10 Properties of Secure Medical Systems

Protecting Can't-Fail Embedded Systems from Tampering, Reverse Engineering, and Other Cyberattacks

www.windriver.com

WNDRVR

It's Not a Fair Fight

When attacking an intelligent edge medical system, it takes only one vulnerability to put patient health at risk.

Security requirements for medical software present a growing challenge as devices move from stand-alone systems or private networks into cloud operations. Intelligent systems offer rewards, but they also introduce risks. Among these risks are the increasing efforts of outside actors to exploit medical devices as entry points for ransomware and other attacks. Worse yet, an attacker may try to use a compromised device to go further, pivoting from one exploited subsystem to another and jeopardizing patient health while causing further damage to the device company's network, mission, and reputation.

This white paper covers the most important security design principles that, if adhered to, give you a fighting chance against any attacker who seeks to gain unauthorized access, reverse engineer, steal sensitive information, or otherwise tamper with your embedded medical system.

The beauty of these 10 principles is that they can be layered together into a cohesive set of countermeasures that achieve a multiplicative effect, making medical device exploitation significantly more difficult and costly for the attacker.

DESIGN PRINCIPLE	BRIEF EXPLANATION	IMPLEMENTATION EXAMPLES
1. Data-at-Rest Protection	Software, data, and configuration files are protected when stored in nonvolatile memory, typically through means of encryption. Keys stored in security hardware.	Full-disk encryption File encryption TPM / HSM
2. Authenticated and/or Secure Boot	Software (including firmware and configuration data) will be authenticated and/or decrypted before use.	TXT, BootGuard UEFI SecureBoot Application whitelisting
3. Hardware Resource Partitioning	Hardware computing resources (processor cores, cache, memory, devices, networks) will be segregated to provide independent functions to the maximum degree possible.	Memory management unit / Paging Multi-core / Multi-socket Cache allocation technology Resource director technology Total memory encryption (TME / MKTME)
4. Software Containerization & Isolation	Software applications will be well defined, self-contained, containerized, and isolated.	Process address spaces / Virtual memory Docker / Containers Virtualization / Separation kernel / Hypervisor
5. Attack Surface Reduction	Minimize dependencies / Trusted computing base Minimize codebase Limited and well-defined interfaces	Code removal Network and application firewalls Software Guard Extensions (SGX)
6. Least Privilege & Mandatory Access Control	Users and applications will be provided only the minimal set of privileges/access necessary to function using non-bypassable mandatory access control (MAC).	SELinux / AppArmor / SMACK SECCOMP / chroot XSM / FLASK (Hypervisor)
7. Implicit Distrust & Secure Communications	Communications with external sources will be expressly denied until the remote source can be authenticated. Data in transit will be encrypted.	SSL / TLS Identity and certificate management
8. Data Input Validation	Any and all data received from untrusted sources (network, file, IPC) should be validated before being passed into software applications.	Data format filters Cross-domain guards
9. Secure Software Development, Build Options, & OS Configuration	Software applications and OS kernel shall be compiled and configured with all available security options enabled and enforced.	Type and memory-safe languages (e.g., Rust) Build parameters (FORTIFY_SOURCE, NX) Kernel configuration (e.g., signed drivers, ASLR)
10. Integrity Monitoring & Auditing	The system will perform ongoing integrity monitoring and audit logging of security-relevant events.	Continous memory hash verification Audited

Data-at-Rest Protection

Your applications, configurations, and data aren't safe if they're not protected at rest. Period.

You can protect your medical applications and data at rest in one of two ways:

1. Prevent the attacker from gaining access to this information in the first place.
2. Make it impossible for the information to be understood.

The connected medical devices market reached $28.24 billion in 2020, and it is expected to grow at a CAGR of 18.92% by 2026.[1] And that means that the medical industry is a prime target for cyberattack. The SANS Institute reported that, as of 2020, 94% of health care organizations reported being cyberattack victims.[2] This includes attacks on medical devices, medical infrastructure, and patient data. **Unless you can guarantee that your system is physically inaccessible, preventing attackers from ever gaining access is an exceedingly tall order.**

This leaves us with method two: Make it impossible to understand the information.

