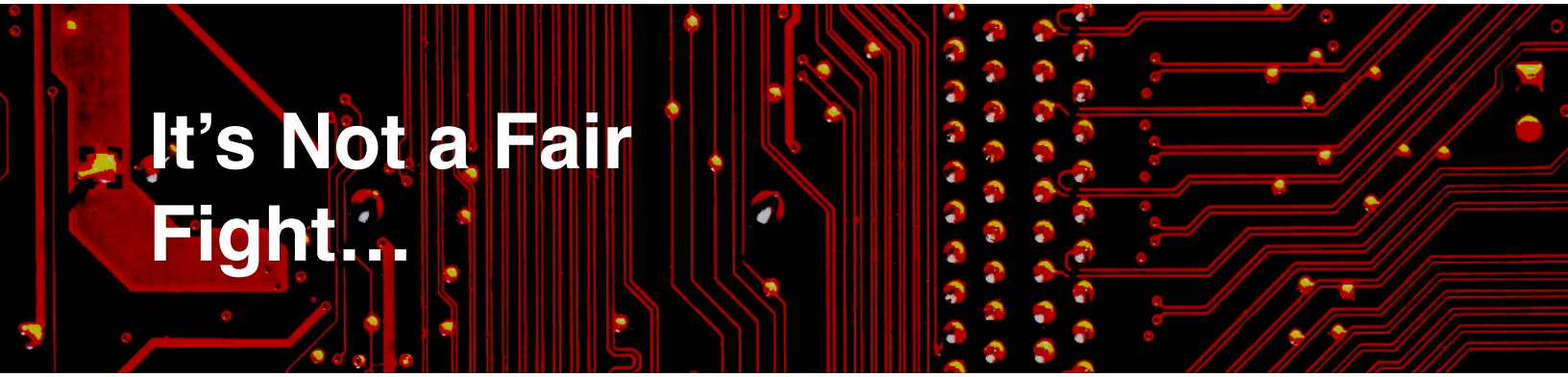
10 Properties of Secure Embedded Systems

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

www.windriver.com



It's Not a Fair Fight...

When attacking an embedded system, it takes only one vulnerability to lead to an exploit.

When tasked with securing an embedded system, you (the defender) must be prepared to protect against every possible vulnerability. Overlook a single opening and the attacker may find it, take control, steal your secrets, and create an exploit for others to use anytime, anywhere.

Worse yet, that same attacker may use an initial compromised device to pivot from one exploited subsystem to another, causing further damage to your network, mission, and reputation.

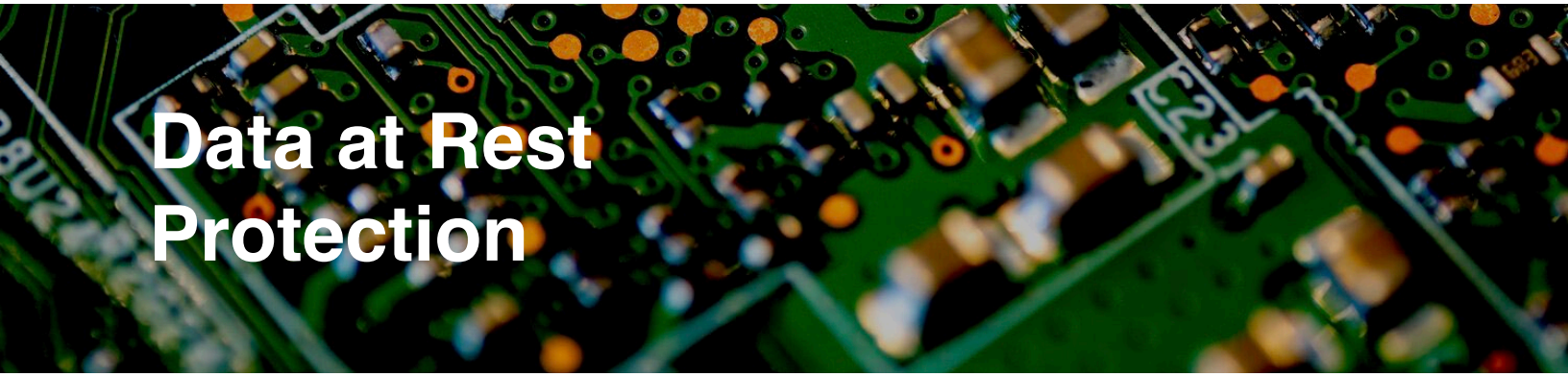
This white paper covers the most important security design principles that, if adhered to, give you a fighting chance against any attacker who seeks to gain unauthorized access, reverse engineer, steal sensitive information, or otherwise tamper with your embedded system.

