A Reference Architecture for Secure Medical Devices

Steven Harp, Todd Carpenter, and John Hatcliff

Abstract

We propose a reference architecture aimed at supporting the safety and security of medical devices. The ISOSCELES (Intrinsically Secure, Open, and Safe Cyber-Physically Enabled, Life-Critical Essential Services) architecture is justified by a collection of design principles that leverage recent advances in software component isolation based on hypervisor and other separation technologies. The instantiation of the architecture for particular medical devices is supported by a development process based on Architecture Analysis and Design Language. The architecture models support safety and security analysis as part of a broader risk management framework. The models also can be used to derive skeletons of the device software and to configure the platform's separation policies and an extensive set of services. We are developing prototypes of the architecture and example medical device instantiations on low-cost boards that can be used in product solutions. The prototype and supporting development and assurance artifacts are being released under an open-source license.

A reference architecture is a domain-specific design template for the structure of a class of systems as a set of constituent parts with special roles and communications patterns. The reference architecture discussed in this article addresses the class of small bedside medical devices (e.g., infusion pumps, electrocardiographs, ventilators). The architecture also aims to support interoperability interfaces through compatibility with the ASTM F2761 Integrated Clinical Environment (ICE)[1] or the IEEE 11073 service-oriented device connectivity standards.[2] Any such architecture must address a wide range of requirements, including safety, cost, maintainability, and performance. It also must address an unpleasant modern reality: Manufacturers and users cannot assume that medical devices will operate in a benign security environment. Any device that is capable of connecting to a network or physically exposes any sort of data port is potentially at risk. How should we think about and manage risk in this context?

This question has been explored extensively.[3,4] Special publications from the National Institute of Standards and Technology (e.g., 800-39) provide a conceptual risk management framework.[5,6] AAMI TIR57[7] describes how security risk management can be integrated with safety risk management (e.g., as addressed in ISO 14971 and specifically for medical devices in IEC 80001). The UL 2900 series of standards provides cybersecurity requirements for medical devices.[8]

In brief, a threat source initiates a threat event, which may exploit a device vulnerability, causing an adverse impact. The impact might affect the mission of the device, the end user, or the organization that created or operated the device. In this framework, risk is based on the likelihood of a threat event occurring and amplified by the potential loss (adverse impact) should the event occur.

Threat sources have capabilities, intents, and often preferences for particular targets. Automated threats, such as malware, can be indiscriminate in their target selection and do not care whether they are exploiting a personal system or a hospital. Threats targeting particular organizations also are well documented.[9–11] The intent of the threat source is frequently financial, as exemplified by the recent spate of ransomware (e.g., WannaCry).[12] In addition to direct extortion, the medical information that is present in some devices can have indirect value, such as protected health information or personally identifiable information being leveraged to steal funds or illegally acquire drugs.

Steven Harp, *is a distinguished engineer at Adventium Labs in Minneapolis, MN. Email: steven.harp@adventiumlabs.com*

Todd Carpenter, *is chief engineer at Adventium Labs in Minneapolis, MN. Email: todd.carpenter@adventiumlabs.com*

John Hatcliff, *PhD, is a distinguished professor at Kansas State University in Manhattan, KS. Email: hatcliff@ksu.edu*

A special type of impact needs to be considered for medical devices: The lives and health of patients may depend on them. Security and safety are intertwined in such systems, and failure to manage security risks may pose safety risks.[7] Beyond being an end target of an attack, a medical device may represent a stepping stone for a larger campaign. Exploitation of a vulnerable device may permit access to other clinical or financial systems by allowing the attacker to leverage trust information or to extract credentials that allow the device to connect to these other systems. An example of a successful attack of this sort is documented in the MEDJACK report on blood-gas-analyzers attacked in 2015–16.[9]

A noteworthy long-term risk is when malware authors or researchers automate the attacks. This could result in multiple, nearly simultaneous malfunctions, similar to what occurred with the WannaCry attack in May 2017. With automation, botnet controllers could use the devices to attack other devices on healthcare facility networks. Decades ago, the attackers were people who actively attacked your computer. Unfortunately, automated botnets—millions of already-compromised machines trying to grow their reach—are now the prevalent attackers.