Though there are many ways to obfuscate or otherwise garble your data and applications to make them more difficult to understand, most aren't worth the effort and are often trivially bypassed or subverted.

When an attacker has access to your software or data, it's only a matter of time before they figure out how your system works. However, if your applications and data are encrypted with proven cryptographic algorithms and the decryption key is not accessible to the attacker, it's game over. At the very least, you have forced the adversary to use a more intrusive method of attack to achieve their objective.

Properly implemented, encryption at rest is designed to protect the confidentiality of your sensitive medical data from physical access.

Encryption can also protect the integrity of the software components on a device. For example, encrypted storage volumes can prevent attackers from injecting malware, modifying configurations, or disabling security features on a device.

Using certified and/or industry standard crypto-algorithms such as AES, RSA, ECC, or SHA will help protect your data at rest and prevent an attacker from gaining access — as long as you keep the secret crypto-keys out of reach when the system is powered off (hint: tamper-resistant hardware), during boot, and throughout runtime operation.

1 *"Connected Medical Device Market: Growth, Trends, COVID-19 Impact, and Forecasts (2021–2026)," Mordor Intelligence, 2021*

2 *Patricia A.H. Williams and Andrew J. Woodward, "Cybersecurity Vulnerabilities in Medical Devices: A Complex Environment and Multifaceted Problem," PubMed Central, U.S. NIH, 2015*

Secure Boot

Your system isn't safe if you can't prove that, while booting up, your code wasn't manipulated, modified, or replaced with an alternate, malicious version.

Yes, handing off control from the hardware to the software is a complicated dance that any medical embedded system conducts to get up and running. But that doesn't mean it's indecipherable.

Hundreds — maybe thousands — of vulnerabilities exist in system boot sequences that, if left unprotected, can and will be exploited by a would-be attacker to gain access to your software and compromise applications and data. For example, boot attacks are the most common method used to "root" popular mobile devices and enable unauthorized applications and system modifications. **A well-engineered secure boot sequence helps protect against system compromise during startup.**

Many secure boot technologies exist, including:

1. UEFI Secure Boot:[3] Free for many platforms, taking static root-of-trust measurements and providing validation of kernel command-line arguments
2. Grub Secure Boot:[4] With options for validating kernel, initramfs, and command line, and also integrating with UEFI secure boot
3. Intel TXT/tboot:[5] Can provide authentication and encryption during a measured launch and also prevent certain advanced hardware attacks
4. uboot: Leverages platform-specific bits (i.e., fuses) to perform a verified boot using encryption and authentication
5. Commercial products, such as Wind River Titanium Security Suite's Secure Boot.[6]

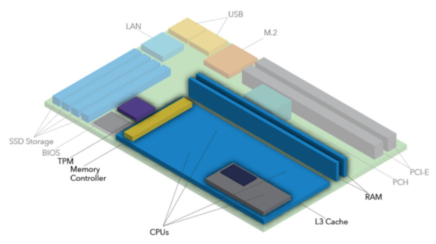
Many other forms of secure boot for SoCs leverage platform-specific bits and perform verified or measured launches of OS code using encryption and authentication.

Whichever secure boot technology you are using, be sure it's as strong as these examples to ensure that your hardware kicks off only the intended and authentic software and not an attacker's malicious code.