"When tasked with securing an embedded system, you (the defender) must be prepared to protect against every possible vulnerability."

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 device exploitation significantly difficult and costly for the attacker.

	Design Principle	Brief Explanation	Implementation Examples (Intel)
1	**Data-at-Rest Protection**	Software, data, and configuration files are protected when stored in non-volatile 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 controls (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 (ex: Rust) Build parameters (FORTIFY_SOURCE, NX) Kernel configuration (ex: signed drivers, ASLR)
10	**Integrity Monitoring & Auditing**	The system will perform ongoing integrity monitoring and audit logging of security relevant events.	Continuous memory hash verification Auditd

WIND



Data at Rest Protection

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

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

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

Embedded devices are now distributed by the millions to consumers around the world. **Therefore, unless you can guarantee that your system remains physically inaccessible behind guns, gates, and (trusted) guards, preventing the attacker from ever gaining access to your data and intellectual property is an exceedingly tall order.**

This leaves us with method two: Making it impossible to understand the information at all.

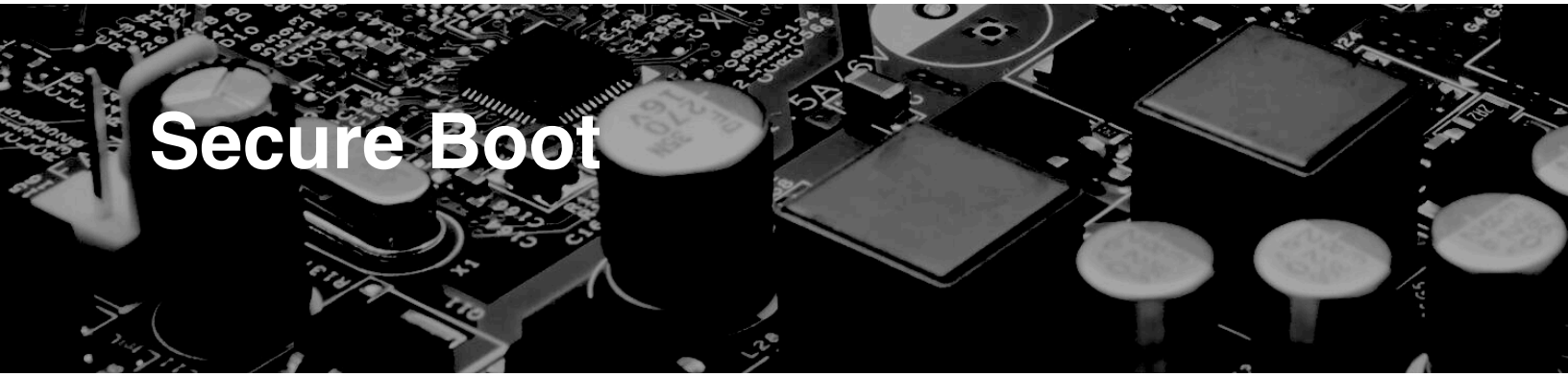
Though there are many ways to obfuscate or otherwise garble your data and applications to make them more difficult to understand, most aren't worth the effort and are often trivially bypassed or subverted.

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 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 – so 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.



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 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.**

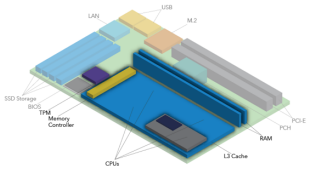
Many secure boot technologies exist including:

1. UEFI Secure Boot, free for many platforms, which takes static root of trust measurements and provides validation of kernel command line arguments.
2. Grub Secure Boot, which has options for validating kernel, initramfs, and command line, and also integrates with UEFI Secure Boot.
3. Intel TXT/ tboot, which can provide authentication and encryption during a measured launch, and also prevents certain advanced hardware attacks.
4. uboot, which leverages platform-specific bits (i.e., fuses) to perform a verified boot using encryption and authentication.
5. Commercial products, such as Star Lab’s own Titanium Secure Boot solution.

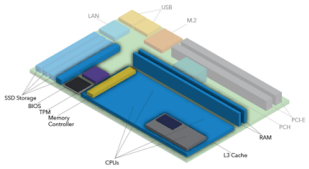
Many other forms of secure boot for SoCs leverage platform-specific bits and perform verified or measured launches of operating system code using encryption and authentication.

