HighIntegritySystems

WITTENSTEIN

RTOS Security: SAFERTOS® and Its Enhanced Security Module

Issue 1.2 - March 07, 2024

WITTENSTEIN high integrity systems
Americas: +1 408 625 4712
ROTW: +44 1275 395 600
Email: sales@highintegritysystems.com
Web: www.highintegritysystems.com

Contents

List of Figures

List of Notation

ACP	Access Control Policy
BSP	Board Support Package
ESM	Enhanced Security Module
MCU	Microcontroller Unit
MPU	Memory Protection Unit
MMU	Memory Management Unit
OACP	Object Access Control Policy
RoT	Root of Trust
RTOS	Real Time Operating System
SOUP	Software of Unknown Provenance
TCB	Task Control Block

CHAPTER 1 Introduction

Embedded device security relies on a series of nested objectives. As these objectives build upon each other, it's vital that all aspects of the system objectives are implemented in a highly robust way. The security of the system is only as strong as its weakest link.

Embedded device security typically involves:

- Protection of sensitive and confidential data, for example health care records or payment details;
- Preventing an external actor taking control of the system, or aspects of the system;
- Limiting the effects of a Denial of Service (DoS) attack;
- The protection of the product IP.

1.1 Root of Trust

In most instances the initial security objective of an embedded device is to confirm that the device that has booted is the correct device, and that the software of the device has not been compromised. This is normally referred to as the Root of Trust.

The root of trust is typically implemented using a dedicated security module entirely in hardware, or a combination of hardware and software integrated in a security coprocessor. This creates a secure mechanism for developers to lock down their code. Each time the system boots, it validates the digital signature of the booting code. Only proven, authentic software can execute. Attacks that modify the system software are detected and blocked.

The actual code booted depends on the application. It could be that just a boot loader is initially authenticated and installed. This boot loader will systematically authenticate and load other elements of the system. In other, typically smaller applications, it could be the complete application that is loaded.

The digital signature uses private/public key cryptography for demonstrating the authenticity of digital data. It provides authentication, non-repudiation and integrity. Authentication is when a valid digital signature gives a recipient reason to believe that the data was created by a known source. Non-repudiation is when the sender cannot deny having sent the message. Finally integrity is where it can be seen that the data was not altered in transit or during storage.

In addition to the authentication of the booting code, digital signatures are also commonly used to authenticate connecting devices, configuration data and remote data messages.

Encryption is used to protect data at rest or in transit. Data at rest could include highly confidential health care records, or the code image. Encryption is used so that attackers can't discover the plain text of a message, or stored data.

1.2 The RTOS and the Root of Trust

The Real Time Operating System (RTOS) forms part of the Root of Trust, on which the application can function, this is initialised and enabled towards the end of the booting process. The RTOS provides a safe and secure platform for the application to function. The type of security required from the RTOS is highly dependent on the software architecture, the types of threats the system will be subjected to, and the attack surface the system is trying to defend.

This white paper explores the type of security that can be provided by a RTOS, and uses SAFE**RTOS**® and its Enhanced Security Model (ESM) as an example of good practice.

CHAPTER 2 The Attack Surface

The type of security required by an embedded device is highly dependent on the software architecture, the perceived threat, and the attack surface being defended.

The attack surface of the software/system is the sum of the different points where a bad actor can try to enter, or extract, data to or from an environment. Keeping the attack surface as small as possible is a basic security measure.

There are many ways to define the attack surface of an embedded system. A few examples are given below.

2.1 A Simple Embedded System

In a simple embedded product, security can be very straightforward where the following conditions are met:

- The software is running from FLASH;
- All the software is trusted;
- The product is in a secure area;
- The system has a very defined and limited interface.

This could be, for example, a remote sensor with a network connection illustrated in Figure 1. Here the attack surface could be placed around the outside of the product. There is only one attack vector, the network channel. To mitigate the security risk of the network channel, a developer could use public/private key authentication and encryption to protect the system.

