proxies, and this is all possible without needing to know the device's IP addresses. Mesh works by having each device generate a new unique RSA key pair, and the hash of the public key becomes the device's identity. Mesh devices register to the Mesh Central Cloud by communicating to devices around them and finding a path to the Mesh servers. This information is found in a signed policy file that is shared among the

devices; however, this requires that devices be preprovisioned with a Mesh client and a policy file, otherwise, and IP address and a path to the device are needed for solutions like SDO to work properly.

The following is a partial list of the actions a Mesh Administrator can do to their connected devices:

- Opening a shell to run commands directly on the device
- Opening the device's graphical desktop, displaying the device's GUI, and providing mouse control on the device
- Installing, removing, and updating software on the device
- Activating a particular piece of software on the device or sending commands to that software (as if on a terminal on the device)
- Viewing files or logs on the device

The following is a brief list of Mesh Central architectural elements:

- Each device is referred to as a node and is identified by a secure, provable identifier based on a self-generated (device-generated) RSA public key.
- Nodes are organized into an overlay network, meaning routing of Mesh messages occur from the Mesh server to the device, but potentially hopping from one device in the Mesh to another device in the Mesh in order to reach the actual destination device; this path may traverse different communication networks connecting each device.

- Agent and Server APIs are available for generic, secure messaging for Admin-to-Device and Device-to-Device messaging.
- Agent Software Update is provided over-the-air (network) using signed and verified updates.
- Direct Connection from an admin web browser (via web sockets) directly to devices for custom applications in the browser to interact with, query, or control devices.
- A Mesh Developer API to add custom actions into the Mesh Agent running on devices.

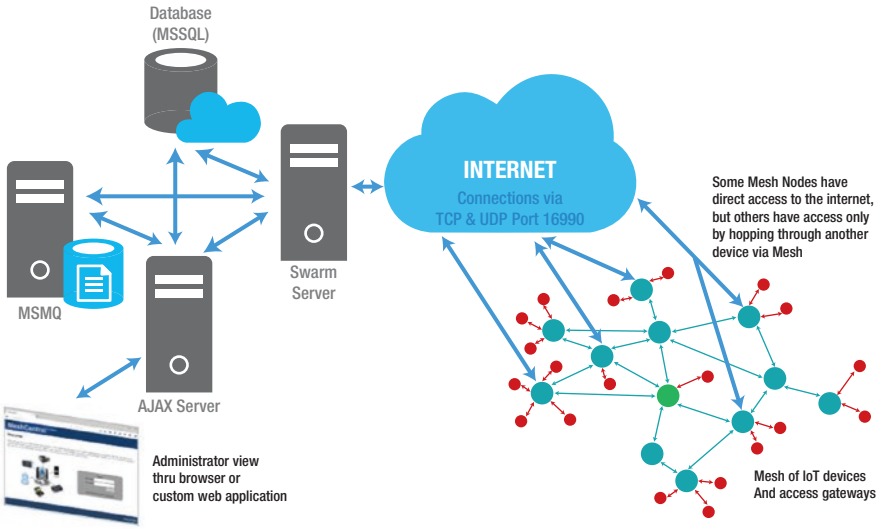

Figure 4-24. *Mesh Central device management system architecture*

As shown in Figure 4-24, Mesh Central is actually composed of four different servers:

- The AJAX Server: Provides the primary interface for Mesh administrators
- The Swarm Server: Provides the primary interface for devices into Mesh
- The Database: Usually Microsoft SQL Server, stores data about devices
- The MSMQ: Provides message delivery among servers

Mesh operates by having a bit of software, called the Mesh Agent Software, on every device. This agent runs under a privileged account on the device so that it is able to perform management on the device (e.g., run software, install applications and services, activate hardware, etc.). The Mesh Agent also has a configuration file, called the Mesh Agent Policy File, that controls what the agent is allowed to do and information about the Mesh control server.

Table 4-8. *Mesh Central Device Management Analysis*

Device Management Security Principles	Grade	Comments
Authentication	C	Mesh requires devices to generate their own identity keys in software and then registers devices to the Mesh Swarm server without any device attestation or proof from a hardware root of trust. This forces device administrators to know the hostnames of devices that should be registering and ignore or boot off devices they do not trust.

(*continued*)

Table 4-8. (*continued*)

Device Management Security Principles	Grade	Comments
Authorization	C	Authorization of commands requires an additional key be shared from the Mesh Administrator, because commands are not protected end to end, only hop to hop. Without this additional layering of authorization (not natively provided by Mesh), commands could be forged by a rogue Mesh node.
Confidentiality and Integrity	A	All messages traversing the Mesh are protected with strong integrity and confidentiality, and session keys are regenerated frequently. Protections are only provided hop to hop, however, not end to end.
Repudiation	D	Mesh does not leverage a hardware root of trust, so all keys are software generated. While all the right actions (e.g., encrypted messages, RSA identity keys, verification by clients) are performed, there is no protection of credentials on the device if an attacker were able to compromise software on one of the systems.
Defense in Depth	D	Mesh runs the Mesh Agent as root by default; significant rework of the client software is required to segment high privilege tasks to protected software agents.