Whichever secure boot technology you are using, be sure to implement a strong one like these to ensure your hardware kicks off only the intended and authentic software instead of an attacker’s malicious code.

SECURE BOOT SEQUENCE

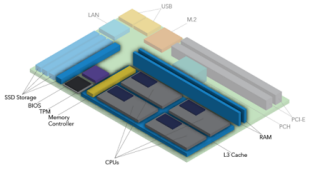


1. Encrypted at Rest

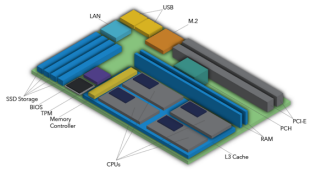


2. Measured Boot

(unlock key material)

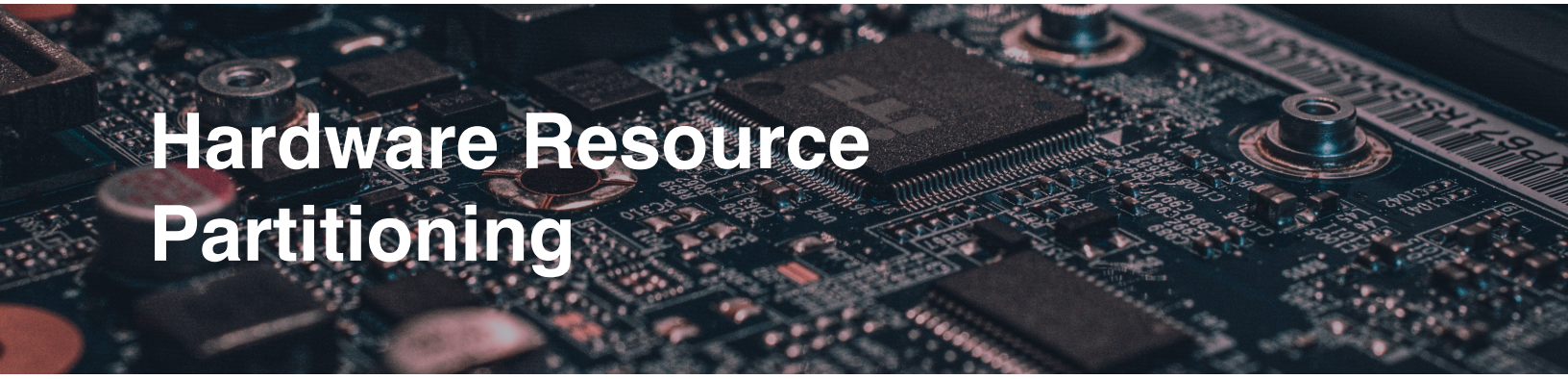


3. Decrypt OS



4. Secure at Runtime

WIND



Hardware Resource Partitioning

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

It is like building a boat without bulkheads – a single leak can compromise the whole ship.

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 operating systems have limited protections between processes and application/system dependencies, and since the operating system 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, and thus 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 like Spectre and Meltdown.

Separating components via hardware partitioning, therefore 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, & 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.

WIND

Software Containerization & Isolation

Just like one rotten apple can spoil the whole bunch, one insecure piece of code can, if not properly isolated, 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.

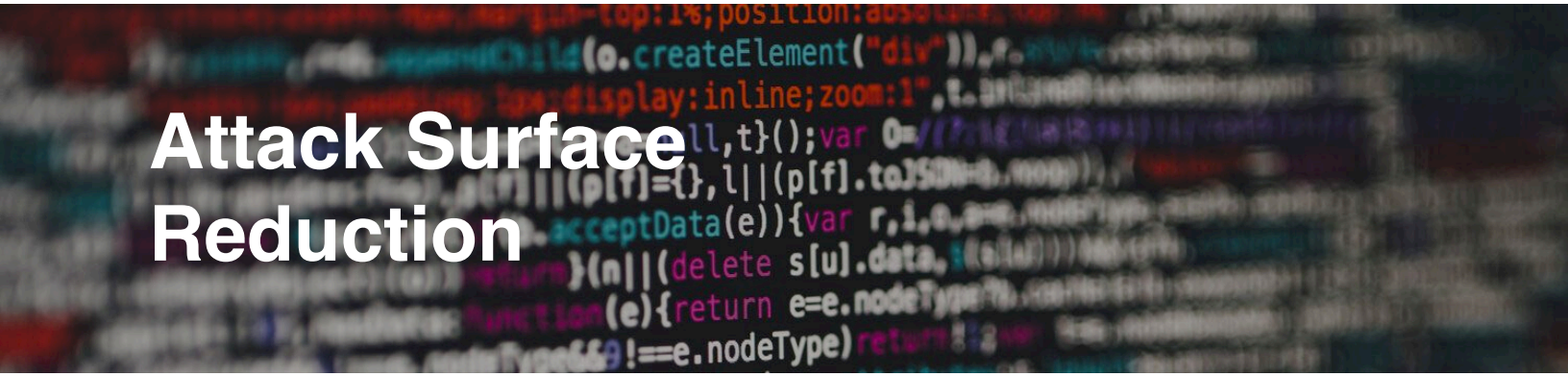
Containerization of code helps to mitigate such attacks, preventing an exploit in one component from affecting another.

