Toradex

Swiss. Embedded. Computing.

Introduction to
Embedded Linux Security

Presented by

Toradex

WHAT WE DO

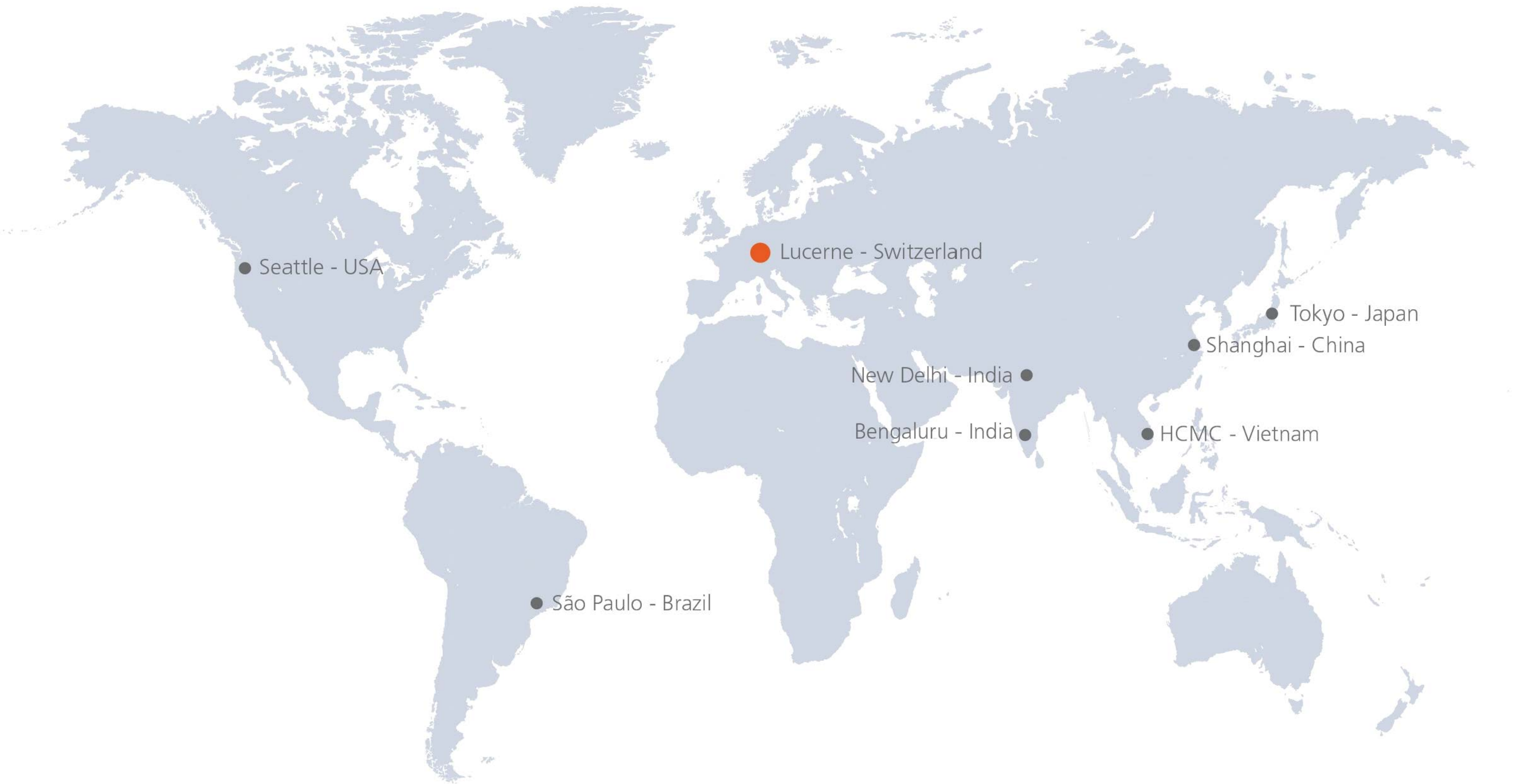
Make Embedded Computing Easy

Reliable Arm® System on Modules

Lowest Cost of Ownership

Industry-leading Support

NXP GOLD PARTNER

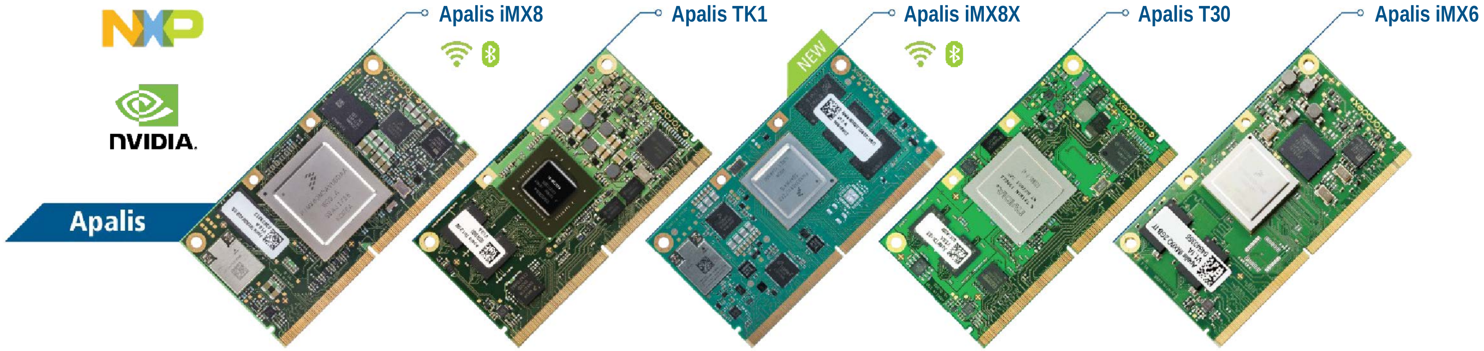


RELIABLE AND EASY-TO-USE EMBEDDED SOLUTIONS FOR YOU

PRODUCT PORTFOLIO

Toradex

Swiss. Embedded. Computing.