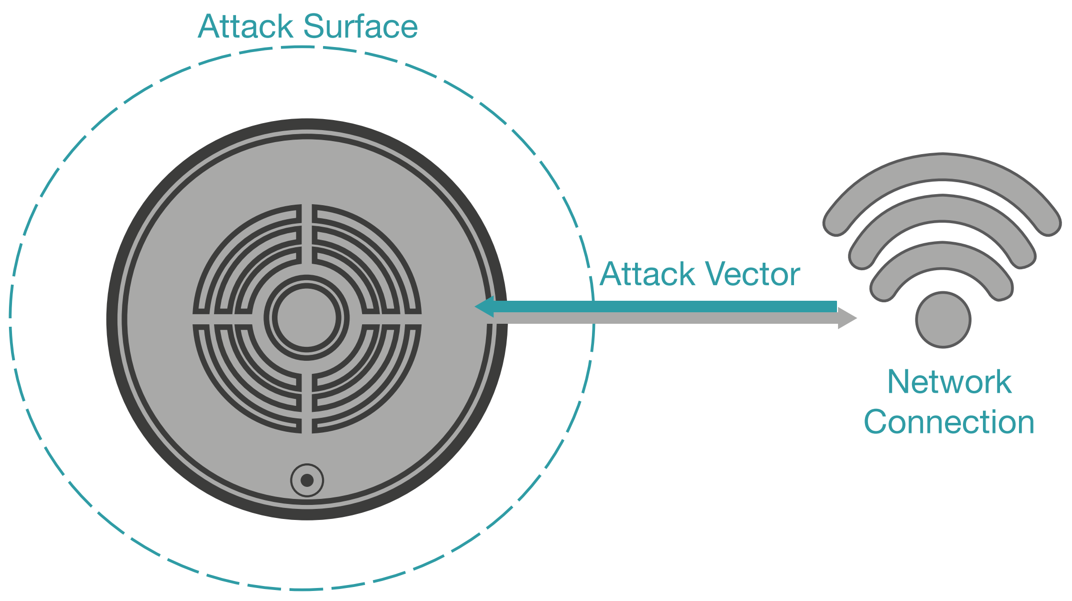

Figure 1. *Potential attack surface for a simple remote sensor*

2.2 An Embedded Medical Device

Complexity is added to a system when more than one processor is involved. For example, Figure 2 illustrates a medical device consisting of two processors. The first processor is a safety processor implementing the safety critical algorithms, and contains sensitive patient Health Care Records. The other processor, an application processor, contains networking stacks, data monitoring software, and graphics libraries, developed from various third party software products. The communication channels between the processors is an SPI link and various digital IO control signals. The developer could determine that the system needs to secure the safety critical operation and the patient Health Care Records, hence an attack surface could be placed around the safety processor.

A threat vector could be a back door into the system contained within the third party software that allows a bad actor to gain remote access to the system. To mitigate this risk security could be built into the communication channel between the two processors, limiting the information obtainable by the application processor. If this is not possible, then the software that controls the interface may need to be hardened.

