embeddedworld2019
Exhibition&Conference
...it's a smarter world

A New Zero-Trust Model for Securing Embedded Systems

Chris Conlon
wolfSSL Inc.
Bozeman, Montana, USA
chris@wolfssl.com

Cesare Garlati
prpl Foundation
Santa Clara, California, USA
cesare@prplFoundation.org

***Abstract* — The attack surface in embedded systems has grown exponentially as connectivity requirements are increasingly met with the integration of readily available 3rd party libraries. A new Zero Trust Model is required to address the intrinsic security threat posed by the resulting monolithic firmware. This paper explores a new modern approach based on open source hardware and software where security through separation is achieved via a state-of-the-art multi-domain Trusted Execution Environment (TEE) for RISC-V processors.**

Keywords — Embedded security; Internet of Things; IoT; TEE; Trusted Execution Environment; RISC-V.

I. INTRODUCTION

Embedded devices are a part of the daily lives of people all around the world. As devices get more personal and become placed in increasingly sensitive environments, the security of those devices becomes paramount. Security is a multi-tier approach, with different solutions being used across the industry depending on device capabilities and functionalities. Most security challenges faced by those resource-constrained devices that make up the Internet of Things can be minimized by enforcing physical separation between functional blocks and by properly implementing established encryption schemas to protect data in transit and at rest.

II. A MULTI-TIER APPROACH TO SECURITY

Embedded devices come in all shapes and sizes - small or large, resource constrained or memory abundant, slow or fast, with or without on-board peripherals, bare metal or with operating system, network connected or air gapped, machine-to-machine oriented or human driven. This heterogeneous landscape means that how each of these devices approaches the topic of security can also be as diverse as their component makeup. In bare metal environments there may be very little to no security in place. Mid-range devices will commonly run a Real Time Operating System (RTOS) offering some level of functional separation through tasks. Higher-end systems may run rich operating systems – such as Linux, which typically provide some additional security features including privilege levels and virtual memory. State-of-the-art security however requires a multi-tier approach all way from the processors up to operating systems and application and networking levels.

In addition, security should be implemented with multiple rings of protection so that a single failure point, such as a vulnerable library component, does not result in a complete system breach.

III. SECURITY THROUGH SEPARATION

Unfortunately, the security of today's embedded devices leaves much to be desired as they are essentially developed as a monolithic block of firmware. Many embedded systems don't provide a means of separation between functional components. Components run with access to the same memory space and peripherals, and may run under the same – single - privilege level. Security can only be as strong as the weakest link in the chain. If one component is compromised, an attacker could use that component as an entry point into the system - taking control of device functionality, causing incorrect behavior, or at the extreme putting lives at risk in safety critical applications.

The most effective means of separation starts at the hardware level. Some embedded platforms provide virtual memory that can help with separation between tasks and/or processes. Virtual memory uses a hardware-based memory management unit (MMU) to assign and manage separate virtual memory regions between tasks or processes. Each virtual memory region starts at address zero, making it easy for applications to use the address space as if they were the only ones on the system.

Virtual memory brings with it a benefit for increased security, and provides one foundational layer in a multi-tiered approach to securing embedded devices. Using virtual memory as one security layer, embedded engineers can begin to create a system-wide security plan by adding additional security measures to the other components in the system. This may include adding security to device communications via SSL/TLS, encrypting sensitive data at rest, or sanitizing user interaction at the application level.

However, Virtual memory suffers from three intrinsic limitations: it requires dedicated hardware such as an MMU, it introduces a large software attack surface necessary to operate the MMU – current Linux kernel is approximately 18 million lines of code, and it only provides separation between kernel and user space. The kernel remains the single point of failure where the most damaging vulnerabilities occur. [1]

A. *Minimizing Code Complexity*

Software is a product of the human intellect and as such all code inherently contains bugs. Bugs can easily turn into security vulnerabilities that may someday compromise a device. To minimize the overall security risk, all software should be as transparent and as simple as possible. To understand how code size can affect the number of defects in a device, we can look at one of the most widely used software packages: the Linux kernel [2]. At the time of writing, running the open source "*gitstats*" tool [3] over the Linux kernel codebase shows a total of 18 Million (18,349,938) lines of C code alone included in the kernel repository.