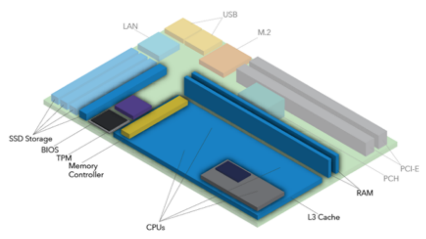
3 *www.linuxjournal.com/content/take-control-your-pc-uefi-secure-boot*
4 *ruderich.org/simon/notes/secure-boot-with-grub-and-signed-linux-and-initrd*
5 *software.intel.com/content/www/us/en/develop/articles/intel-trusted-execution-technology-intel-txt-enabling-guide.html*
6 *www.starlab.io/titanium-product*

SECURE BOOT SEQUENCE

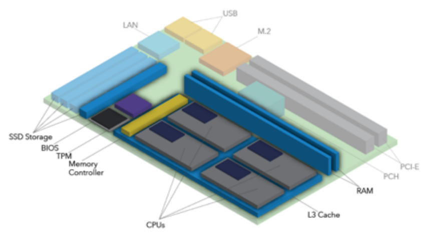
1. Encrypted at rest



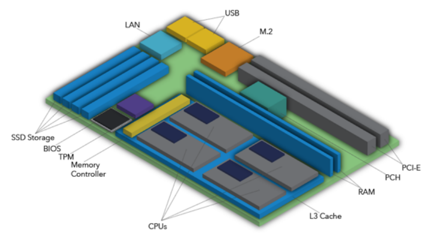
2. Measured boot
(Unlock key material)



3. Decrypt OS



4. Secure at runtime



Hardware Resource Partitioning

If your software stack is allowed unconstrained access to every hardware component on your medical system, then an attacker can potentially leverage that same access to catastrophic effect.

It's like trying to host a safe event during a pandemic — just one infected person can jeopardize everyone's health.

Constraining software workloads to particular hardware components (CPU cores, cache, memory, devices, etc.) leads to a cleaner, more straightforward system configuration. It also happens to provide very important security properties.

Traditional embedded OSes have limited protections between processes and application/system dependencies, and since the OS kernel is similarly not separate from the individual device driver services, the attack surface is large and enables a single exploit to compromise the integrity of the entire system.

An architecture in which components are isolated via strong, hardware-enforced boundaries enables defense in depth, especially if interfaces between separated components are tightly controlled. Any vulnerabilities exploited in one application remain constrained to that application. Thus they cannot spill over into other (isolated) components to disrupt the entire system. Furthermore, strict partitioning and isolation can prevent co-execution vulnerabilities, which is an enabling factor for exploit families such as Spectre and Meltdown.[7]

Separating components via hardware partitioning improves the overall resiliency of the system, as one component can no longer directly or indirectly affect another component.

Additionally, partitioning the system into discrete components reduces the collective attack surface and increases overall system security by reducing and/or minimizing privilege escalation, preventing resource starvation and denial of service, mitigating side channel and/or timing attacks, and laying the groundwork for future fault-tolerant application approaches.

> Good security practice requires reasoning through potential attacks at every level of the system, understanding and questioning design assumptions, and implementing a defense-in-depth security posture.

7 *meltdownattack.com*

Software Containerization and Isolation

Just as one rotten apple can spoil the whole barrel, one insecure piece of code, if not properly isolated, can compromise the entire system.

This is possible because a vulnerability exploited in one piece of code enables the attacker to run arbitrary commands with the same set of privileges as that application — possibly writing to memory or devices where other software components reside. Thus, an initial exploit can quickly gain the attacker unrestricted access to the entire system or, even worse, long-term persistence.

Containerization of code helps mitigate such attacks, preventing an exploit in one component from affecting another component.

To mitigate the effects of software exploitation attacks, the defender should containerize, sandbox, and isolate different system functions into separate enclaves. This approach starts at the system architecture stage, ensuring that applications and subcomponents are well defined and self-contained with clearly understood and enforced boundaries. Next, data flows should be analyzed to ensure that inter-component interactions are known and can be controlled.

Containerization can be accomplished at multiple levels within the software stack, including separate name spaces (i.e., Docker[8]), virtual machines, separation kernels, and/or hardware-enforced memory spaces. When implemented correctly, even exploited software remains constrained to just its process address space, VM, or container, thereby limiting the reach of an attacker and preventing the unintended escalation of access across system components.