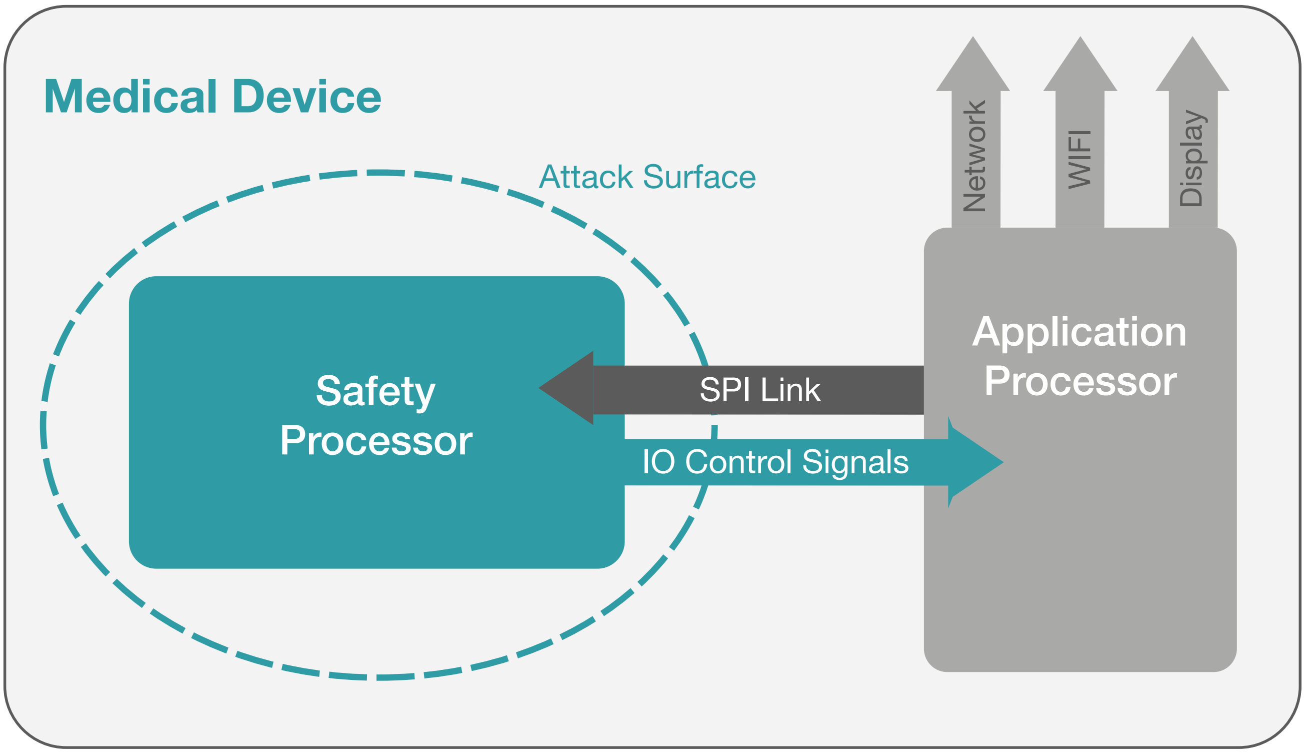

Figure 2. *Potential attack surface for a dual core embedded medical device*

2.3 Complex Embedded Systems

In high value, complex systems, security is more complicated and will have to be addressed on several levels. Here we will use the example of an autonomous vehicle control system, consisting of a multi-core System on Chip device.

A multi-core, System on Chip, autonomous vehicle control system will consist of processor cores undertaking safety operations and handling sensitive data, and other processor cores that may contain large amounts of commercial grade third party code. There will be numerous high-speed interfaces and network connections, some of which may not be encrypted or authenticated. See Figure 3.

The initial attack surface should be placed around the System on Chip device; however, this attack surface may have already been breached by the inclusion of third party code. There is also the risk of a security breach through one of the unencrypted/unauthenticated interfaces. It is good practice to minimise the attack vectors on the outer attack surface, but other measures are still required.

Given the outer attack surface is porous, each processor core needs to provide internal protection mechanisms to prevent, detect and slow bad actors from gaining access to sensitive data, or gaining control of the system. One approach is to use an RTOS that can constrain a Task's available access to a limited range of pre-allocated memory regions and RTOS resources.

In this scenario a bad actor could breach the system through a weakness in the interface, and gain access to the Task that manages the interface. However, the RTOS would contain the bad actor to this single compromised Task, and prevent it from moving deeper into the system, by limiting the access this Task has to the rest of the system.