Security controls in the device can further reduce an attacker's ability to exploit certain outstanding vulnerabilities and minimize the impact of successful exploitation of others.

Recent Department of Homeland Security advisories documented hard-coded passwords and credentials in medical devices.[13,14] Vulnerabilities such as these make it easier for malware (e.g., botnets, ransomware) to gain a foothold on the device.

By itself, a medical device architecture cannot address all aspects of risk. Security analyst Sergey Lozhkin noted that although vulnerable devices were revealed in penetration testing, "the problem is not only one of weak protection of medical equipment, it has a much wider scope. The whole IT infrastructure of modern hospitals is not properly organized and protected, and the problem persists worldwide."[10]

Until the day when software can be automatically and completely validated, we must assume that some vulnerabilities remain even after formal design and extensive testing. However, a suitable reference architecture and tools to help instantiate it can reduce the attack surface. Security controls in the device can further reduce an attacker's ability to exploit certain outstanding vulnerabilities and minimize the impact of successful exploitation of others.

The remainder of this article presents principles driving the design and instantiation of Intrinsically Secure, Open, and Safe Cyber-Physically Enabled, Life-Critical Essential Services (ISOSCELES)—a safe and secure architecture for medical devices.

Architecture

The ISOSCELES reference architecture is specified with tiered requirements:

1. **Platform core requirements** for hardware, system software, and services supporting the medical application.
2. **Platform design requirements** derived from the platform core requirements and refined to a specific design.
3. **Device design requirements** specific to a particular medical device being developed (e.g., an infusion pump).

The first two platform tiers drive the reference architecture. These 134 requirements are intended to be suitable for a broad family of potential medical devices.

Figure 1 illustrates various components of a hypothetical medical device design based on this reference architecture. The exact medical applications and services needed may vary according to the device. A hardware layer (bottom of figure) contains processors and peripheral devices that support security mechanisms. A low-level separation layer of software uses these mechanisms to isolate all other software components from each other, except for specifically allowed interactions. The ISOSCELES platform services satisfy typical needs of the top-layer medical applications that provide medical diagnostics and therapies. The design of a specific device might incorporate only the required elements needed for that class of device.

Architectural Principles

The ISOSCELES reference architecture attempts to promote basic principles that provide security and safety. These principles and their motivations are described below.

Use Strong Time and Space Separation

Many modern systems provide a basic security perimeter but lack internal security barriers, making it easy for attackers to access arbitrary information within the device, and even to control the device, after they have breached the perimeter. Strong barriers between software components help contain faults and attacks from propagating to safety-critical components.

For example, if an infusion pump requires a network connection (e.g., to maintain current drug libraries), the network stack and code associated with retrieving the drug library should be wholly separate from the software that interprets the contents of the drug library. In turn, that function also should be entirely separate from the function that controls the rate at which the drug is pumped. In safety architectures, each unit of separation is called a partition.[15] Separation kernels and some real-time operating systems (OSs) provide strong separation. Historically, application software on medical devices has been built either on minimal OSs that fail to provide strong separation or without the benefit of any OS at all.

Spatial separation refers to barriers between the memory regions and addressable peripherals assigned to different components. A component operating in one partition should not be able to directly reference the memory or input/output (IO) devices assigned exclusively to another component. Failure to do so would allow a defective or compromised component to affect the missions of other components.

Temporal separation refers to the inability of one component to affect or measure the time intervals during which another component accesses a resource. For example, a defective or compromised component should not be able to dominate a central processing unit (CPU) and thereby keep another component from performing its mission in a timely manner. Programs running in one partition should be oblivious to the operation and resource usage of programs in other partitions, except as explicitly intended.