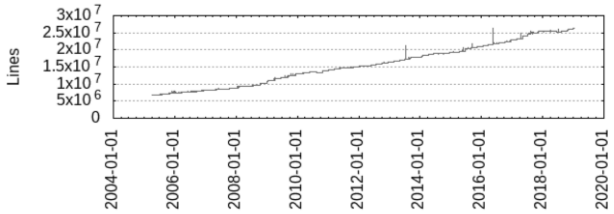

Fig. 1. Lines of Code in Linux Kernel (gitstats)

Assuming that all 18 Million lines of code are compiled as part of the kernel, how many possible bugs or defects could be included in those lines of code? Best practices yield an exploitable defect rate of 0.1 per 1000 lines of code [3][4], in this case projecting 1,835 possible vulnerabilities.

B. *Third Party Software Risks*

Most commercial products are not developed by any single vendor. They are rather composed of hundreds of hardware and software libraries combined with some unique code to meet the functional and budgetary constraints of the system. These 3rd party libraries pose an increasing threat vector for embedded systems as they may contain intentional or unintentional exploitable vulnerabilities.

In some cases, this 3rd party software is available in the form of source code. This is certainly good for transparency and maintenance, although downstream developers rarely have the expertise or the resources to perform independent code review and formal verification. In other cases, such as 3rd party security or DRM libraries, the software is only available in the form of object code or binaries. This black box approach precludes independent code review and forces the adopter to trust with their product the whole upstream supply chain responsible for the code base.

If this 3rd party code is assembled without any functional separation, then a single vulnerability can compromise the whole system. A new paradigm of a Zero Trust Model is necessary to protect a system from these new system-internal threat vectors.

IV. Implementing a Zero Trust Model

A Zero Trust Model in the scenario described above requires separation between each primary threat vector and the core function of the system. Envision a simple system which contains the following elements:

1. An exposed network interface.
2. A set of secrets used to secure the network interface and validate the system back to a host.
3. A real-time operating system running a set of business logic functions – tasks.
4. A set of actuators to control external physical systems.

Each one of these elements would likely contain 3rd party code and thus should be isolated from the other elements. The exposed networking interface and the secrets used to secure the network present an additional attack surface as they could be accessed externally and allow an attacker to penetrate a broader communication system.

Implementing a Zero Trust Model for this scenario requires at least four different blocks of separation or containers. As noted earlier, MMUs allow separation only between user and kernel functions – several of these blocks are traditionally built inside a monolithic kernel and thus impossible to separate and protect using virtual memory and MMU.

A traditional TEE such as ARM's TrustZone [6] provides only two areas: a secure world which is typically used to store secrets and a rich OS world which is typically used to do everything else. This antiquated design fails to provide enough blocks to properly implement the proposed Zero-Trust Model. In fact, it forces the system designer to combine operating system, network stack and business logic inside a single "world" where the exploit of any block vulnerabilities results in the whole system breach [reference].

V. Leveraging free and open RISC-V ISA

Oftentimes the implementation of an embedded TEE requires additional specialized hardware, internal or external to the main processor. In practice, this makes the TEE implementation very difficult if not impossible. On the contrary, the free and open RISC-V ISA specifies most of the TEE hardware requirements as part of the Privileged specs, which makes the TEE hardware enablers available out-of-the-box in any RISC-V processors without the need for additional specialized hardware.

RISC-V is an open instruction set architecture (ISA) [7][8], that offers innovative features helpful to a TEE implementation. Some of these features include privileged execution levels, physical memory protection (PMP), and user-mode interrupt delegation. The first helpful feature of the RISC-V cores is privilege levels. At any time, a RISC-V hardware thread, or *hart*, runs at a specific privilege level. Privilege levels include User/Application (U), Supervisor (S), and Machine (M). There is a fourth privilege mode for Hypervisor (H) that is reserved in the current specification. These privilege levels are used to provide protection between different components of the software stack (RISCV Privileged v1.10). An exception is raised if a *hart* performs operations not permitted by its

designated privilege level. A second security built-in feature is the Physical Memory Protection (PMP) that allows the highest privilege level (M) to protect specific memory regions and allow access to them by only threads with a specific privilege level. This allow the partition of functionality between execution environments and other functional component behavior. A third security feature relevant to the TEE implementation is the "N" extension. It allows the interrupt controller to delegate - transfer - control directly to a secure user-level handler rather than processing it at the highest privilege level in the outer execution environment.