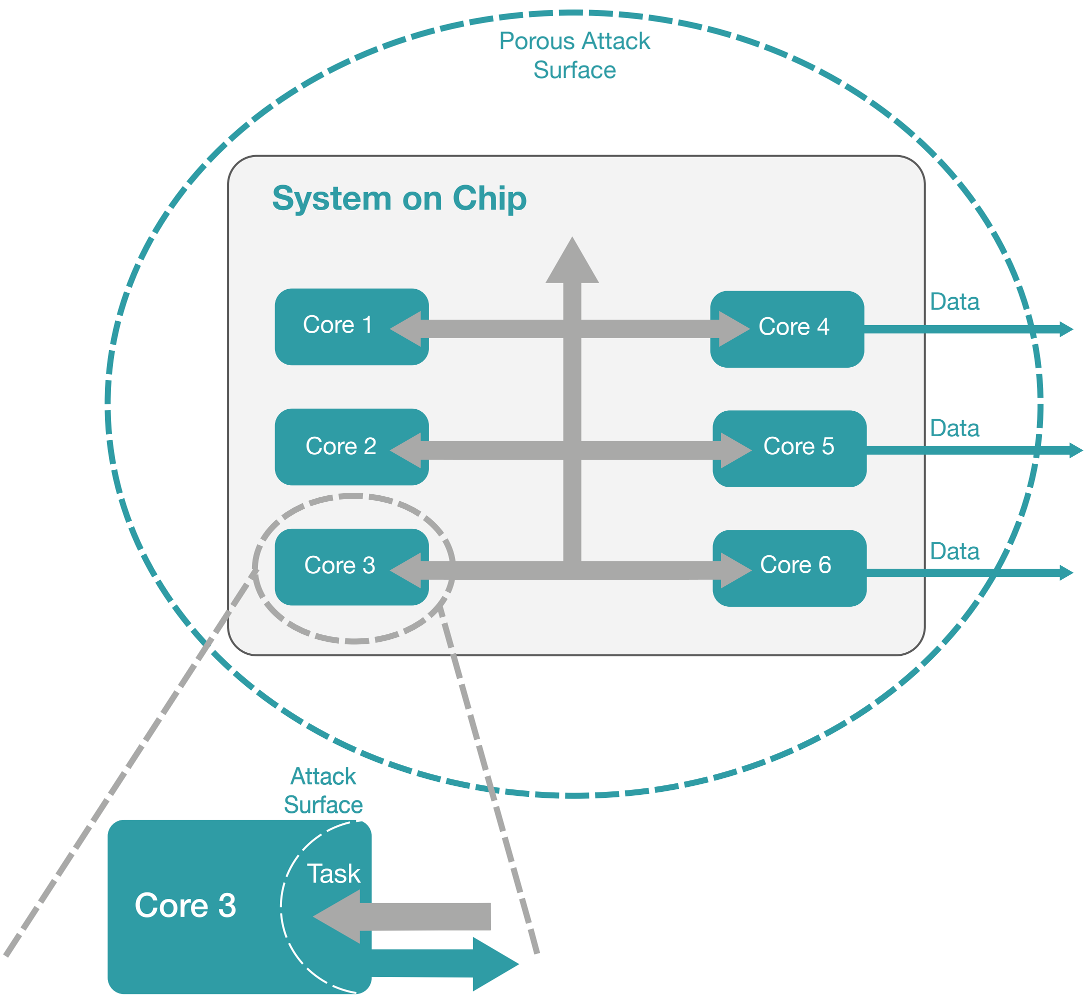

Figure 3. *Potential attack surfaces for a complex embedded system*

CHAPTER 3 SAFE**RTOS**® AND SECURITY

3.1 SAFE**RTOS**® Task Spatial Separation

SAFE**RTOS**® is a safety critical RTOS from WITTENSTEIN high integrity systems. SAFE**RTOS**® has been designed for use in systems across the Automotive, Medical, and Industrial market sectors, where the end product requires safety certification.

A full description of SAFE**RTOS**® can be found by following the link www.highintegritysystems.com.

SAFE**RTOS**® uses the processor's Memory Protection Unit (MPU), or Memory Management Unit (MMU) to create spatial separation between individual Tasks. This prevents one Task from overwriting the memory space of another, minimising the possibility of corrupted memory causing failure of the application.

MPUs allow regions to be defined in memory. Each region consists of a memory range and associated access permissions; read-only, write, execution, and user or privileged rights The processor will generate an exception if an illegal access is detected. This can be used to protect access to both memory, registers and peripherals.