Apalis

Apalis iMX8

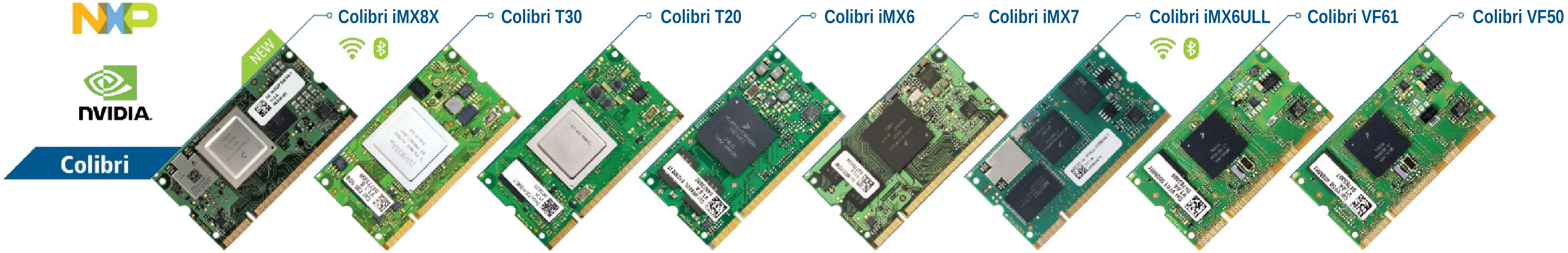
Apalis TK1

NEW

Apalis iMX8X

Apalis T30

Apalis iMX6



Colibri

NEW

Colibri iMX8X

Colibri T30

Colibri T20

Colibri iMX6

Colibri iMX7

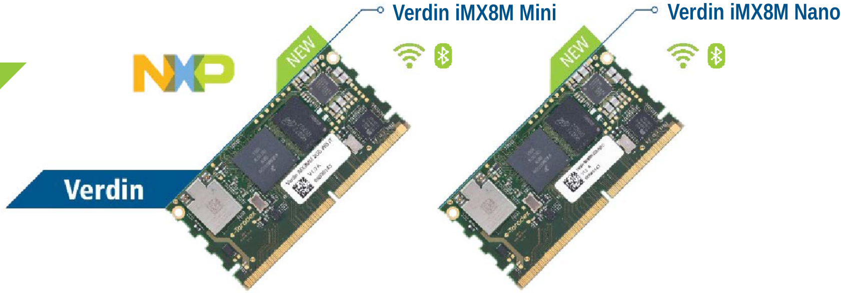
Colibri iMX6ULL

Colibri VF61

Colibri VF50

Torizon™

Introducing Verdin



Verdin

NEW

Verdin iMX8M Mini

NEW

Verdin iMX8M Nano

TYPICAL APPLICATIONS

TOP MARKETS

- Industrial Automation
- Medical / Laboratory
- Test & Measurement
- Transportation / Specialty Vehicles
- Building Automation
- Environment Monitoring

ANNUAL VOLUMES

100 to 50k

Torizon™

Easy-to-use
Industrial Linux Software Platform

Fast time-to-market
Ready-to-use
Linux distribution

Open Source
Based on open software,
No lock-in

Simple updates
Built-in, automotive-grade,
over-the-air update
capabilities

Secure
Frequent updates, accessible
security features

Real-time
Optimized real-time option

yocto PROJECT PARTICIPANT

docker

Visual Studio

CODESYS

INTRODUCTION TO SECURITY

- Computer security is basically the **protection of computer systems and information** from harm, theft, and unauthorized use.

- On an embedded device, both hardware and software need to be protected to some degree.

- The computer security field is really about managing risks!

SECURITY CONCEPTS I

- **Owners**: those who benefits from the product (user, manufacturer, business owner, etc).

- **Assets**: anything that has value to the owners (data, code, etc).

- **Threats**: anything that are capable of acting against an asset in a manner that can result in harm.