Wind River Helix Device Cloud

Helix Device Cloud (HDC) is an IoT device management solution by Wind River. HDC is able to connect to IoT devices and gateways, manage device-generated data, automatically respond to device events, and perform remote (OTA) software updates. HDC includes a significant back-end system using Kafka that enables intelligent autonomous management of devices and provides easy and secure device onboarding and provisioning through Intel Secure Device Onboard (SDO). HDC adds an agent protocol called DXL (Data Exchange Layer) to each edge device that enables intelligent processing of data and secure end-to-end communication.

With Helix Device Cloud, administrators can

- Maintain secure two-way connectivity to gateways and devices
- Perform flexible data collection to the Cloud and even distribute that data across multiple edge nodes using DXL's powerful edge capabilities
- Receive immediate notification of device issues and use HDC Agent tools for remote diagnosis and repair
- Securely onboard new devices using SDO and upgrade new devices when first activated in the field
- Push new updates out to connected devices
- Collect and import data from IoT devices directly to enterprise systems

HDC focuses on the device management and edge aggregation services; HDC does not address applications and data analytics, but provides mechanisms for these services to reach devices through HDC using McAfee ePO plugins and call-outs to external services. For a more complete overview of HDC, see the HDC Overview whitepaper.[56]

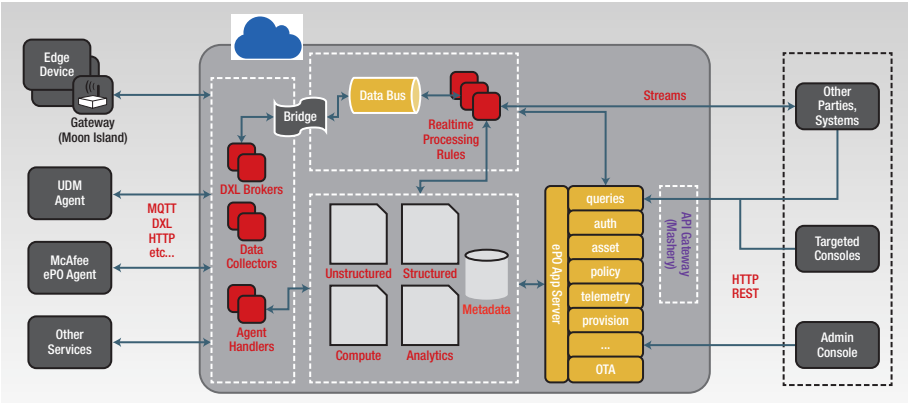

Figure 4-25. *Wind River Helix Device Cloud device management architecture*

Figure 4-25 shows the architecture of HDC. Devices connect to HDC Cloud using HTTP, DXL, or MQTT[57] protocols, and enterprise services leverage the data in HDC through a set of REST APIs exposed by HDC on the back end. Within HDC, there are three primary components: the device connection protocols, the data bus that organizes and routes messages and events, and the database that holds structured and unstructured data, analytics, metadata, and compute workloads. A fourth

[56] https://software.intel.com/en-us/iot/cloud-analytics/cloud-helix

[57] MQTT – Message Queuing Telemetry Transport is an ISO standard protocol based on the publish-subscribe design pattern. MQTT is described in more detail in "Message Orchestration" section.

part of HDC provides an extension interface to add features to HDC using the same extension interface as McAfee ePO,[58] allowing them to leverage each other's extensions.

One of the most interesting elements of HDC is the data bus and real-time processing rules. HDC utilizes an open source topic organization server called Kafka. With Kafka, incoming messages from devices are filtered through a set of rules to determine appropriate actions. Actions can include storing the message data into a part of the database, passing the message off to an ePO plugin, generating an alert to an administrator, or even activating some compute element in the database to create an immediate response to the reporting device. In fact, with the Kafka rules, multiple actions can be executed as a result of receiving a single message.

HDC is a secure device management system due to its use of Intel Secure Device Onboard (SDO) to provision devices and the use of DXL for secure communication. As discussed in the section on security management, SDO leverages the device's root of trust to authenticate the device during onboarding, ensuring the device is not being spoofed by an attacker. During onboarding, HDC leverages the secure channel authenticated with the device's root-of-trust key to install a new device identity key. DXL uses this new key for authentication back to HDC during TLS session establishment, making all messages passed over TLS authenticated back to the device. SDO also installs a trust anchor key for the HDC server; a trust anchor key is a key that is inherently trusted for a particular purpose. The DXL client stores the HDC trust anchor key so that it can authenticate the HDC server over TLS. On platforms that support the SGX TEE (see the section on software containment), the DXL client uses SGX to protect its identity key and the trust anchor key from attack by any malware that is able to infiltrate the device.

[58]McAfee ePO is an enterprise Policy Orchestration product that provides a unified and centralized management console for security management.

Table 4-9. *Helix Device Cloud Device Management Analysis*

Device Management Security Principles	Grade	Comments
Authentication	A	The device and HDC server are authenticated with RSA key pairs that were established over a secure channel through SDO using a hardware root-of-trust key.
Authorization	A	All commands down to the device are verified as authentic through DXL using a trust anchor key established during device provisioning through SDO.
Confidentiality and Integrity	A	All data and commands are protected over TLS.
Repudiation	A	The strong identity keys established using SDO validate the true device identity and link that to the RSA identity keys. Any actions or data are tied to this identity key and cannot be repudiated.
Defense in Depth	A	DXL uses the SGX TEE to ensure its operations and key material are not compromised, even if the platform is infected with malware.