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 namespaces (i.e. Docker), 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, containerized, and isolated:

- Process Address Spaces
- Virtual Memory
- Docker
- Containers
- Virtualization
- Separation Kernel
- Hypervisor



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 only has to exploit one vulnerability to be successful, while the defender must protect against all vulnerabilities. As such, **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 embedded 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.

The more a defender can do to prevent an exploit from occurring in the first place, the better.

One of the best ways to do that is by reducing the system's attack surface.

97% of risk management professionals stated that they believed that unsecured IoT devices could be open to a "catastrophic" security breach.

WIND

Least Privilege & 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, embedded 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 upon Mandatory Access Control quantify access grants and restriction policies during system design – controls that are always enforced in the fielded device. As such, there is no user or administrative way to bypass/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 security settings of the device. When combined with least privilege, Mandatory Access Controls greatly constrain the attacker's freedom of maneuver, and blocks their ability to modify, disable, or disrupt system services.

Properly implemented MAC 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, a principle known as Least Privilege & Mandatory Access Control.

Implicit Distrust & 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 just let 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 to those you trust in a closed room where no one else is around to hear it, 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 using widely used, easily accessible, and proven encryption communications protocols like SSL and TLS with Identity and Certificate management. Of course, anytime 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 (and 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 the information sent to prevent malicious data input attacks.

"There are updates that happen every single day about potential security exposures. We have a team here at TGCS that focuses on that; we partner with Wind River to make sure that the known risks are identified and that we respond quickly for our retailers."

—Gregg Margosian,
COO, Toshiba

98% of all IoT device traffic—including medical device traffic—is left unencrypted.

WIND

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.

Data entering a system via any interface can become a vector for attack – exploiting software vulnerabilities to gain unauthorized access or corrupting system/application memory to create a denial of service.

In other words, inputs from a variety of external sources such as sensors, radios, 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 adversary, 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.

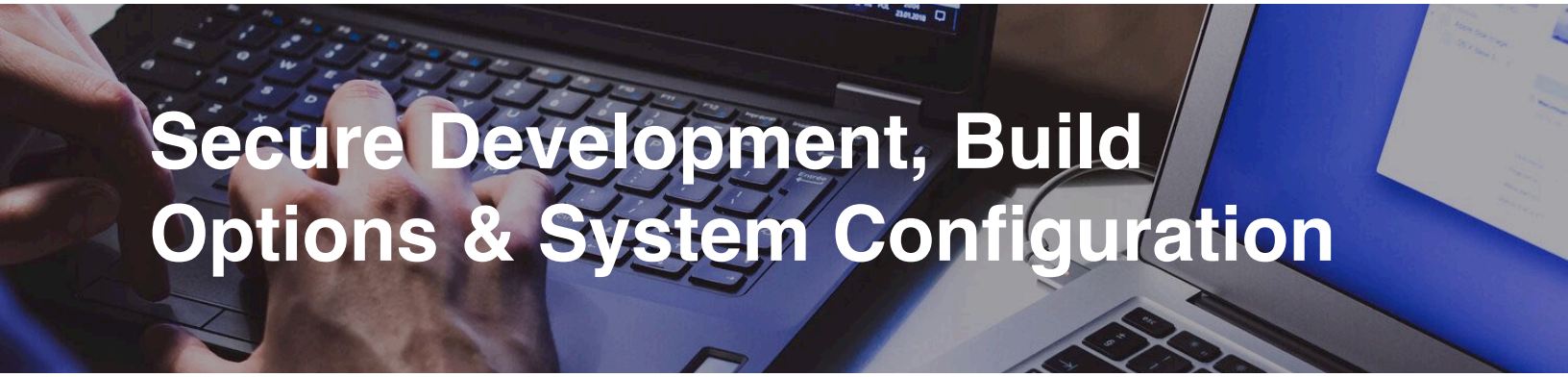
Device-to-device authentication is often enforced during network formation and at random times thereafter. Message signing and verification are typically included in all messages between authenticated devices.