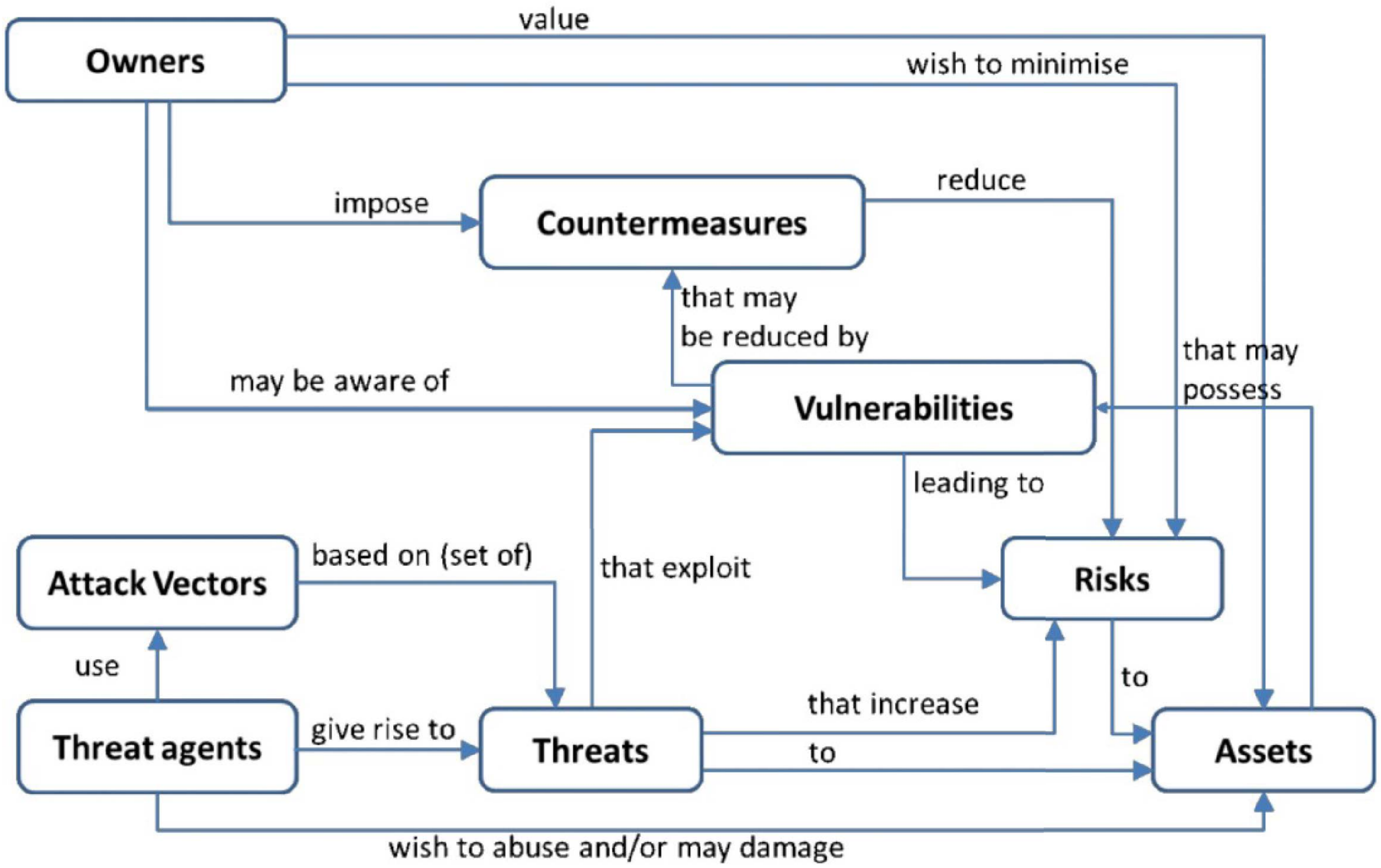
SECURITY CONCEPTS II

- **Threat actors** (or threat agents): person or thing that can manifest a threat (malicious hacker, government, etc).

- **Attack vectors** (or attack surface): methods or pathways used by a threat actor to access or penetrate the target system.

- **Vulnerabilities**: weakness which can be exploited by a threat actor.