Device Management Summary

Managing the devices of an IoT system is critical to security. Since all the management services occur over the network, attacks such as device spoofing, message forgery, and data disclosure are all possible. Although basic security protections over messages are possible, in

IoT system, attacks on the devices themselves can compromise key material and lead to questions regarding the provenance of data collected in the Cloud. The use of hardware security capabilities, like hardware root-of-trust keys and Trusted Execution Environments (TEEs), drastically improves the security of device management systems and, due to the lower risk of attacks, reduces the total cost of ownership of IoT systems.

System Firmware and Root-of-Trust Update Service

At the beginning of the chapter, in the "Operating Systems" section, we discussed the update problem. The Linux distributions reviewed in that section had different strategies for solving consistency among the packages and services being updated. However, the section identified a remaining problem regarding firmware updates which is how to gain the required access to firmware on the platform with the ability to perform updates.

Firmware is notoriously difficult to update. It typically resides in flash or other nonvolatile storage that is locked or inaccessible even to the operating system itself. The reason for this inaccessibility is security. Firmware is part of the most trusted parts of a system. The BIOS is the first part of the system that executes during power-on and represents the root of trust of the entire system. Other firmware may implement root-of-trust functions, such as system measurements, secure storage, or attestation reporting. Firmware in the security engines control cryptographic algorithms and keys. Firmware in network controllers (Ethernet, Wi-Fi, Bluetooth, Zigbee, LoRa) have access to all traffic entering and exiting the device and may even have access to cryptographic keys for encrypted traffic.

On personal computer-like systems using BIOS, the standard way to perform secure firmware updates is through the *Capsule Update*.[59] A Capsule Update is a function in the BIOS that is activated by the operating system. The Capsule Update function is provided the addresses of capsules in memory containing updates for certain firmware, and then the system performs a soft reset. When BIOS takes control of the platform, it verifies the capsules in memory, and if they are authentic and appropriate for the platform, the capsules are used to update the appropriate firmware. For Capsule Update to work properly, the operating system must be capable of engaging the update service.

Not all devices support the BIOS Capsule Update. And of course for systems without BIOS, or for IoT systems that do not use standard BIOS, some other solution is required. In these cases, some type of custom update procedure is required; as an example, see the update procedure required for the Infineon TPM, a standard device on many PC platforms (https://www.infineon.com/cms/en/product/promopages/tpm-update/ and https://support.microsoft.com/en-us/help/4096377/windows-10-update-security-processor-tpm-firmware).

Threats to Firmware and RoT Update

Firmware update for IoT systems is being addressed by an Internet Engineering Task Force (IETF) working group named SUITS (Software Updates for Internet of Things). The SUITS working group[60] compiled a detailed set of threats and requirements that systems implementing updates should adhere to.

---

[59] https://software.intel.com/en-us/blogs/2015/06/23/better-firmware-updates-in-linux-using-uefi-capsules and https://software.intel.com/en-us/blogs/2017/02/04/signed-uefi-firmware-updates-in-edk-ii

[60] https://datatracker.ietf.org/wg/suit/documents/ – At the time of this writing, all documents in SUIT are still in the draft stage, but should be approved as full RFCs by the time of publication.

- **Modified/Malicious Firmware Updates**: The first threat considered when updating firmware is corrupted or maliciously modified firmware. If an attacker is able to modify the firmware in transit to the platform, or even during the process of updating the firmware, then the attacker is able to inject features into the device. Accidental corruption is just as dangerous since corruption of firmware during the update process can *brick* a system (cause the system to be permanently broken).
- **Rollback to Old (Vulnerable) Firmware**: The second common threat considered for firmware is rolling back the firmware to an older version. An attacker that is able to force a system to reload an older version of firmware may be able to force an old vulnerability back into the platform, allowing them to take over the system. This is especially dangerous since the platform owner erroneously believes they are protected from that vulnerability and may not be watching for indications of compromise for that particular attack.
- **Unauthorized Update Request**: An often overlooked threat to firmware and RoT updates is the person or entity authorized to update firmware on the platform. Allowing a network attacker to force an upgrade of firmware is problematic. Obviously, an attacker successfully pushing corrupt or invalid firmware into a platform would create a problem, but even pushing a valid firmware update could create instability in the platform or a denial of service. Firmware update mechanisms should validate the entity requesting the update is authorized to do so, either because they

are acting under an administrator account or their request is cryptographically proven to originate from an authorized administrator.

- **Unknown Source of Firmware**: Even if an authorized entity issues the firmware update request, the actual source of the firmware (the firmware code itself) should come from a known and approved source. Firmware that is intended to update an Infineon TPM device should not be written by Broadcom; there are potential exceptions, most notably in cases where an OEM repackages an update for their device (i.e., HP repackaging a TPM update for the devices they manufacture).
- **Application of Incorrect Firmware**: Finally, firmware must be matched to the system model and version of the hardware on which they execute. There can be many different revisions of hardware components, and firmware for one component may not operate properly on a different stepping or version.