ISOSCELES addresses these requirements by specifying a low-level software kernel dedicated to establishing and maintaining

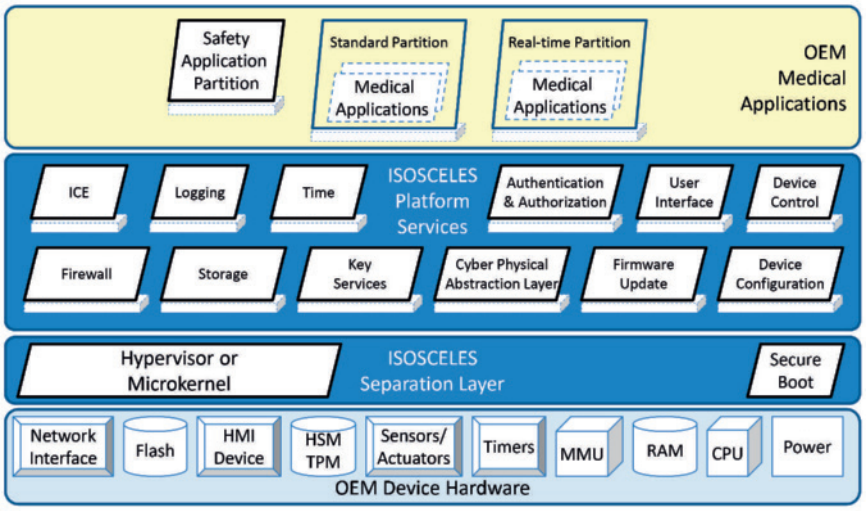

Figure 1. The layered ISOSCELES (Intrinsically Secure, Open, and Safe Cyber-Physically Enabled, Life-Critical Essential Services) architecture. Abbreviations used: CPU, central processing unit; HMI, human-machine interface; HSM, hardware security module; ICE, integrated clinical environment; MMU, memory management unit; OEM, original equipment manufacturer; RAM, random-access memory; TPM, trusted platform module.

spatial and temporal separation. It is the one program on the device that has direct ability to control the protection facilities provided by the hardware. It is intentionally as small and simple as possible to increase its trustworthiness.

Key hardware features that support this role are hardware memory management units (MMUs) and input/output MMUs (IOMMUs). While standard on desktop systems, these features historically have not been present on inexpensive or low-power microprocessors. For example, ARM microcontrollers prior to the ARMv3 family lacked integral MMUs, and Intel Atom processors prior to 2016 lacked IOMMUs. These classes of processors are present in many fielded medical devices, providing opportunities for attacks that begin in vulnerable processes to escape and tamper with the resources of other processes. However these features have recently appeared in affordable embedded CPUs in both ARM and x86 processor architectures.

An MMU is programmable hardware that controls the physical memory that a given process may access while providing the process with the appearance of a well-ordered "virtual address space." It also can mark a section of memory as read-only. In some cases, it may be able to prevent the processor from executing instructions in certain areas of memory. If a process

attempts to address memory outside of its partition or in ways not permitted by the programmed policy, then access is blocked and the CPU will trap to a handler to deal with the fault.

Recent Meltdown and Spectre attacks have demonstrated several flaws in super-scalar processors that violate this protection; the processors specified by ISOSCELES, such as the ARM A53, do not currently exhibit these issues. The IOMMU performs a similar function for peripheral devices. Failure to isolate the IO space opens a door for defective or compromised software components to breach their partitions and interfere with other components and external devices. For example, a process designed to check for network updates might be compromised by a maliciously crafted packet, and injected code could modify the GPIO (general-purpose input/output) pins to turn on a pump or change the power state of the device.

Prototypes of ISOSCELES have used both the seL4 microkernel and the Xen hypervisor for separation.[16,17] A microkernel is like an OS kernel but is devoid of the usual panoply of device drivers, network stacks, and file system drivers. It specializes in strongly separating memory and processor time into partitions, along with minimal support for interpartition communication (IPC). The seL4 microkernel is distinguished by a formal proof of its correctness on certain processor architectures.[18] Hypervisors offer the additional ability to provide each partition an emulated environment, giving the component the illusion that it is running on its own hardware. This permits partitions to run general-purpose OSs (e.g., Linux, Berkeley Software Distribution, Microsoft Windows). So-called type 1 hypervisors (e.g., Xen) support both low-level separation and the emulation capabilities.