Validating data before use helps to ensure that external inputs cannot unintentionally interrupt or maliciously exploit system functionality leading to compromise 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."

The majority of malicious data input manipulation attacks target known vulnerabilities in application software and common libraries, which leads us to Secure Software Development practices.

WIND



Secure Development, Build Options & System Configuration

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

You've probably heard of a buffer overflow attack. 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 issues and provide security enhancements such as:

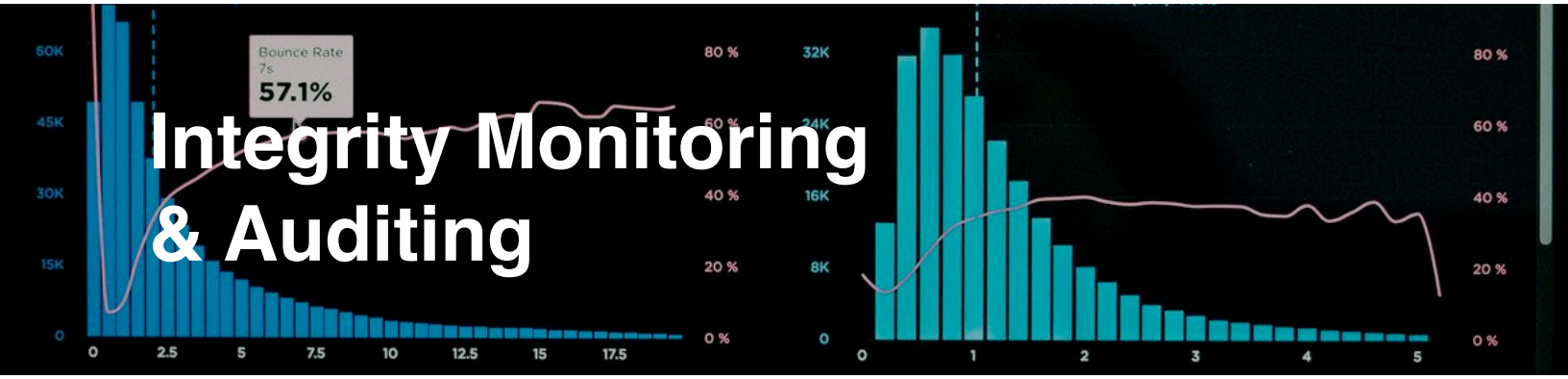
1. Detection of signed/unsigned conversions
2. Warning for uses of format functions that represent possible security problems
3. Take advantage of 64-bit address space layout randomization
4. Compiling code with unintended return addresses
5. Mitigating variants of Spectre
6. Defeating stack smashing attacks
7. Protecting the stack and heap against code execution
8. Enabling code instrumentation of control-flow transfers
9. And many more…

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 driving circles around your other cyber defenses.

By following defensive coding practices, using secure build options, and configuring the end system for maximum security (depending upon your security requirements), you can significantly decrease the number of possible attacks that can compromise one or more parts of your system.

WIND



Integrity Monitoring & 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 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 after the fact.

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.

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

2B+

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

Wind River is a global leader in delivering software for the intelligent edge. Its comprehensive portfolio is supported by world-class professional services and support and a broad partner ecosystem. Wind River is accelerating digital transformation of critical infrastructure systems that demand the highest levels of safety, security, and reliability.

© 2020 Wind River Systems, Inc. The Wind River logo is a trademark of Wind River Systems, Inc., and Wind River and VxWorks are registered trademarks of Wind River Systems, Inc. Rev. 07/2020

WIND

No One Property to Rule Them All

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

There's no one 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 ten properties in order to build security into the design, implementation, and operation of your embedded system:

1. Encrypt sensitive applications and data.
2. Ensure 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 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 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 seek to exploit your system, steal your IP, or impact your brand reputation.

10 Properties of Secure Embedded 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 & Mandatory Access Control
7. Implicit Distrust & 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 ten properties can be applied to your use case and what technologies Star Lab can bring to quickly and easily meet your security requirements and protect your system against the full spectrum of reverse engineering and cyber-attacks.

WIND