Turtle Creek System Update and Manageability Service

Turtle Creek is the code name for an Intel product that manages application and platform updates over the air for Intel® Atom, ARM, Core, and Xeon processors. Turtle Creek allows a system administrator to remotely schedule and deploy software updates and recover malfunctioning systems to ensure business continuity and system availability. It is a cloud-based system that interfaces to many other device management systems, including Helix Device Cloud and Mesh Central which were covered in a previous section.

Turtle Creek is a microservice cloud system where each feature of the system is implemented by a microservice in a container hosted on the Cloud. This allows customized deployment of Turtle Creek features, which include the following capabilities:

- Update of the OS, application, and system firmware on supported platforms
- Recovery of platform software and firmware to known good status (factory reset)
- Control of system restart and shutdown
- Device telemetry reception for device health, data logs, and management messages
- Device diagnostics to execute pre-install and post-install checks
- Rollback recovery for any update
- Device system performance monitoring (e.g., CPU utilization, memory utilization, container performance)
- Centralized configuration manager that stores and retrieves configuration for devices used by all microservices, supporting various formats including XML, Consul database, or name-value pairs
- Comprehensive security using cryptographic signature verification for all packages using the TPM 2.0 for key and secret management and secure MQTT for messaging using TLS with end-to-end mutual authentication based on X.509 certificates.

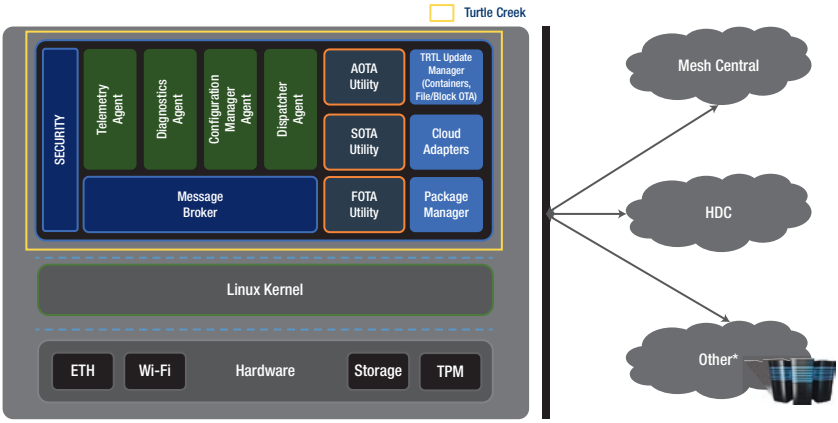

Figure 4-26. *Turtle Creek architecture*

Figure 4-26 shows the architecture of the Turtle Creek client software. Turtle Creek separates updates into three different categories based on the type of update and the repository from which the update packages are retrieved. These include Application Over-the-Air (AOTA), Software Over-the-Air (SOTA), and Firmware Over-the-Air (FOTA). AOTA supports update of application and individual software vendor's services via an update mechanism based on packages and signed RPMs using SMART and Docker container update mechanisms. SOTA supports operating system updates from an OS vendor's repository, which includes the use of the OS standard update mechanisms, like Ubuntu Update Manager and Mender[61] (for Yocto Linux). FOTA supports device or component manufacturer's ability to update custom firmware over the air and integrates firmware-specific functionality to update the device firmware components. The primary mechanism for FOTA support is BIOS Capsule Update.

[61]Mender is a client software embedded in Yocto that enables updates to the operating system to be installed. https://docs.mender.io/1.6/artifacts/building-mender-yocto-image

Turtle Creek's contribution to the IoT platform is twofold. First it unifies multiple disparate platform and software update mechanisms under a single management tool, making the process of managing and distributing updates easier. Second, it incorporates significant security protections on the update process, overlaying them on top of existing capabilities where necessary. Turtle Creek creates a manifest format to convey update commands and requires this update to be signed with a key in the TPM. This satisfies the security requirement for authorization of updates and ensures that the versions and source (repository) for the updates are genuine. If update packages do not include an embedded signature or source authentication, Turtle Creek's manifest can include a detached signature so the actual bits downloaded for the update can be verified that they have not been accidentally or maliciously modified. Table 4-10 outlines a more complete security analysis of Turtle Creek.

Table 4-10. *Security Analysis of Turtle Creek System Update and Management Service*

System Update Security Principles	Grade	Comments
Protect Against Modified Update Packages	A	Turtle Creek enforces RSA signatures over all update packages.
Prevent Update Rollbacks	A	Turtle Creek maintains a database of configured version numbers and packages on each device and ensures rollbacks do not occur.
Accept only Authorized Update Requests	A	Update requests are received over an authenticated MQTT channel and are contained in signed manifest file.

(*continued*)

Table 4-10. (*continued*)

System Update Security Principles	Grade	Comments
Use only Authorized Update Sources	A	Manifest file contains authorized source for download of the update mechanism.
Apply Correct Firmware/Software to the System	A	Manifest file contains attributes of the update that are checked on install to ensure invalid updates are not applied. In the event of a failed update or problems during update, Turtle Creek is able to restore the previous version of the software or firmware on the system reducing downtime.