The number of MPU regions available is processor specific. MPU memory regions are reserved for kernel code and data, to protect SAFE**RTOS**® from unauthorised access by the application. The kernel code and data regions are allocated to memory with privileged rights. The remaining MPU regions are allocated to Tasks and these can be set at the appropriate access level (user or privileged).

MPU memory regions relating to Tasks are reprogrammed at each context switch, enabling each Task to have an individual memory access profile.

MPU memory regions are assigned to a Task and configured as read-only, write or execute when the Task is created. The MPU regions can be reconfigured at runtime. A user-mode Task can only access its own stack and the user-defined memory regions, but in privileged mode it has access to all memory regions. Therefore Tasks should not run in privileged mode unless this is absolutely necessary, as they can overwrite both user and privileged memory.

User mode Tasks can pass messages to each other using the standard queue and semaphore mechanisms. Shared memory regions can be explicitly created but this is discouraged.

Calling an API function will require a temporary switch to privileged mode. All non-stack data required by the RTOS kernel is also stored in privileged memory areas.

Interrupt handlers run in privileged mode and therefore typically are not constrained by the MPU.

The SAFE**RTOS**® spatial separation feature has been developed to guard against a Task making an unintentional access to an incorrect memory region due to a memory corruption or programming error. It has not been designed to guard against a deliberate and malicious attempt to gain access to restricted memory.

3.2 SAFE**RTOS** Enhanced Security Module

The SAFE**RTOS**® Enhanced Security Module (ESM) hardens the spatial separation between user mode Tasks. Its objective is to prevent a compromised user mode Task from obtaining information from other Tasks, gaining control of the system, and to reduce the effects of a DoS attack.

Data breaches of embedded systems rarely happen on the first attempt; normally bad actors will have to probe and learn the system architecture to detect areas of weaknesses that can be exploited, which usually results in abnormal system behaviour. To help prevent security breaches, the SAFE**RTOS**® ESM supports a penetration detection monitor, used to detect and report any abnormal system behaviour back to the application.

When designing the ESM, the attack surface considered was the boundary of a user mode Task. The objective of the ESM is to reduce the attack surface of a user mode Task to a minimum, thereby restricting a bad actor to just the compromised user mode Task, and preventing access to the rest of the system.

The ESM is constructed from a series of security features that constrain the access a user mode Task has with the rest of the system.

3.3 Access Control Policy

The Access Control Policy (ACP) allows developers to restrict the number of SAFE**RTOS**® APIs each individual Task can access. For example, a Task could be blocked from changing its privilege level, changing its access rights, or creating new system resources (queues, semaphores, or new privilege mode Tasks etc). This reduces the attack surface for each Task thereby decreasing the risk of a compromised Task gaining access to other parts of the system.

In a system that has been designed for security, each Task may only require access to just a few SAFE**RTOS**® API calls, which greatly limits the effects individual Tasks can have on the system.

3.4 Object Access Control Policy

RTOS Objects such as Queues, Semaphores, Event Groups etc. allow data passing and synchronization between Tasks.

The Object Access Control Policy (OACP) restricts the number of RTOS objects each Task has access to. This constrains the Task to accessing only those RTOS objects that have been assigned to the Task during the initialisation process, and hence closes possible access points into the system.

3.5 Task/Object Handle Obfuscation

When using SAFE**RTOS**® the memory location of the RTOS Objects and Task Control Blocks (TCBs) is user defined, and should be in global memory. The RTOS Objects and TCBs are referenced by Handles, which are basic ‘C’ pointers. If a Task has access to a Handle, then it can use this knowledge to determine the memory location of the RTOS Object or TCB, allowing a Task to access the RTOS Object or TCB and indirectly obtain access to other system data and information.

The ESM replaces the traditional SAFE**RTOS**® Handles, with indirect Object IDs. The ESM contains a cross referencing table that allows SAFE**RTOS**® to match the Object ID with a specific RTOS Object or TCB memory block.