Minimize Privilege

Going hand-in-hand with separation, the principle of minimal privilege, or least privilege, is a cornerstone of security. In the context of the reference architecture, it means that any given component should be granted only the privileges required to perform its mission. Privileges here include the ability to access resources such as memory, arbitrary processor instructions, and peripheral devices. Unfortunately, embedded systems based on commodity OSs (e.g., Linux, Windows) are all too often implemented with "root" access, which means that as soon as the perimeter is breached, the attacker can easily access anything.

For our reference architecture, the hardware layer assists in this goal in ways beyond the separation mechanisms described above. Modern microprocessors offer multiple operation modes, including a supervisory mode with higher privilege and a user mode with lower privilege. The supervisory mode is allowed to execute all of the processor instruction set and is permitted to program the MMU and IOMMU. The user mode cannot change the memory management and is blocked from using privileged instructions, such as controlling interrupt handling or processor faults.

The principle of minimal privilege, or least privilege, is a cornerstone of security. In the context of the reference architecture, it means that any given component should be granted only the privileges required to perform its mission.

ISOSCELES stipulates that only the separation layer operates in the most privileged mode. All other software components run in an unprivileged mode and rely on the separation layer to delegate to them only those special privileges needed for their missions. Although something like this "kernel-versus-userland" distinction exists in general purpose OSs, systems based on microkernels extend this demotion to components such as device drivers, file system drivers, and protocol stacks. These often are relatively complex components, and their failure should not jeopardize other parts of the overall system. For example, the compromise of a network device driver in a monolithic kernel would threaten *every* aspect of the system's operation. (The driver would run with full privileges in the kernel memory space.) In a microkernel, the device

driver is just another user-level component, and compromise can only affect other components that interact with it.

Minimize Complexity

The trustworthiness of software tends to correlate inversely with complexity; the more complex code is, the less trustworthy it is. Many factors influence this, including the number of attack surfaces, the difficulty of correctly developing code, and the effort required to verify and validate desired properties. The reference architecture encourages the construction of components that are as small and simple as possible.

Many application components can perform their primary functions without all of the accouterments of typical applications running in a general purpose OS. Example "extra baggage" includes general purpose C and other standard software libraries, arbitrary software languages, and full access to all system calls. The Linux 2.6 kernel has 338 system calls.[19] The Standard C library adds 500 kB (and often much more) binary code.[20] The presence of this code in the address space of a component provides an attack surface that facilitates remote code execution using techniques such as return-oriented programming.[21]

The reference architecture allows designers to place less trustworthy components in a subordinate role to more trustworthy peers.

In contrast, ISOSCELES specifies a minimal component runtime environment that includes only essential IPC primitives. Components can leverage the architecture's service layer using IPC and avoid including service layer code in their own memory partitions. Using this methodology, components (in the microkernel realization of ISOSCELES) can be shrunk from hundreds to tens of kilobytes and only expose a very narrow IPC attack surface. The Architecture Analysis and Design Language (AADL) modeling tools (see LEVERAGE MODELS OF CORRECTNESS section) help specify and construct appropriately minimal designs.

Manage Trust Relations

Even in the simplest systems, a device's software components must communicate with each other. Modern medical devices also must communicate with external systems. This interaction carries inherent risk, as one party must trust another to provide valid and timely messages. The safety and liveness of critical processes depends on this trust being warranted.

The reference architecture addresses this challenge in several ways. First, it restricts internal communications by policy to predefined channels. A defective or compromised component is incapable of opening new channels to other components. These channels are specified and checked in the design's AADL model, then translated to the enforcement mechanism in the microkernel or hypervisor.

External communications are further regulated by an integral firewall that is independent of the internal communicating component. A default-deny firewall policy ensures that the device only communicates with a "whitelist" of approved parties. COTS OSs often ship with open policies to ease integration and development overheads; these policies must be explicitly tightened down.

The framework encourages communications patterns that limit the degree of trust required. In many cases, the relationships between components can be ordered by an evaluation of their relative trustworthiness or by their relative criticality. For example, some components are measurably more complex than others or contain third-party code libraries that cannot be fully evaluated. Components that communicate directly with external networks might be less trustworthy, as they could be compromised by external attack. Safety-related components should be treated as more critical than other operational components and should not directly depend on those external-facing components.[15] The reference architecture allows designers to place less trustworthy components in a subordinate role to more trustworthy peers.