Software applications will be well defined, self-contained, isolated, and containerized with these elements:

- Process address spaces
- Virtual memory
- Docker
- Containers
- Virtualization
- Separation kernel
- Hypervisor

8 www.docker.com

Attack Surface Reduction

The more code you deploy, the more opportunity an attacker has to find an entry point into the system.

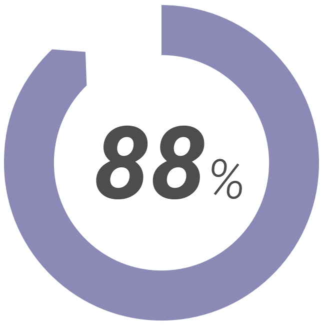
Recall that an attacker has to exploit only one vulnerability to be successful, while the defender must protect against all vulnerabilities. Thus **every additional line of deployed code potentially introduces software bugs that an attacker can exploit for their nefarious reasons.**

It's a losing battle.

The best approach, then, is to reduce the attack surface by removing code and interfaces that are not absolutely required.

For example, instead of mindlessly deploying a monolithic Linux distribution onto an intelligent edge device, cut out the drivers, features, and code you don't actually need. A zero-day attack on a graphics card driver can't be successful on a system that doesn't include that driver to begin with.

Similarly, even a known vulnerable service cannot be exploited if the service has been disabled or the interface is removed.



of surveyed medtech leaders in the U.S. said they did not think their companies were prepared to prevent cyberattacks.[9]

Check After Changes

"Whenever software is changed, a validation analysis should be conducted not just for validation of the individual change but also to determine the extent and impact of that change on the entire software system. Based on this analysis, the software developer should then conduct an appropriate level of software regression testing to show that unchanged but vulnerable portions of the system have not been adversely affected."

— U.S. Food and Drug Administration[10]

9 *Maria Fontanazza, "IoMT, Connected Devices Introduce More Cyber Threats into Med Tech and Healthcare Organizations," MedTech Intelligence, July 12, 2021*

10 *General Principles of Software Validation; Final Guidance for Industry and FDA Staff, U.S. Department of Health and Human Services, Food and Drug Administration, January 2002*

Least Privilege and Mandatory Access Control

The principle of least privilege says that your systems' software components should only be granted the minimal privileges necessary to do their job, and nothing more.

Applications (and users/operators) should only have access to the minimum set of interfaces and services necessary for their job.

Too often software developers and system engineers take the shortcut — inadvertently (or even explicitly) granting excessive privileges to applications, with an assumption of trusted operator and/or application behavior. That assumption will be quickly invalidated by the attacker.

Instead, intelligent edge systems should be built using mandatory access controls (MAC). Unlike discretionary access controls (which can be modified at will by users and administrators), systems built on mandatory access control quantify access grants and restriction policies during system design — controls that are always enforced in the fielded device. There is no user or administrative way to bypass or disable the security controls within the fielded device.

Even if an attacker is successful in compromising a subcomponent of the system or gains root-level access, they will not have a way to modify or disable the security settings of the device. When combined with least privilege, mandatory access controls greatly constrain the attacker's freedom to maneuver and block the ability to modify, disable, or disrupt system services.

Properly implemented mandatory access control policies do not interfere with normal system operation, and they still allow the system to work as designed and intended. The policies can also be updated in a secure and controlled manner by the system implementer. However, mandatory access control intentionally prevents systems from operating in unintended ways, which is a highly desirable property in embedded computing.

If you need to deploy that graphics driver for functionality, then go for it.

Just be careful not to allow unauthorized components to access it if not absolutely necessary. This principle is known as least privilege and mandatory access control.

Implicit Distrust and Secure Communications

Communication received on your system from external sources should be expressly denied until the remote source has been authenticated.