SECURITY CONCEPTS III

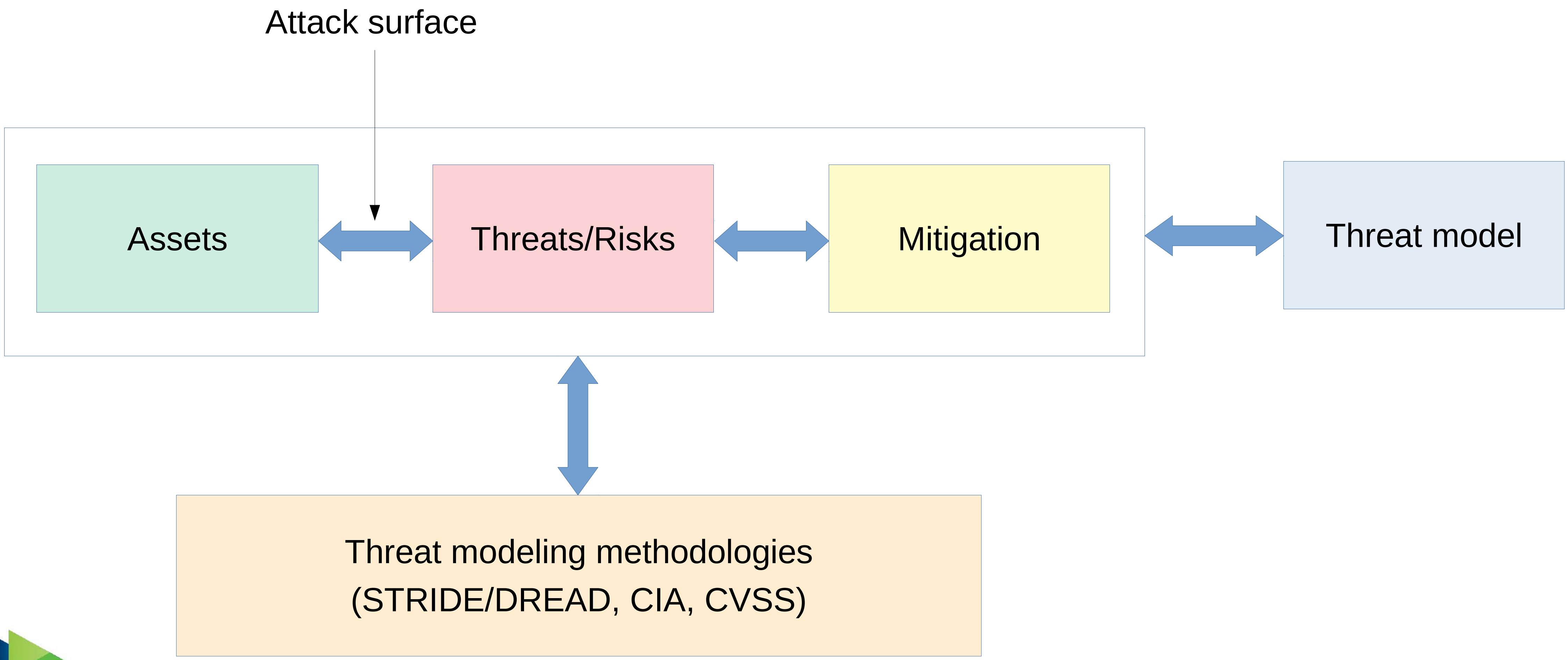


Source: Enisa - Hardware Threat Landscape and Good Practice Guide

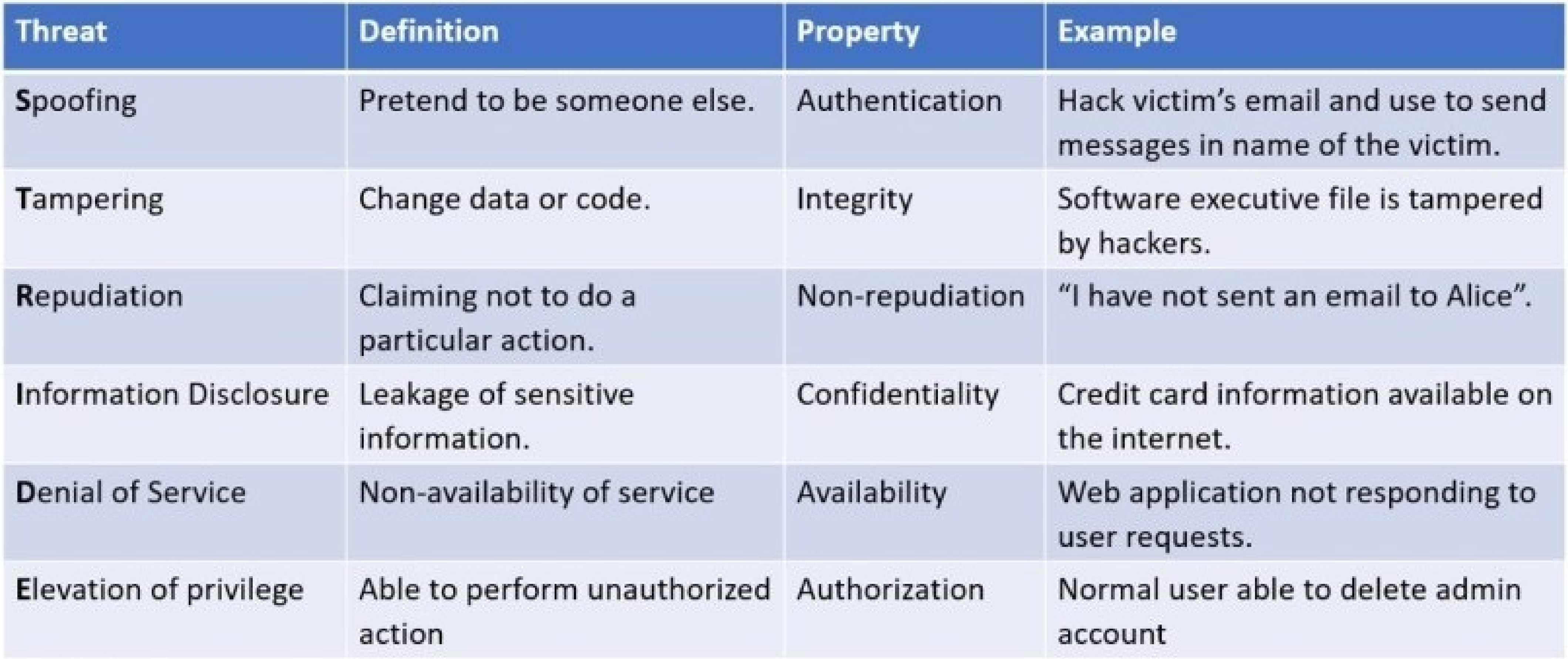
THREAT MODELING

- So security is all about identifying threats to minimize risks of assets being compromised.
- **Threat modeling** is a process where potential threats can be identified, enumerated, and mitigations can be prioritized.
- It is basically a risk assessment process where you evaluate the value of your assets and the cost to protect them.
- The result of threat modeling is the **threat model** of your product.

THREAT MODELING



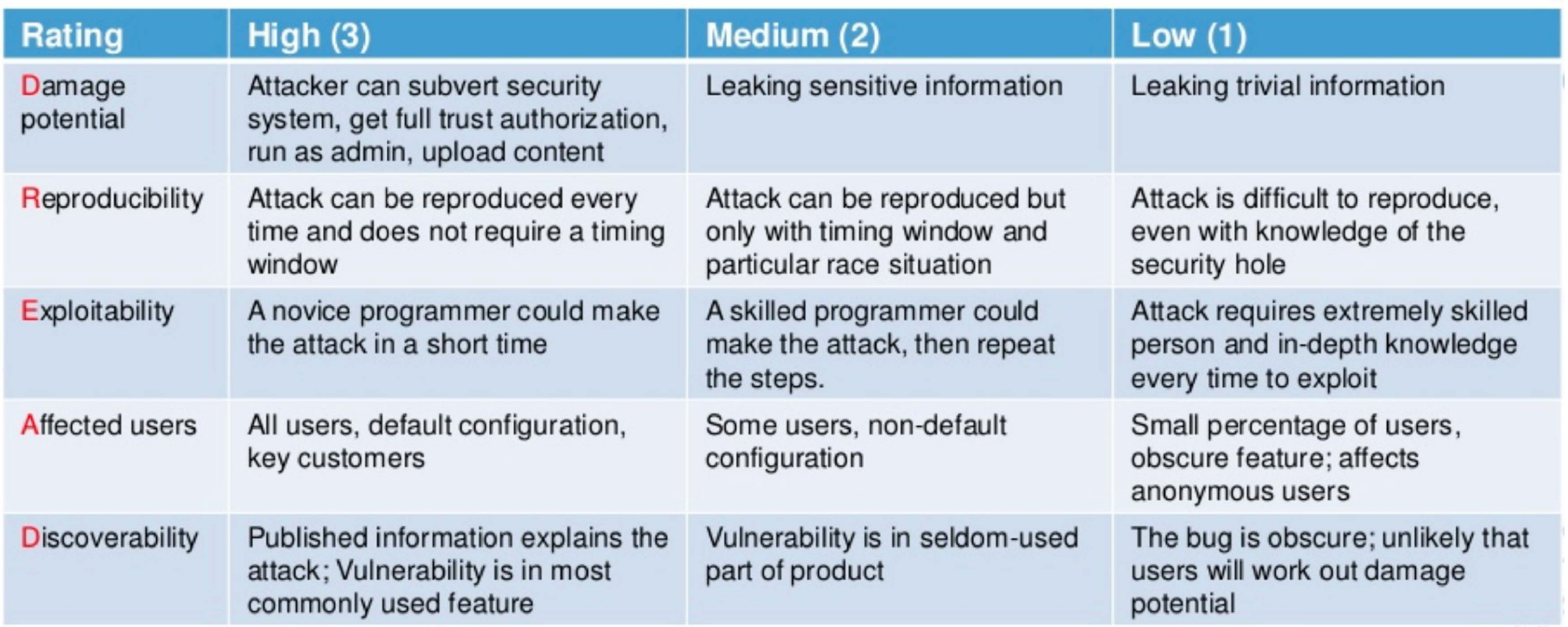
STRIDE



Threat	Definition	Property	Example
Spoofing	Pretend to be someone else.	Authentication	Hack victim's email and use to send messages in name of the victim.
Tampering	Change data or code.	Integrity	Software executive file is tampered by hackers.
Repudiation	Claiming not to do a particular action.	Non-repudiation	"I have not sent an email to Alice".
Information Disclosure	Leakage of sensitive information.	Confidentiality	Credit card information available on the internet.
Denial of Service	Non-availability of service	Availability	Web application not responding to user requests.
Elevation of privilege	Able to perform unauthorized action	Authorization	Normal user able to delete admin account