Level	Encoding	Name	Abbreviation
0	00	User / Application	U
1	01	Supervisor	S
2	10	*Reserved*	
3	11	Machine	M

Fig. 2 RISC-V Privilege Levels

To enable these hardware capabilities, we based our study on the open standard TEE developed by Hex Five Security – a member of the RISC-V Foundation – which is freely available on GitHub. Hex Five's MultiZone Security orchestrates all the RISC-V security building blocks to support an unlimited number of equally secure functional blocks called Zones. Each Zone is intrinsically secure as it runs at the User (U) level: if the Zone's code attempts to access any memory-mapped resource – ram/rom/peripherals – that is not specifically assigned to the Zone's policy, the processor itself raises a non-maskable exception and the access is the-facto prevented.

VI. Putting It All Together

Our practical demonstration system is comprised of a 32-bit RISC-V core [9], the MultiZone Security TEE [10], the real time embedded operating system FreeRTOS and some popular open source libraries for security – WolfSSL - and networking – picoTCP. Four Zones - security domains - are defined via policies. Each zone is physically separated and shares no resources with the others. Inter-zone secure communications are provided by the MultiZone TEE.

1. Zone number one runs FreeRTOS. Its built-in TCP/IP stack removed and replaced with an interface to the messenger infrastructure provided by the TEE. Three FreeRTOS tasks include: a CLI application providing a user console, a real-time C application controlling the movements of a robotic arm, and a PWM application controlling the fading of a multicolor led based on interrupts generated via push buttons.
2. Zone number two runs the PicoTCP TCP/IP server [11] secured with the WolfSSL TLS library [12. This isolate the remote connectivity attack surface from the real time operating system.
3. Zone number three runs the Root of Trust. It stores and provides the secret key needed to secure the TLS link. Secrets can only be accessed via secure messages.
4. Zone number four runs a bare metal terminal accessible via UART. This interactive application allows to verify and benchmark the performance of the TEE.

Fig. 1 - MultiZone system configured to run four independent zones.

To enable communications between the otherwise physically separated Zones, their library APIs are wrapped in MultiZone messages and routed via the secure infrastructure provided by the TEE. This technique allows quick and robust integration of the individual library functionality across Zones without having to rewrite any of their functions.

Performance test are performed via Zone number four and show median context switch overhead of 120 instructions, and memory allocation of 2kB RAM and 4kB flash for the MultiZone TEE. In aggregate this overhead is less than 0.1% of the CPU and negligible in terms of overall memory resources available.

A more sophisticated and performant implementation of the communication interface between operating system and TCP/IP stack may include dedicated shared split buffers protected via R/W security policies. This will likely be explored in a future follow-up research.

VII. Conclusion

Embedded device security requires a multi-tiered approach, where security through separation plays a key role in ensuring overall system security. Traditional trusted zone architectures appear antiquated and insufficient to protect the attack surface introduced by the rapid, unverified integration of 3rd party libraries.

This research shows how a new Zero-Trust Model is not only feasible but also well suited to take advantage of modern processor platforms, such as RISC-V, and open standard Trusted Execution Environments such as MultiZone.

References

[1] Ori Karliner, "FreeRTOS TCP/IP Stack Vulnerabilities Put A Wide Range of Devices at Risk of Compromise: From Smart Homes to Critical Infrastructure Systems." Zimperium, 18 October, 2018.

[2] Torvalds, L. (2019). Linux. Retrieved from GitHub repository, https://github.com/torvalds/linux

[3] Hokkanen, Heikki. (2019). GitStats. Retrieved from gitstats.sourceforge.net

[4] McConnel, S. (2004). Code Complete, Second Edition, Redmond, WA, USA: Microsoft Press. ISBN: 0735619670

[5] Cobb, R.H., & and Mills, H.D. (1990). Engineering Software Under Statistical Quality Control. IEEE Software, Volume VII (Issue 6), pp. 44-54.

[6] ARM. (2019). Introducing Arm TrustZone.. https:/developer.arm.com/technologies/trustzone

[7] RISC-V Foundation. (2017). The RISC-V Instruction Set Manual. Volume II: Privileged Architecture. https://riscv.org/specifications/privileged-isa/

[8] RISC-V Foundation. (2017). The RISC-V Instruction Set Manual. Volume I: User-Level ISA. Retrieved from https://riscv.org/specifications

[9] Hex Five Security Inc. "MultiZone Security", Git Repository, 2019. https://github.com/hex-five/ MultiZone-sdk

[10] Hex Five Security Inc. "MultiZone Security". White Paper, 2018. https://hex-five.com

[11] Altran Intelligent Systems. "picoTCP." 2018. https://github.com/tass-belgium/picotcp

[12] WolfSSL, Inc. "Embedded TLS Library." 2019. https://www.wolfssl.com/