In other words, a secure system doesn't let just any other system talk to it; it forces external systems to prove themselves. **The starting point for secure communication should be default-deny.**

More so, just as it is better to share your credit card information only to those you trust, in a closed room where no one else is around to hear, your system should enforce secure communication even after the other party has been authenticated.

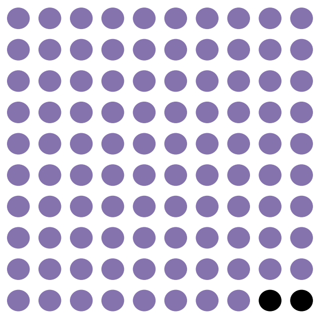
That typically means data in transit will be encrypted.

Luckily, both of these properties can be implemented with widely used, easily accessible, proven encryption communications protocols, such as SSL and TLS[11] with identity and certificate management. Of course, any time crypto is involved, it raises the question of how you plan to protect those TLS keys and certificates (hint: tamper-resistant hardware).

By implementing mutual authentication and encryption, you'll have more certainty that you are only communicating with trusted entities (not the attacker), and that nobody else can eavesdrop on what is being communicated.

Once you are able to securely transmit information from one system to another, you can focus on validating that information to prevent malicious data input attacks.

98%



of all IoT device traffic — including medical device traffic — is left unencrypted.[12]

11 *www.websecurity.digicert.com/security-topics/what-is-ssl-tls-https*

12 *IoMT, Connected Devices Introduce More Cyber Threats into MedTech and Healthcare Organizations," MedTech Intelligence, July 12, 2021*

Data Input Validation

A secure software architecture does not make assumptions about the acceptability of a given input and will validate the format and content of that input before allowing it to be processed by the rest of the system.

Many developers fail to imagine how a malicious attacker may intentionally craft malformed inputs that are designed to cause the software to malfunction.

Data entering a system via any interface — whether a sophisticated MRI machine or a patient's wearable fitness-tracking device — can become a vector for attack, exploiting software vulnerabilities to gain unauthorized access or corrupting system or application memory to create a denial of service.

In other words, inputs from a variety of external sources, such as sensors, networks, etc., should be subject to data input validation before use.

Additional vetting of user input (where *user* means an actual human user, a peripheral user, or a machine operator) is required. But all devices should inspect the conformance of messages to a prescribed data standard as they are passed from device to device.

Furthermore, because any component of the system could become compromised at any point, and thus any message may be maliciously crafted and sent by an attacker, a secure software architecture operates on the principle of mutual distrust.

Components within the system must prove their trustworthiness through a continuous (or at least frequent) authentication step. Furthermore, authentication must expire periodically and be reaffirmed.

Device-to-device authentication is often enforced during network formation and at random times thereafter. Message signing and verification are typically included in all messages between authenticated devices.

Validating data before use helps to ensure that external inputs cannot unintentionally interrupt or maliciously exploit system functionality and lead to compromise of the system.

Secure Development, Build Options, and System Configuration

Adding some security features is as simple as configuring your build options correctly.

You've probably heard of a buffer overflow attack.[13] It's a common attack aimed at overwriting memory regions with malicious code. **Many compilers, by simply configuring them correctly, can now identify whether such an attack is possible by analyzing your code long before it's deployed.**

Of course, other build options can be set to warn you (or error out) on many types of potential security[14] issues and also provide security enhancements, such as:

1. Detection of signed/unsigned conversions
2. Warning of uses of format functions that represent possible security problems
3. 64-bit address space layout randomization
4. Compilation of code with unintended return addresses
5. Mitigated variants of Spectre
6. Defeat of stack-smashing attacks
7. Protection for the stack and heap against code execution
8. Enablement of code instrumentation of control-flow transfers

These are just a few of the available security measures you can implement. Even better, if you have the ability to specify the programming language for your system, you can eliminate entire classes of software vulnerability. For example, the popular Rust programming language can eliminate memory-safety and type-safety programming concerns.

Secure software build options and system configuration to validated standards are low-effort, bare minimum requirements that go a long way toward preventing attackers from running circles around your other cyberdefenses.