Source: https://allabouttesting.org/stride-acronym-of-threat-modeling-system/

DREAD



Rating	High (3)	Medium (2)	Low (1)
Damage potential	Attacker can subvert security system, get full trust authorization, run as admin, upload content	Leaking sensitive information	Leaking trivial information
Reproducibility	Attack can be reproduced every time and does not require a timing window	Attack can be reproduced but only with timing window and particular race situation	Attack is difficult to reproduce, even with knowledge of the security hole
Exploitability	A novice programmer could make the attack in a short time	A skilled programmer could make the attack, then repeat the steps.	Attack requires extremely skilled person and in-depth knowledge every time to exploit
Affected users	All users, default configuration, key customers	Some users, non-default configuration	Small percentage of users, obscure feature; affects anonymous users
Discoverability	Published information explains the attack; Vulnerability is in most commonly used feature	Vulnerability is in seldom-used part of product	The bug is obscure; unlikely that users will work out damage potential

Source: https://www.slideshare.net/SecurityInnovation/threat-modeling-to-reduce-software-security-risk

THREAT MODEL EXAMPLE

Threat	Score	Mitigation
Any user is able to login in the admin web page and change device configuration	14	Implement a web-based authentication mechanism
A network application could be exploited to run unauthorized code	13	Drop application privileges and run it inside a container
With physical access, a threat actor could extract user data	12	Use encryption to protect user data
Using a MitM attack, a threat actor could change the firmware image during the update process	11	Check the signature of the update image
A threat actor could execute DoS attacks on the device	11	Create firewall rules to avoid or minimize DoS attacks' impact

...

SECURE BOOT CONCEPTS

- The objective of a secure boot is to protect the integrity and authenticity of the code.

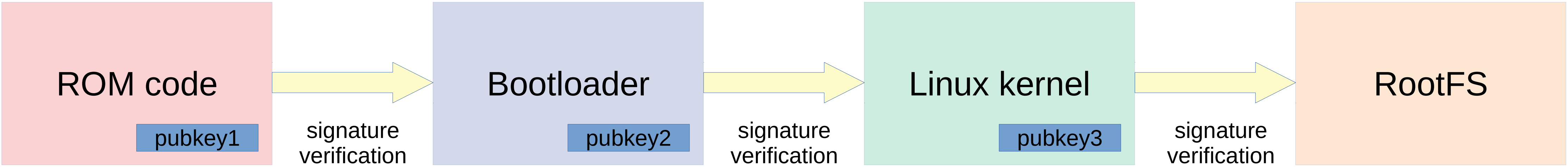
- Why? To make sure the binaries you're running were built by a trustworthy person or company!

- It has costs like key management, boot time, harder to develop on the platform, etc.

HOW DOES IT WORK?

- Everything is based on the verification of digital signatures (no encryption involved).
- The authenticity of every component of the system should be verified (bootloader, kernel, rootfs, etc).
- That means the first element in the boot process authenticates the second, that authenticates the third, etc.
- This is called a **chain-of-trust**.

HOW DOES IT WORK?



HOW TO IMPLEMENT IT?

- Everything starts in the ROM code inside the SoC (**Root of Trust**).

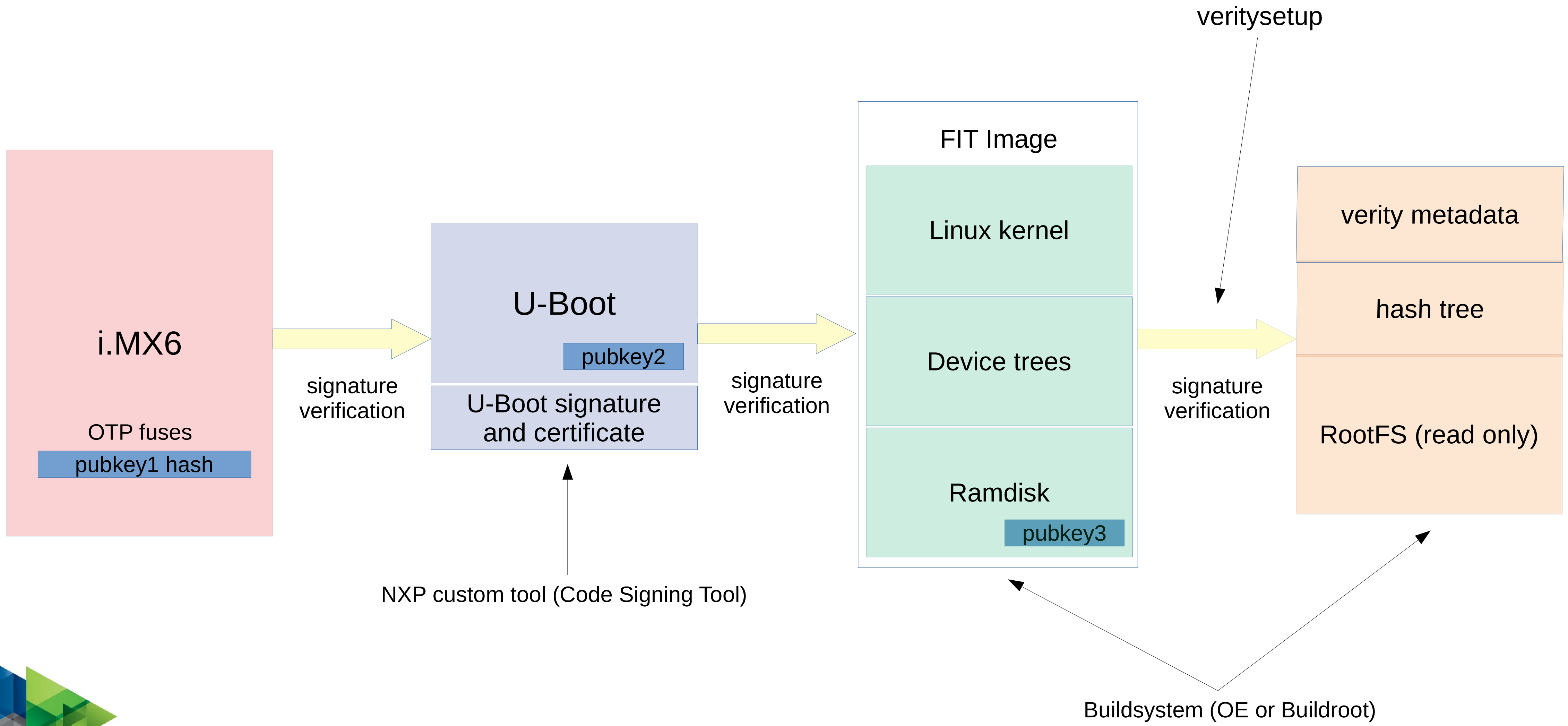
- The ROM code will check the signature of the bootloader.
  - It needs a way to store the public key(s) (e.g. OTP fuses).
  - To make it less expensive, usually only the hash of the public key is stored.