Designs may select communications mechanisms on a per-channel basis to provide the desired separation, including:

- Synchronous interpartition remote procedure calls.
- Message queues.
- Shared memory marked as read-only to some components.
- Asynchronous event signaling.

Leverage Common Services

The service layer shown in Figure 1 contains several components that support common medical device needs. A medical device based on the reference architecture would include implementations of these services; device designers simply select those required, avoiding the need to fully develop them in house.

- A **logging service** can direct device log messages to a local persistent store and/or over the network to a remote logging host.
- The **time service** provides current synchronized time, as well as precision delayed signals, to trigger timed actions.
- A **storage service** manages the persistent storage devices on the device, which may include multiple encrypted logical partitions.
- The **cyber-physical abstraction layer** component isolates the medical applications from directly interfacing with medical sensors and actuators. This adds flexibility by allowing devices to map logical sensor or actuator roles to different physical devices, thereby reducing the verification and validation impacts if a sensor is updated. It also can support networked devices. For example, a future infusion pump might leverage a blood oxygen sensor that resides on a different medical device in the same ICE.[22]
- A **firewall service** controls access to data networks connected to the device. Internal components that need network access do so through this internal firewall, which exposes a single external address.
- An **update service** manages secure device firmware upgrades in the field, which may occur over the network or through an attached storage device.
- **Device control** and **device configuration services** support the operating life cycle and the configuration of the device and its components.
- The **authentication** and **authorization services** control the access that users are granted to particular device capabilities. Users may be granted roles (e.g., clinician, administrator) that, through policy, dictate the operations they may perform on the device.
- A **user interface service** provides display and human input devices to allow device users to monitor and control it.
- The **key service** controls cryptographic operations, as discussed below.

Software engineering has made progress in machine-generated proofs from formal models of systems. When the system implementation is generated from these models, in part or in whole, developers can reap additional advantages of reduced engineering effort and a correct design.

Leverage Cryptography for Confidentiality and Integrity

Medical devices require information protection to preserve confidentiality and integrity. This is vital to several tasks for the medical device:

- Verifying software before executing it
- Transferring sensitive data to or from persistent storage
- Validating software updates for the device
- Communicating with external services (e.g., pharmacies)
- Communicating with peer devices in an ICE

Cryptography plays a central role in satisfying these requirements. Any component in a design may include a cryptographic library to satisfy its information protection needs. This includes standard services (e.g., storage, update, ICE) and device-specific components (e.g., a drug library component that must synchronize with an external pharmacy). The reference architecture calls for a cryptographic key management service (CKMS), which provides a mechanism to control cryptographic materials such as keys, shared secrets, and certificates. This enforces a policy that controls which operations may be performed by which components on any given cryptographic datum. The CKMS may in turn use a hardware security module, if available (e.g. a trusted platform module). Such devices are designed to be tamper resistant, and keys stored within them are protected even from attackers who attempt to disassemble a device.

Leverage Models of Correctness

While system testing is the common approach for verifying the correctness of an implementation, analytical proofs of correctness are even more valuable. Software engineering has made progress in machine-generated proofs from formal models of systems. When the system implementation is generated from these models, in part or in whole, developers can reap additional advantages of reduced engineering effort and a correct design.

The ISOSCELES reference architecture is represented in an AADL model.[23,24] This language has well-defined semantics for hardware and software components of a system, with attributes that support reasoning about system-level characteristics of a design. Figure 2 shows example communications flows for part of a design. These flow annotations specify the allowed information flows between particular components operating in different partitions (e.g. from diagnostics to interoperability services), then out to network services (e.g., electronic health records). Flows can be checked at design time against a security policy that might forbid information flowing directly from unregulated components, such as the interoperability services flowing to the class III therapy services.

In addition, tools convert these modeled flows into the runtime communications tables, which provides the tie between the model that was analyzed and the actual implementation.