System Firmware and RoT Summary

One of the most difficult problems in IoT systems is updating the base system firmware or recovering from a security vulnerability in a root-of-trust component like a TPM. Oftentimes, these firmware elements are designed to require a trusted administrator to manually watch over an update or install process. IoT devices in remote environments or hard-to-reach places cannot afford to miss such updates, but also cannot be sustained if a skilled administrator must manually install such updates. Services such as Turtle Creek which enable remote update of all software and firmware on a device are vital to both the security posture and ROI of IoT systems.

Application-Level Language Frameworks

The application-level language frameworks are the first topic in the application plane of our generic IoT architectural model from Figure 4-2. Although we are several software layers removed from the hardware of the

platform, hardware-based security still plays a role in providing best-in-class software and services. As we look at different options in this space, we want to focus on how an application developer might be able to leverage hardware-based security features.

Application developers tend to choose an application framework based on the programming language they have chosen, and not vice versa. And particular programming languages tend to have certain frameworks that are popular with a majority of programmers. In this section, we will examine the common security APIs available within some of these frameworks and evaluate the ease of use for developers to utilize hardware security features.

The hardware security features focused here are partly based on the hardware features we have discussed throughout the previous sections of this chapter, as well as security features advantageous to common use cases encountered by IoT developers. These features include

- Access to **Trusted Execution Environments (TEEs)** to leverage highly secure containment features for sensitive data and operations
- Access to **Secure Storage or Protected Keystores** to protect credentials and application secrets
- Access to **message and network security features** to protect communication to other devices
- Access to **cryptographic functions in hardware**, including AES, SHA, and random number generation in order to build other security features not available from available services.

JavaScript and Node.js or Sails

JavaScript is a common language used in IoT and web services today. It is an event-driven interpreted language with a rich set of frameworks. Node.js is one common framework, designed to build network

applications that handle events concurrently. Node.js is extremely flexible, so other frameworks are used to create more structure around Node.js. Sails is an example of such an extension framework.

As far as security goes, Node.js is far removed from most platform security features. However, the crypto API provided in Node.js is a wrapper around the latest OpenSSL library. This means that Node.js developers get access to the hardware implementations of AES-NI and SHA-NI through OpenSSL, as well as the hardware random number generator. Best of all, developers do not have to configure anything or worry about any platform settings – it is all handled inside OpenSSL.

One of the great advantages of Node.js is npm (node package manager). One of the great security problems with Node.js is also npm. The node package manager makes it extremely easy to add packages into your Node.js project. A simple install command issued on the command line and a *require* expression in the code add any package registered in the Node.js npm repository to your application. npm has over half a million packages and over three billion downloads every day.[62] This makes using JavaScript widgets and gadgets built by others very easy (a great benefit!). But what are you really downloading? Are you getting the latest version with the latest bug fixes? Or are you installing the latest version that was corrupted with malware? Often developers set up their Node.js applications and never audit the npm repository again. This poor discipline proliferates security vulnerabilities.

Java and Android

The Java programming language is the language used for Android devices, and because of this popularity has found its way into IoT devices as well. Google provides their Android Things operating system as a base OS and framework built on Java for small IoT devices and provides the same base

[62] https://nemethgergely.com/nodejs-security-overview/

security for the smallest system on a module (SoM) devices as found on larger devices, including secure boot and a secure hardware keystore. Android Things is built from the base Android system, as shown in Figure 4-27, and uses the same kernel, hardware abstraction layer, native libraries, and Java API framework as the standard Android. Android Things is intended for smallest of devices.

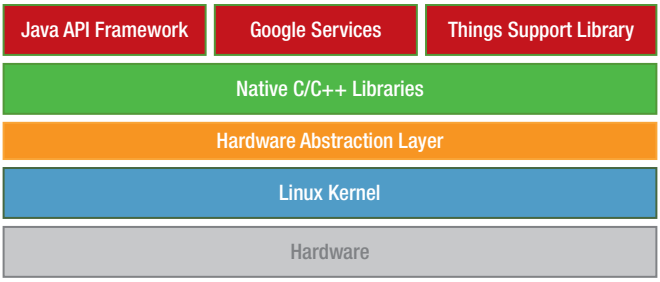

Figure 4-27. *Android Things architecture*

Android itself is popular in many larger IoT devices, including in-vehicle infotainment (IVI) systems in autonomous and smart vehicles. And the security services available through Java and the Android framework are significant.[63]

As we discussed previously, Android supports the Trusty TEE, which can be used to hold sensitive applications for the platform. One of those applications is a hardware-backed secure keystore to protection keys. This prevents keys from being used by unauthorized applications or users and can prevent keys from being exfiltrated off the device. On Intel devices, the Trusty TEE can be used to provide this service, or the keystore can be implemented in the CSME (see Chapter 3). Android also supports a verified boot mechanism where the stages of boot verify each software component is signed with a valid cryptographic key (see Chapter 3 for secure boot details).