- The Bootloader (e.g. U-Boot) will check the integrity of the **FIT image**.
  - The FIT image is a container for multiple binaries with hashing and signature support.
  - It contains the Linux Kernel image, device tree files and an initial ramdisk.

HOW TO IMPLEMENT IT?

- The ramdisk will have the logic to verify and mount the rootfs using the **dm-verity** kernel module and the **veritysetup** tool.
  - The device-mapper verity provides integrity checking of block devices.
  - It requires a read-only rootfs (squashfs can be a good solution).
- The rootfs partition should generated with dm-verity support.
  - Another approach would be IMA or dm-integrity for read-write filesystems.
- This is only one example of secure boot implementation, although it could be applied to a different set of boards and ARM SoCs.

SECURE BOOT ON i.MX6



OOPS...

- Nothing is 100% secure!
- Secure boot vulnerabilities in ROM code of i.MX6, i.MX50, i.MX53, i.MX7, i.MX28 and Vybrid families publicly disclosed July 17th, 2017.

  https://blog.quarkslab.com/vulnerabilities-in-high-assurance-boot-of-nxp-imx-microprocessors.html

  https://community.nxp.com/docs/DOC-334996
- The vulnerabilities were fixed with new silicon.

  https://developer.toradex.com/knowledge-base/secure-boot-vulnerabilities-for-imx-vybrid-errata-err010872-err010873

CODE AND DATA ENCRYPTION

- While secure boot ensures authenticity, it does not protect the device from being counterfeited or prevent threat actors from extracting code or data from the device.

- If you want to protect your intellectual property or ensure data confidentiality, you will need to use encryption.

- It not common to encrypt the code on an embedded Linux system (but you could want to encrypt your applications).
  - Be aware of GPLv3 (Tivoization).

- On the other hand, application or user data confidentiality and protection might be a requirement, for example on medical devices.

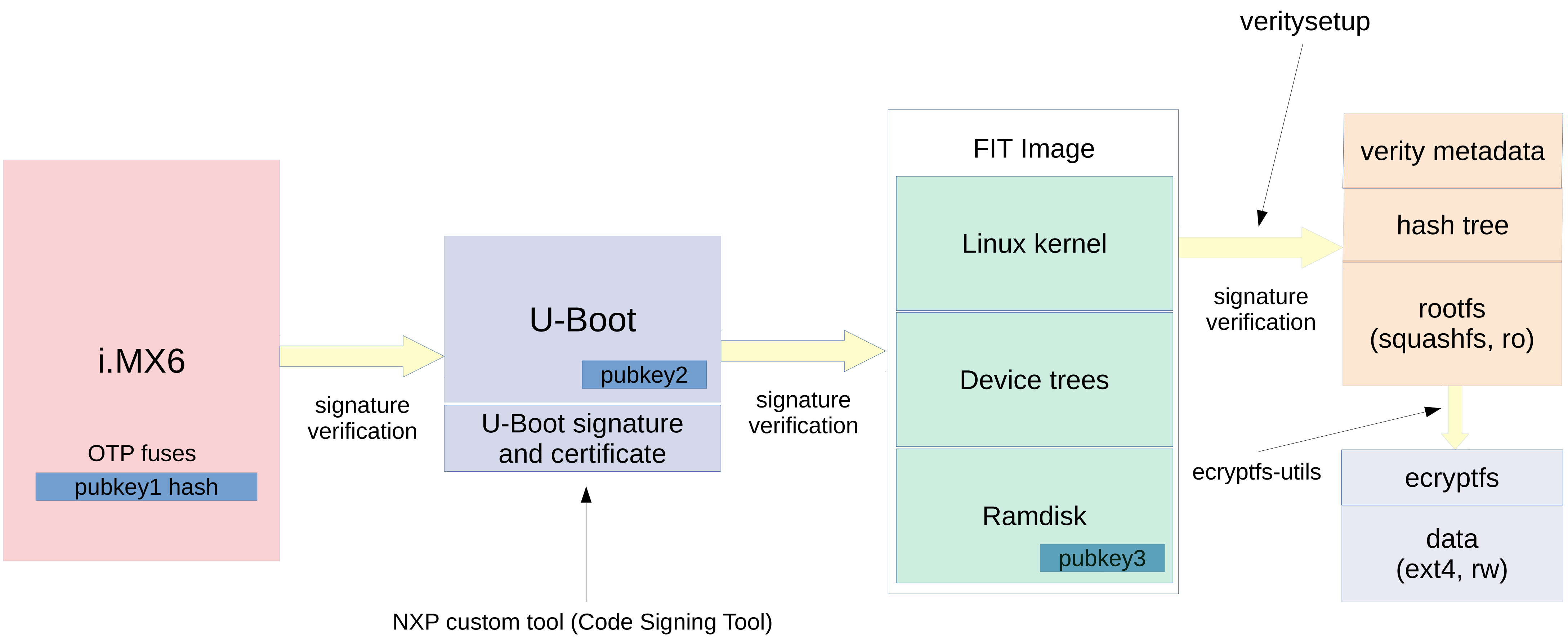
CODE AND DATA ENCRYPTION

- There are basically two main approaches to file encryption in Linux: full disk encryption and file-based encryption.