The models also support temporal analysis. The model may specify tolerances on latencies between events or the amount of jitter tolerated in a cyclic schedule. Tools can examine models and evaluate whether the process schedule will meet all of the requirements, accounting for factors such as interpartition transition overhead. The tools also can generate the runtime schedules.

AADL also supports modeling features that permit reasoning about faults and fault management strategies.[25] Formalizing the assumptions about failure modes and effects analysis means that the design's hazards can be assessed against fault behavior and propagation in both qualitative and quantitative ways.

Conclusion

The ISOSCELES medical device reference architecture currently is under development. When completed, it will provide FDA-required documentation and an exemplar device to demonstrate the use of the architecture. One prototype is based on the Xen hypervisor[16] and an embedded computer using an ARM CPU (dual-core Cortex-A7)

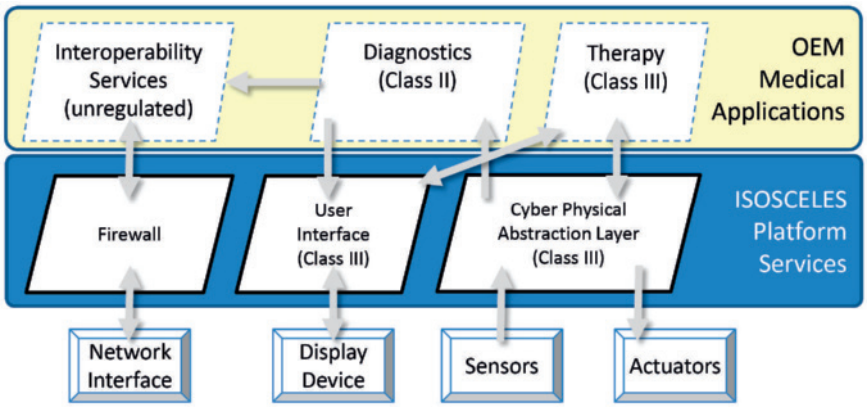

Figure 2. AADL models of interaction between components in an ISOSCELES (Intrinsically Secure, Open, and Safe Cyber-Physically Enabled, Life-Critical Essential Services) design. Abbreviation used: OEM, original equipment manufacturer.