[63] https://source.android.com/security/

EdgeX Foundry

EdgeX Foundry is a new Internet of Things framework for industrial edge computing sponsored by the Linux Foundation.[64] EdgeX Foundry is platform agnostic, flexible, and extensible framework providing capabilities for "intelligence at the edge" for data storage, aggregation, analysis, and action all organized into sets of microservices using Docker containers.

Figure 4-28 is the platform architecture for EdgeX Foundry, which includes four service layers and two system services. The service layers are the Export services, Supporting services, Core services, and Device services. The system services are security and device/system management.

The Export services allows data to be communicated to the Cloud and supports several protocols, including REST or message queue protocols (see the section "Message Orchestration"); Google IoT Core is also supported for sending telemetry and receiving configuration and commands. The Device services enables connections to sensors and actuators and supports multiple protocols for this purpose. Some of these protocols are wireless or wired communications protocols which are covered in more detail in Chapter 5; other protocols are message orchestration protocols, like MQTT, which is covered in the section "Message Orchestration." The Supporting services handles edge intelligence and analytics capabilities. The Core services is the linkage between *northbound* communications to the Cloud and *southbound* communications to the sensors and actuators.

[64] https://www.edgexfoundry.org/

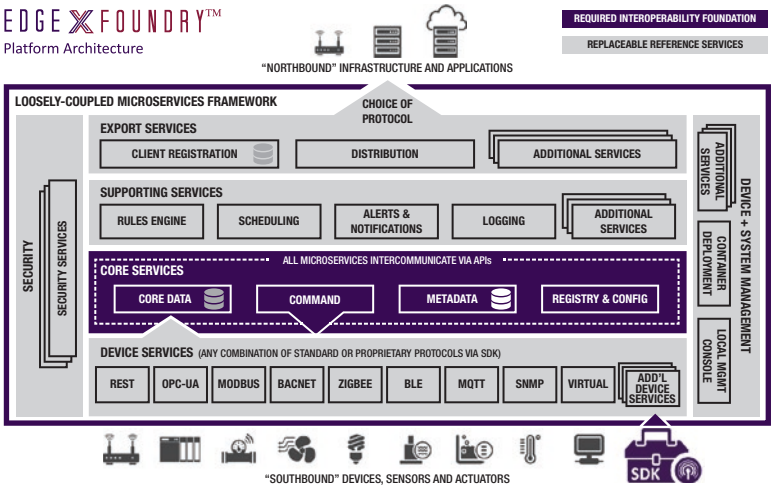

Figure 4-28. *EdgeX Foundry architecture*[65]

The Security system service includes a security store to maintain cryptographic keys and credentials and an access control service to manage REST resources and access keys using either OAuth2 or JWT tokens.

The interesting part of EdgeX is the ability to rewrite any part of the EdgeX Foundry by modifying the Docker container that supplies that service and not having to contend with changing other parts of the system. Security services for key storage can be extended to use a TPM or SGX enclave for enhanced security. Encryption routines in the Distribution container of the Export services can be upgraded to use hardware-based encryption without affecting other elements of the Supporting or Core services. This type of flexible framework makes it easy to utilize the important hardware security features that make an IoT instance more secure.

[65] https://docs.edgexfoundry.org/Ch-Intro.html

Application-Level Framework Summary

The application framework chosen for an IoT device can make a significant difference on the security provided to IoT applications. Frameworks like Node.js have few hardware security features built into the framework, but make it easy to add capabilities. However, access to hardware devices is rather difficult through JavaScript, limiting the options for developers.

Android takes an alternative approach and builds in many sophisticated security features into the operating system and framework itself. However, limitations, such as with the Trusty TEE which cannot dynamically add secured applications, make adding hardware-based security features difficult.

EdgeX Foundry takes a different approach, using containers to separate functionality into microservices. This framework expends effort to create the connections and APIs between components so that services can be shared. In this model, it is much easier to upgrade a service to make use of hardware security features on the platform, but allow platforms that do not have such services to use alternative implementations. Although EdgeX Foundry does not have many hardware security features built into the framework at present, the intention to encourage platform differentiation through service modifications is clearly stated.

Message Orchestration

Message orchestration performs the orderly reception and delivery of data and commands on an IoT platform. As briefly mentioned in "EdgeX Foundry" section, message orchestration protocols enable data delivery and reception off the platform to devices and the Cloud, but can also be used to move data around within an IoT platform. Message orchestration implements the publish-subscribe design pattern, often referred to as pub-sub. In this design pattern, entities with data (publishers) publish their data

to a broker or message bus, and recipients subscribe to certain messages from the broker and are given only the messages for which they register. The beauty of this design pattern is that publishers do not need to know who or how many subscribers are out there, and subscribers do not have to be prepared to receive and parse messages that they are not interested in.

Several message orchestration protocols are common in IoT devices, including Message Queuing Telemetry Transport (MQTT), Constrained Application Protocol (CoAP), eXtensible Messaging and Presence Protocol (XMPP)[66], and OPC Unified Architecture (OPC UA).