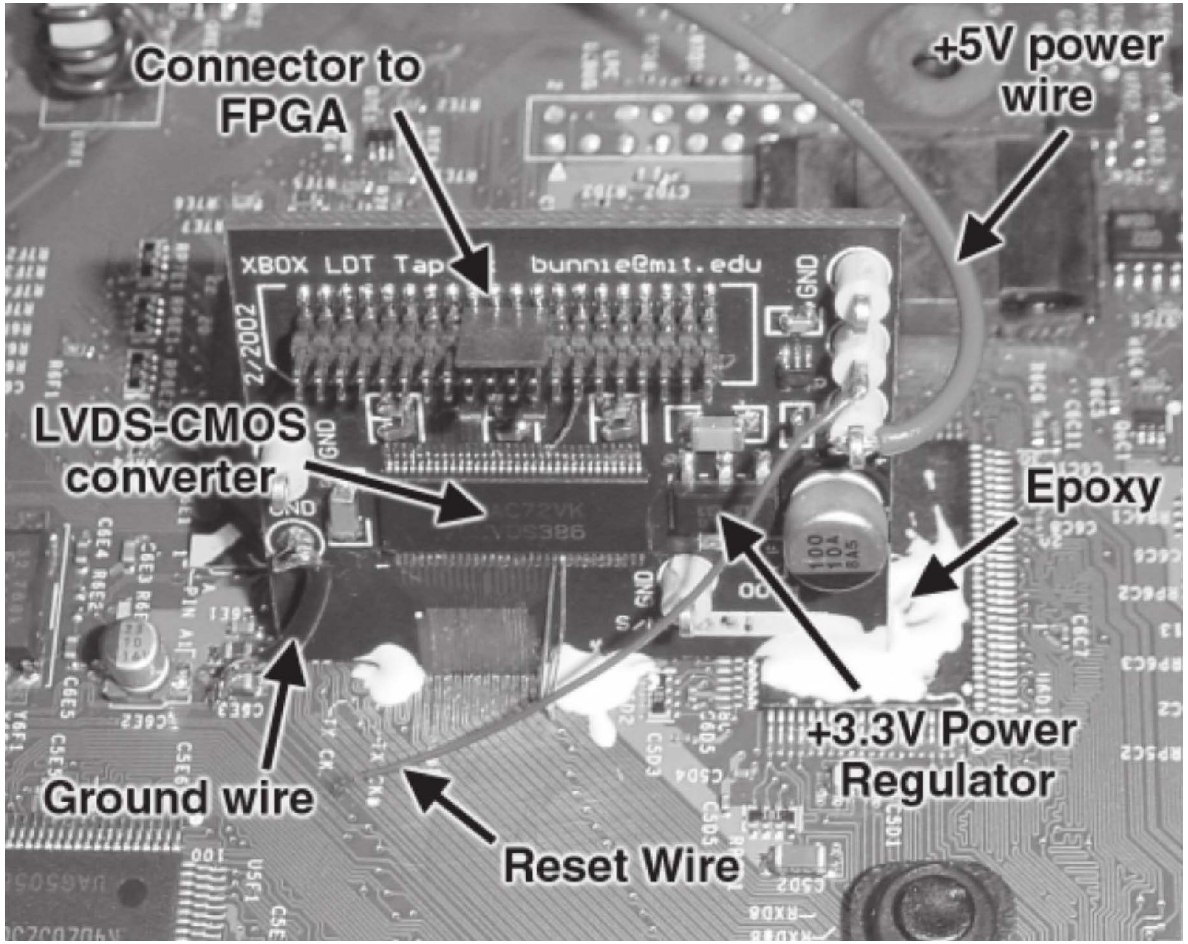
- A full disk encryption provides encryption at the block level and the whole disk or a disk partition is encrypted.
  - **dm-crypt** is the Linux kernel's device mapper crypto target.

- A file-based encryption provides encryption at the file system level, where each directory may be separately and optionally encrypted with a different key.
  - **fscrypt** is an API available on some filesystems like EXT4, UBIFS and F2FS.
  - **eCryptFS** is implemented as a layer that stacks on top of an existing filesystem.

SECURE BOOT WITH ENCRYPTION



WHERE IS THE KEY?



Source: https://bunniefoo.com/nostarch/HackingTheXbox_Free.pdf

PRIVATE KEY STORAGE

- A symmetric-key algorithm is usually used for encryption, so you have to store the private key somewhere in the system to decrypt the data.
- And the protection of the encrypted data is as secure as the protection of the key to decrypt it!
- On a desktop or smartphone, the key used to encrypt the filesystem is derived from a user password (passphrase) entered interactively.
- On an embedded system, it should be stored encrypted in the filesystem or in a secure storage isolated from the system.

KEY STORAGE ON i.MX PROCESSORS

- On i.MX, each processor has a unique master key (pre-programmed by NXP) that can only be accessed by the CAAM (Cryptographic Accelerator and Assurance Module) module.

- So the CAMM module can be used to encrypt the filesystem encryption key with the unique processor master key (this would have to be done during manufacturing).

- The encrypted key could be stored in the boot or rootfs partition.

- During boot, the CAMM module would be used to decrypt the key and restore the plain key that would be used to decrypt the filesystem.

KEY STORAGE ON EXTERNAL DEVICES

- If you don’t have security features in your processor, you could achieve the same results with an external hardware like a **Secure Element** or a **TPM** device.

- These external devices usually provide secure storage, so they could be used to store a master key that could be used to encrypt/decrypt the filesystem encryption key.

- These devices also offer a lot of security features like random number generation, hash calculation, crypto and signing functions, etc.

- Also, a **TEE** (Trusted Execution Environment) could also be used to securely store the key (we'll talk about TEE later in this presentation).

SECURE ELEMENT

- A Secure Element is a secure computing system.
- It is basically a secure storage with its own secure applications (usually implemented using Java Card, but not necessary).
- What a secure element does is very open and depends on the implementation, but most of them implement Public-Key Cryptography Standard 11 (**PKCS#11**).
- Examples of Secure Elements are smart-cards and SIM-cards.