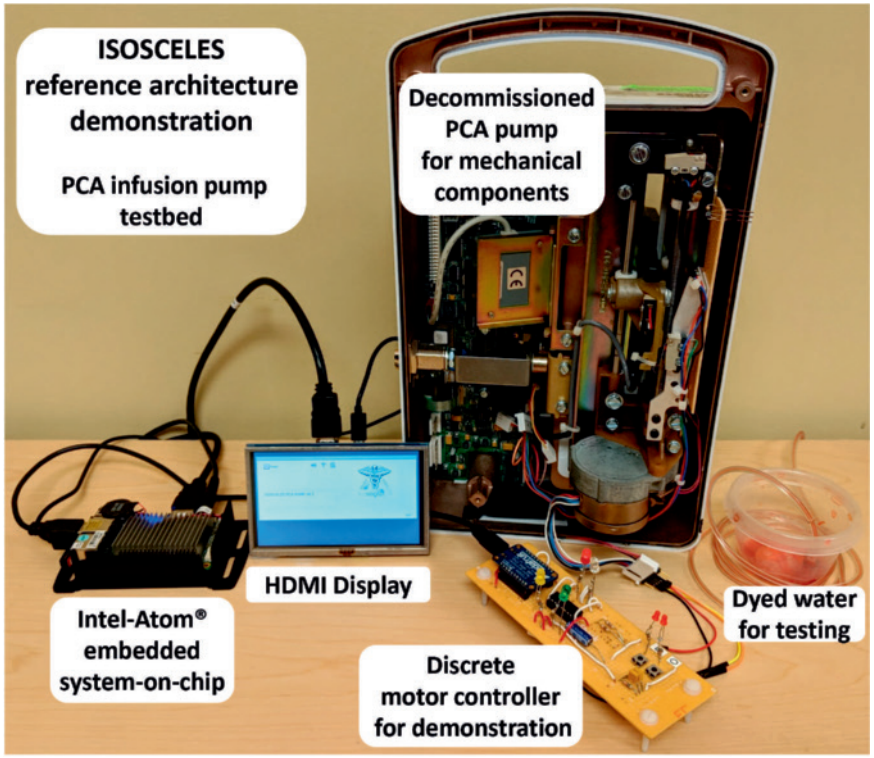

Figure 3. A laboratory prototype of a patient-controlled analgesia (PCA) pump demonstrating ISOSCELES (Intrinsically Secure, Open, and Safe Cyber-Physically Enabled, Life-Critical Essential Services) partitioning on the seL4 separation kernel. The ISOSCELES-based PCA pump software runs on the small Intel Atom embedded development board. That board is connected to modified surplus pump hardware to demonstrate functionality. Abbreviation used: HDMI, high-definition multimedia interface.

system-on-chip (SoC). It uses a commodity compute module and interfaces with the electromechanical components of a decommissioned patient-controlled analgesia pump. Another prototype runs on ARM, Intel Atom, and AMD G-series–based SoCs running the seL4 or NOVA microkernels with Genode application framework (Figure 3).[26]

ISOSCELES requirements, designs, example implementations, and associated development tools will be made available as open source products. Medical device firms may use these artifacts as a starting point to develop their proprietary designs. Original equipment manufacturers (OEMs) will need to select a hardware and separation approach based on their unique product needs. Next, if not already available, OEMs should port the chosen separation approach to their hardware and particular interface devices. They can use the ISOSCELES platform services on top of that separation layer and refine features, such as the specific key management and authentication approaches, to satisfy their market needs.

Original equipment manufacturers can use the ISOSCELES platform services on top of that separation layer and refine features, such as the specific key management and authentication approaches, to satisfy their market needs.

ISOSCELES was developed specifically to target ICEs with interoperable devices, with strong user and machine-to-machine authentication and configurable networking and key management. The current form of ISOSCELES targets external devices with modern processors, communications, and power supplies and is not intended for extremely power-constrained environments. Although the architecture principles remain important in those environments, the power constraints driven by the ever-shrinking device volumes will challenge current technologies due to increased memory and computation requirements.

Acknowledgments

This material is based on research sponsored by the Department of Homeland Security (DHS) Science and Technology Directorate; Homeland Security Advanced Research Projects Agency (HSARPA), Cyber Security Division (DHS S&T/HSARPA/CDS) Broad Area Announcement number HSHQDC-14-R-B0005; the Government of Israel; and the National Cyber Bureau in the Government of Israel via contract number D16PC00057. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the DHS, U.S. government, Government of Israel, or National Cyber Bureau in the Government of Israel.

Several artifacts are being developed in collaboration with Food and Drug Administration (FDA) engineers through the National Science Foundation/FDA Scholar-in-Residence program, which is providing partial support for this work.

References

1. ASTM F2761. *Medical Devices and Medical Systems—Essential Safety Requirements for Equipment Comprising the Patient-Centric Integrated Clinical Environment (ICE)—Part 1: General Requirements and Conceptual Model*. West Conshohocken, PA: ASTM International; 2009.

2. IEEE 11073-10207-2017. *IEEE Health informatics—Point-of-care medical device communication Part 10207: Domain Information and Service Model for Service-Oriented Point-of-Care Medical Device Communication*. Piscataway, NJ: IEEE; 2017.

3. Almohri H, Cheng L, Yao D, Alemzadeh H. On Threat Modeling and Mitigation of Medical Cyber-Physical Systems. In: *The Second IEEE/ACM International Conference on Connected Health: Applications, Systems and Engineering Technologies (CHASE), July 17–19, 2017, Philadelphia*. 114–9.

4. Martins G, Bhatia S, Koutsoukos X, et al. Towards a Systematic Threat Modeling Approach for Cyber-physical Systems. In: *2015 Resilience Week (RWS)*. Piscataway, NJ: IEEE; 2015:1–6.

5. National Institute of Standards and Technology. *Managing Information Security Risk: Organization, Mission, and Information System View.* Special Publication 800-39. Gaithersburg, MD: National Institute of Standards and Technology; 2011.

6. National Institute of Standards and Technology. *Guide for Conducting Risk Assessments*. Special Publication 800-30. Gaithersburg, MD: National Institute of Standards and Technology; 2012.