Start with Requirements

"Software requirement specifications should clearly identify the potential hazards that can result from a software failure in the system, as well as any safety requirements to be implemented in software. The consequences of software failure should be evaluated, along with means of mitigating such failures (e.g., hardware mitigation, defense programming, etc.). From this analysis, it should be possible to identify the most appropriate measures necessary to prevent harm."

— U.S. Food and Drug Administration[15]

13 *www.imperva.com/learn/application-security/buffer-overflow*
14 *security.stackexchange.com/questions/24444/what-is-the-most-hardened-set-of-options-for-gcc-compiling-c-c*
15 *General Principles of Software Validation; Final Guidance for Industry and FDA Staff, U.S. Department of Health and Human Services, Food and Drug Administration, January 2002*

Integrity Monitoring and Auditing

You can't take action against an attacker if you don't know when your system is being attacked.

Integrity monitoring and auditing are important techniques for knowing when a medical device is being attacked and/or whether it has been compromised. These warnings give you the potential to stop an attacker before it is too late, or at least learn how they exploited your system and what they were able to accomplish.

Typical techniques include network and OS-level anomaly detection, system log monitoring, and scanning for known malware. They allow the system operator to recognize when some portion of the system may be compromised and take action against the attacker, revoke trust accordingly, or both.

Furthermore, auditing is a requirement of many compliance regulations, as the techniques help organizations detect unauthorized modifications to important files, data, or other aspects of your system. HIPAA, NIST, FISMA, NERC, and PCI all require or recommend integrity monitoring and auditing for critical applications and data on distributed systems.

Properly implemented, auditing and monitoring allow you to know when you've been attacked, help quantify the damage, and enable you to recover more quickly — preventing lost time, revenue, and damage to your reputation.

Wind River platforms serve as a trusted foundation so you can innovate securely and protect your device against current and future threats.

2B+

Wind River technology is in more than 2 billion devices throughout the world.

Wind River is a global leader of software for the intelligent edge. Its technology has been powering the safest, most secure devices since 1981 and is in billions of products. Wind River is accelerating the digital transformation of mission-critical edge systems that demand the highest levels of security, safety, and reliability.

WNDRVR

Appendix

No One Property to Rule Them All

Unfortunately, there is no one security property to rule them all.

There's no single tip or trick or technology or technique that can immediately and permanently prevent an attacker from compromising your system. It takes a combination of many techniques to do that.

Start with these 10 properties in order to build security into the design, implementation, and operation of your intelligent edge medical system:

1. Encrypt sensitive applications and data.
2. Ensure that your firmware, OS, and config settings are authentic before use.
3. Separate system functions into distinct enclaves.
4. Sandbox exploits and prevent attackers from expanding their reach.
5. Reduce the amount of code and interfaces that an attacker will have the opportunity to exploit.
6. Ensure that software components can only do what they were intended to do, and nothing more.
7. Secure data in transit and expressly deny external communication unless authenticated.
8. Do not implicitly trust data received from untrusted sources.
9. Ensure that software applications are compiled and configured with all available security options enabled and enforced.
10. Detect and take action that protects the system against relevant security events.

If all of these properties are in place, implemented properly on your system, you'll have a fighting chance against any attacker who seeks to exploit your system, steal your IP, or impact your brand reputation.

10 Properties of Secure Intelligent Edge Systems

1 Data-at-rest protection

2 Authenticated and/or secure boot

3 Hardware resource partitioning

4 Containerization and isolation

5 Attack surface reduction

6 Least privilege and mandatory access control

7 Implicit distrust and secure communications

8 Data input validation

9 Secure development, build options, and OS config

10 Integrity monitoring and auditing

Contact us if you are interested in learning how these 10 properties can be applied to your use case and what technologies Star Lab (a Wind River company) can bring to quickly and easily meet your security requirements and protect your system against the full spectrum of reverse engineering and cyberattacks.