Message orchestration needs to deal with several security issues in order to be secure:

- Publishers must have an identity and must be authenticated against that identity so that the source of messages are attributable to an *Authorized Publisher*.
- Subscribers must have an identity and must be authenticated against that identity so that messages are delivered only to *Authorized Subscribers*.
- Authorized Publishers may assign access control lists to messages that restrict which subscribers are allowed to receive their messages.
- Administrators may assign access control lists to message types restricting Publishers from publishing certain message types and/or restricting Subscribers from receiving certain message types.
- Authorized Subscribers may register to receive message types that do not violate an access control list.

[66]XMPP is not covered in this chapter due to space constraints, however details can be found in RFC 6102, https://tools.ietf.org/html/rfc6120

- The message broker will accept a message only from an Authorized Publisher, and only if the message type sent by the Authorized Publisher does not violate an access control list.
- The message broker will deliver a message to an Authorized Subscriber only if that subscriber requested messages of that type, and if that subscriber is not prohibited from receiving that message type by a valid access control list.
- Messages shall be protected from unauthorized disclosure, tampering, unauthorized deletion, reordering, and message delay.

Message Queuing Telemetry Transport

Message Queuing Telemetry Transport (MQTT) is a commonly used message orchestration protocol that enables sending data between entities on a system. The protocol is based on topic names in data packets that define a title for the data. Subscribers subscribe to topics; subscribers may use wildcards within the topic names to which they subscribe. MQTT operates over TCP/IP and supports basic operations, such as CONNECT, PUBLISH, SUBSCRIBE, UNSUBSCRIBE, and several types of acknowledgment packets.

The MQTT protocol published by OASIS[67] supports some basic security services including password-based authentication of publishers and subscribers and recommends the use of TLS for data privacy and integrity.

Several open source implementations of MQTT are in common use including Mosquitto, RabbitMQ, and HiveMQ. Table 4-11 provides a security analysis of Mosquitto MQTT.

[67] http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html

Table 4-11. *Security Analysis of Mosquitto MQTT*

System Update Security Principles	Grade	Comments
Authentication of Publishers and Subscribers	B	MQTT supports usernames and passwords natively. Mutually authenticated TLS is the best option for authentication over the network using public key certificates; using user ids and password is acceptable, but should be protected by TLS if the communication is over a network (broker protection of passwords should be addressed through secure storage). A security vulnerability in Mosquitto up until 1.4.12 allows a user with a specially formatted id to overcome the access permissions set by Mosquitto, allowing them to read or write topics they do not have permissions to access.
Access Controls on Message Topics	B	Mosquitto provides a topic configuration file that allows topics to be restricted by anonymous users, by username, or by a pattern that uses the username or client name; access control is based on "read," "write," or "readwrite" actions. This file must be manually configured, and it is a bit difficult to get correct especially when there are many topics.

(*continued*)

Table 4-11. (*continued*)

System Update Security Principles	Grade	Comments
Message Privacy and Integrity	D	No special protections are provided for messages, and even using TLS does not protect messages while they wait in the queue for delivery, opening the possibility for malware on the broker device to modify messages. Consider adding encryption and message integrity to MQTT messages at the application layer; this provides security end to end and can be used to prevent repudiation attacks as well.
Message Delivery Protections (Deletion, Delay, Reordering)	D	No special protections are afforded to the broker's queue. The broker should not be run as root, but run under a special service user id. In some installations of Mosquitto, the message queue is written to disk and susceptible to tampering. The configuration of your Mosquitto installation should be examined to ensure any files used for queuing are properly protected.

OPC Unified Architecture

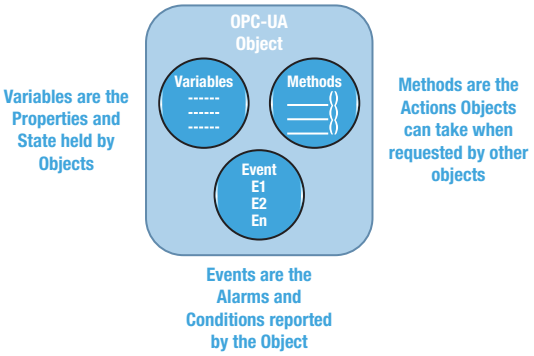

Figure 4-29. *OPC-UA notional object*[68]

OPC-UA[69, 70] is a platform-independent service-oriented architecture targeted to the industrial segment of IoT and is based on the earlier OPC Classic protocols that used the Microsoft Component Object Model (COM) and Distributed Component Object Model (DCOM). OPC-UA is therefore an object-based technology, defining objects as notionally shown in Figure 4-29 and using the TCP/IP protocol for communication between objects, which provides a much richer set of services than MQTT, but it is also much more complex with a 13-part specification of over 1200 pages.

[68] https://opcfoundation.org/wp-content/uploads/2016/05/OPC-UA-Interoperability-For-Industrie4-and-IoT-EN-v5.pdf

[69] OPC officially stands for **O**bject Linking and Embedding (OLE) for **P**rocess **C**ontrol, but since OPC-UA has moved away from strict COM and DCOM protocols, the full expansion of the acronym is no longer widely used.

[70] https://opcfoundation.org/about/opc-technologies/opc-ua/

OPC-UA provides communication between components (objects) on a device and between devices using the publisher-subscriber design pattern described earlier, the observer design pattern where objects notify other objects of events, and using direct method calls between objects (even across devices using a DCOM-like mechanism). OPC-UA includes a discovery service allowing objects and devices to find each other on a network.