7. AAMI TIR57. *Principles for medical device information security—risk management*. Arlington, VA: Association for the Advancement of Medical Instrumentation; 2016.

8. UL 2900. *Software Cybersecurity for Network-Connectable Products*. Washington, DC: UL; 2017.

9. TrapX. Attackers Target Blood Gas Analyzers. Available at: https://trapx.com/wp-content/uploads/2017/08/Case_Study_TrapX_Healthcare_MEDJACK_BGA.pdf. Accessed Aug. 2, 2018.

10. Lozhkin S. Hospitals are under attack in 2016. Available at: https://securelist.com/blog/research/74249/hospitals-are-under-attack-in-2016. Accessed Aug. 2, 2018.

11. Brook C. Sergey Lozhkin on How He Hacked His Hospital. Available at: https://threatpost.com/sergey-lozhkin-on-how-he-hacked-his-hospital/116314. Accessed Aug. 2, 2018.

12. Fox-Brewster T. Medical Devices Hit by Ransomware for the First Time in US Hospitals. Available at: www.forbes.com/sites/thomasbrewster/2017/05/17/wannacry-ransomware-hit-real-medical-devices. Accessed Aug. 2, 2018.

13. Department of Homeland Security. Advisory (ICSMA-17-250-02A): Smiths Medical Medfusion 4000 Wireless Syringe Infusion Pump Vulnerabilities (Update A). Available at: https://ics-cert.us-cert.gov/advisories/ICSMA-17-250-02A. Accessed Aug. 17, 2018.

14. Department of Homeland Security. Advisory (ICSMA-18-037-02): GE Medical Devices Vulnerability. https://ics-cert.us-cert.gov/advisories/ICSMA-18-037-02. Accessed Aug. 17, 2018.

15. Larson BR, Jones P, Zhang Y, Hatcliff J. Principles and Benefits of Explicitly Designed Medical Device Safety Architecture. *Biomed Instrum Technol*. 2017;51(5):380–9.

16. Xen Project. Homepage. Available at: www.xenproject.org. Accessed Aug. 2, 2018.

17. Carpenter T, Hatcliff J, Vasserman EY. A reference separation architecture for mixed-criticality medical and iot devices. In: *Proceedings of the 1st ACM Workshop on the Internet of Safe Things, Delft, Netherlands, Nov. 6–8, 2017*. New York: ACM; 2017.

18. Heiser G, Elphinstone K. L4 Microkernels: The Lessons from 20 Years of Research and Deployment. *ACM Transactions on Computer Systems*. 2016:34(1):1–30.

19. Linux Syscall Reference. Available at: http://syscalls.kernelgrok.com. Accessed Aug. 2, 2018.

20. Eta Labs. Comparison of C/POSIX standard library implementations for Linux. Available at: www.etalabs.net/compare_libcs.html. Accessed Aug. 2, 2018.

21. Shacham H. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In: *Proceedings of the 14th ACM Conference on Computer and Communications Security, Oct. 29 through Nov. 2, 2007, Alexandria, VA*. New York: ACM; 2007:552–61.

22. King A, Arney D, Lee I, et al. Prototyping closed loop physiologic control with the medical device coordination framework. In: *Proceedings of the 32nd International Conference on Software Engineering*, May 1–8, 2010, Cape Town, South Africa. New York: ACM; 2010:1–11.

23. AS5506. *Architecture Analysis & Design Language (AADL)*. Warrendale, PA: SAE International; 2004.

24. Software Engineering Institute. Open Source AADL Tool Environment (OSATE). Available at: http://osate.org. Accessed Aug. 17, 2018.

25. SAE International. SAE Architecture Analysis and Design Language (AADL) Annex Volume 1: Annex A: ARINC653 Annex, Annex C: Code Generation Annex, Annex E: Error Model Annex AS5506/1A. Available at: www.sae.org/standards/content/as5506/1a. Accessed Aug 17, 2018.

26. Genode Labs. Documentation of the Genode OS Framework. Available at: https://genode.org/documentation/index. Accessed Aug. 2, 2018.