By removing the pointer belonging to the Handle Tasks are prevented from learning where key items of data are stored in memory, and limits the amount of information a bad actor can learn about the other resources within the system is reduced.

3.6 Memory Configuration

SAFE**RTOS**® and its ESM must be located in memory with privileged only access rights. It is highly recommended that the TCBs, RTOS Objects, Error Hooks and kernel callback functions are all located in privileged memory. This reduces the amount of sensitive information stored within User space, and prevents direct access to the data from a user mode Tasks.

For example, a Task's MPU memory regions are stored within its TCB, to ensure a Task cannot gain access to its own MPU configuration strict spatial separation is required, this is achieved by setting the Task to user mode and placing the TCB within privileged memory.

The Tasks MPU regions must be configured to constrain the Task to the smallest memory region possible. Care must be taken when granting a Task access to processor registers or peripherals. The smaller the memory regions the Task can access, the smaller the attack surface.

3.6 Security Aware Portable Layer

SAFE**RTOS**® consists of two functional code layers. There is the core SAFE**RTOS**® code layer, containing the SAFE**RTOS**® API and RTOS functionality that is common between all variants of SAFE**RTOS**®. This accounts for around 80% of the SAFE**RTOS**® code base. The remaining 20% relates to the portable layer, used to encapsulate the specific processor and compiler functionality.

The ESM requires a new security aware portable layer which provides a harder boundary between privileged and user mode memory spaces when accessing RTOS APIs, and can even prevent a user task from elevating its own privilege.

3.7 Penetration Detection Monitor

The SAFE**RTOS**® kernel will monitor for violations and breaches of the ACP, the OACP, or any API failure. These are reported to the application using a new centralised error reporting mechanism. The centralised error reporting allows the system to respond quickly if a compromised task is probing the system.

3.8 SAFE**RTOS** ESM Usage Model

SAFE**RTOS**® and its ESM are initialised and enabled as one of the final operations of the 'Root of Trust' boot sequence. The SAFE**RTOS**® and the ESM, being part of the Root of Trust, has its kernel code and data memory segments set to privileged mode.

Where possible all Tasks should be configured to user mode. Interrupt Service Routines (ISRs) have to be set to privileged mode, but all ISRs should be kept as short as possible, and where possible a Task Notification or Event Group should be used to trigger Tasks.

Tasks should not be set to privileged mode. However, if this is unavoidable, then no privileged Task should contain an interface that crosses the attack surface. Any security attack will need to cross the attack surface, and it is very important that once in the system the bad actor is contained within a user mode Task that manages that interface.

Care must be taken when setting the Task priority levels; the lower priority, the less impact the task can have on a running system.

The Tasks MPU regions must be configured to constrain the Task to the smallest memory region possible. Care must be taken when granting a Task access to processor registers or peripherals.

CHAPTER 4 Conclusion

4.1 Conclusion

This white paper has looked at what security typically means to an embedded system. No one action will make an embedded device secure, just as there is no one security risk. Thorough security requires a multi layered approach.

The Enhanced Security Module reduces the attack surface of a user mode task to a minimum, which helps constrain a bad actor to a single Task. This helps prevents bad actors from moving deeper into the system, by limiting the access this Task has to the rest of the system.

This white paper demonstrates how SAFE**RTOS**® and its Enhanced Security Module can be used to provide internal protection mechanisms to prevent, detect, and slow bad actors from gaining access to sensitive data, or gaining control of the system.

Contact Information

Your Feedback

We would be pleased to receive your comments and any suggestions for improvement to our software and documentation. Please email us at: support@highintegritysystems.com

Contact WITTENSTEIN high integrity systems

Address:	WITTENSTEIN high integrity systems Brown's Court, Long Ashton Business Park Yanley Lane, Long Ashton Bristol, BS41 9LB England
Americas	+1 408 625 4712
ROTW	+44 1275 395600
Email:	support@HighIntegritySystems.com
Website	www.HighIntegritySystems.com

All Trademarks acknowledged.