OPC-UA defines a comprehensive security model[71] based on security above the transport layer and uses certificate-based identities for applications and users. By default, all communication between devices is encrypted and signed, and the algorithms are negotiated at session establishment between the two parties, just like TLS. All applications are assigned a unique identity certificate, which is used to perform authentication during session establishment to other entities. The other devices/applications/servers a device is allowed to communicate with are defined in a trust list that contains those other applications' identity certificates. Access control and rights can be managed in three different ways: username and passwords, Kerberos tickets, or certificates. Table 4-12 provides a security analysis of OPC-UA.

[71] www.dsinteroperability.com/OPCClassicVSUA.pdf and https://opcfoundation.org/wp-content/uploads/2014/05/OPC-UA_Security_Model_for_Administrators_V1.00.pdf

Table 4-12. *Security Analysis of OPC-UA*

System Update Security Principles	Grade	Comments
Authentication of Publishers and Subscribers	A	OPC-UA includes multiple options for authentication, with public key certificates being included by default. Issuance of these keys can still be an issues that need to be dealt with, but from a security perspective, this is the best solution.
Access Controls on Message Topics	C	Rough access control is provided at the trust list level. OPC-UA applications have to implement their own access control in order to implement anything greater than just this device/application-level trust. Access control functions can take advantage of other information (usernames, certificates, Kerberos tokens), but this requires custom programming.
Message Privacy and Integrity	A	Message encryption and message integrity is built into OPC-UA above the transport layer and can be used to prevent repudiation attacks as well. Session security is provided end to end.
Message Delivery Protections (Deletion, Delay, Reordering)	D	For store-and-forward or pub-sub broker type message delivery, the application is responsible for creating the behavior of the application. Although patterns exist for good design, they are not provided by default for applications and require custom programming.

Constrained Application Protocol

The Constrained Application Protocol (CoAP) is a web transfer protocol specified in IETF RFC 7252[72] specifically designed for devices with limited computation and/or on a network with limited bandwidth. CoAP is a lightweight HTTP protocol and based on the same request-response REST interaction model, using commands GET, PUT, POST, and DELETE. CoAP requires DTLS (Datagram Transport Layer Security, which is TLS over the UDP protocol) for security, and much like HTTP/TLS combination, any additional access control or security on the messages themselves must be added to the applications. Table 4-13 provides a security analysis of CoAP.

Table 4-13. *Security Analysis of CoAP*

System Update Security Principles	Grade	Comments
Authentication of Publishers and Subscribers	A	Mutually authenticated DTLS is the best option for authentication over the network using public key certificates; many other authentication options are possible, but would need to be integrated into the applications (e.g., OAuth, JWT, Kerberos).
Access Controls on Message Topics	D	No special access control is provided above the rough authentication performed by DTLS. Any additional access control must be provided by the application.

(*continued*)

[72] https://tools.ietf.org/html/rfc7252

Table 4-13. (*continued*)

System Update Security Principles	Grade	Comments
Message Privacy and Integrity	D	No special protections are provided for messages beyond the network protections afforded by DTLS.
Message Delivery Protections (Deletion, Delay, Reordering)	C	Messages may be transmitted with reliability (marked as Confirmable), and for those messages, deletion recovery is handled through the acknowledgment mechanism. Every message has a unique 16-bit message id that allows detection of replay.

Message Orchestration Summary

Message orchestration solutions vary widely in their offerings from simple (CoAP) to complex (OPC-UA). The security offerings for the simpler solutions leave much to the application to implement. One of the primary benefits for MQTT is the ease with which network security can be added with TLS, and the rich set of access controls that can be configured without having to add custom code. Other solutions require applications to implement access controls, which can result in harder to diagnose defects, and duplication of the access control code in many places.

Applications

The applications are the components that give IoT devices their behaviors and consume and benefit from the security in the hardware and the software. There is much to explore in the application space, which we leave for Chapter 6, where we explore different vertical IoT applications in great detail.

Summary

Software in IoT is an enormous subject. In writing this chapter, there were many things that had to be left out or shortened in order to meet the page count and retain some semblance of a publishing deadline. If we have omitted your favorite IoT software component or feature, we assure you it is only due to the space limitations. However, we feel that the coverage we have provided of the software elements of an IoT stack is adequate to engage your design enthusiasm and get you thinking about how to expose useful security features in your IoT designs.

The goal of this chapter was to introduce how security could be provided in IoT systems, and we have shown, layer by layer, where platform security features can be exposed and built upon to add strong and effective security services to IoT devices. If the "S" for security is left out of our IoT devices, it is because we have not leveraged the software and capabilities that are available to us to make security a reality.

While it is true that the most constrained devices have less software and less hardware services, this should not be an excuse to remove security entirely. There are too many good options to solve this tough problem. When the constraints get tighter, it should mean that we focus back on the basics and jettison everything we do not need, but retain the most basic security capabilities. These basic security capabilities are the hardware features for the secure minimum viable platform enabled with the basic platform software – secure boot, secure identity, and secure storage. This is not impossible. In Chapter 6, we will show examples of exactly how to do this.