TPM

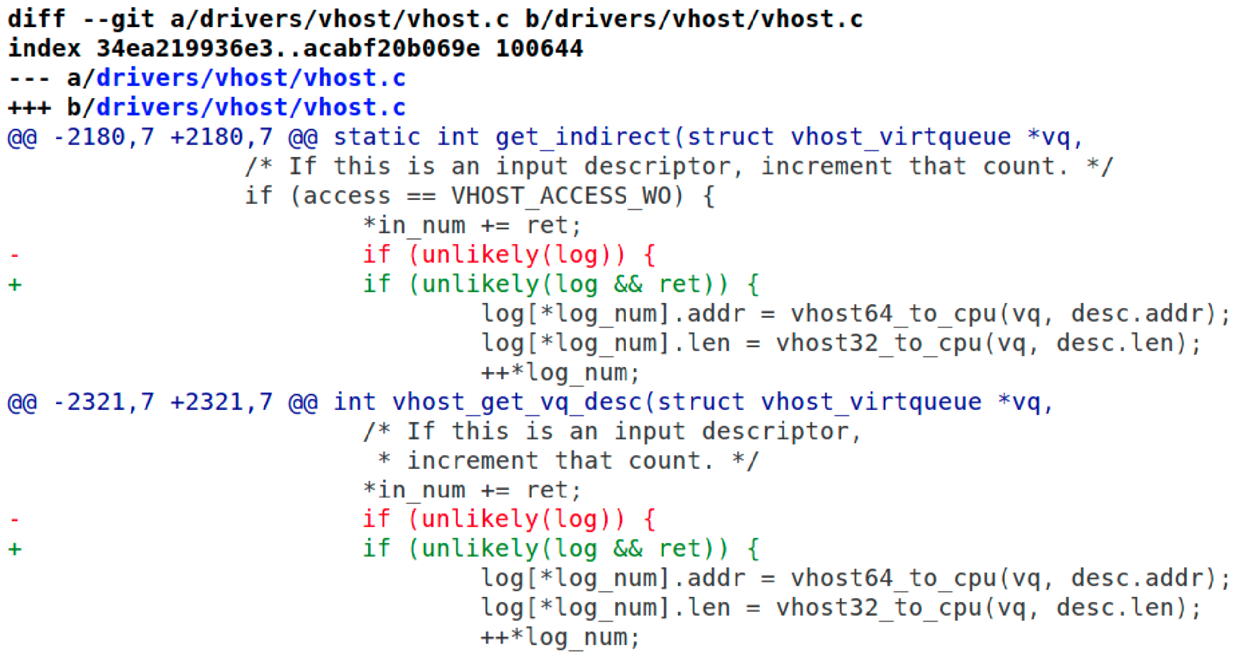
- A TPM (Trusted Platform Module) is a specification and an international standard (ISO/IEC 11889).
- TPM is not a Secure Element, although it could be implemented inside one.
- Can be implemented in hardware or software, but most implementations are in hardware.
- It provides a set of limited security features defined by the standard, including secure storage and cryptographic functions.

SECURE CODING

- You could protect your code and data with encryption, but if you running an application with bugs that could be exploited, your assets are still at risk anyway.
- If an application has attack vectors (user input, configuration files, network communications, etc), a bug could be used to exploit the application.
- Especially programs written in memory unsafe languages like C/C++, bugs like **buffer overflows** could be used in attacks like stack smashing and format strings.

CVE-2019-14835

- A buffer overflow flaw was found, in versions from **2.6.34 to 5.2.x**, in the way Linux kernel's vhost functionality that translates virtqueue buffers to IOVs, logged the buffer descriptors during migration. A privileged guest user able to pass descriptors with invalid length to the host when migration is underway, could use this flaw to increase their privileges on the host.



```
diff --git a/drivers/vhost/vhost.c b/drivers/vhost/vhost.c
index 34ea219936e3..acabf20b069e 100644
--- a/drivers/vhost/vhost.c
+++ b/drivers/vhost/vhost.c
@@ -2180,7 +2180,7 @@ static int get_indirect(struct vhost_virtqueue *vq,
                /* If this is an input descriptor, increment that count. */
                if (access == VHOST_ACCESS_WO) {
                        *in_num += ret;
-                       if (unlikely(log)) {
+                       if (unlikely(log && ret)) {
                                log[*log_num].addr = vhost64_to_cpu(vq, desc.addr);
                                log[*log_num].len = vhost32_to_cpu(vq, desc.len);
                                ++*log_num;
@@ -2321,7 +2321,7 @@ int vhost_get_vq_desc(struct vhost_virtqueue *vq,
                        /* If this is an input descriptor,
                         * increment that count. */
                        *in_num += ret;
-                       if (unlikely(log)) {
+                       if (unlikely(log && ret)) {
                                log[*log_num].addr = vhost64_to_cpu(vq, desc.addr);
                                log[*log_num].len = vhost32_to_cpu(vq, desc.len);
                                ++*log_num;
```

Source: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-14835

STATIC CODE ANALYSIS

- Always use static analysis tools to check your code and don't ignore compiler warnings.
- Static analysis tools are able to analyze the source code (without running the program) to find problems before they happen.
- These tools can find program errors like null pointer dereferences, memory leaks, integer overflow, out of bounds access, use before initialization, etc!
- There are many good open source (cppcheck, splint, clang, etc) and commercial (Coverity, PC-Lint, etc) options for static code analysis.

RUNTIME PROTECTIONS

- Build the system with protections (ASLR, stack canaries, electric fence, ASan, etc) and do dynamic analysis of you programs!
- **ASLR** (Address Space Layout Randomization) is a computer security technique that randomly arranges the address space positions of key data areas of a process (text, stack, heap, libraries, etc).
- **AddressSanitizer** (ASan) is an instrumentation tool created by Google security researchers to identify memory access problems in C and C++ programs.
- **Valgrind** could help to detect memory related problems like leaks and data races.

FUZZING TOOLS

- Test the system with fuzzing tools!
- Fuzzing or fuzz testing is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a program.
- The program is then monitored for exceptions such as crashes, failing built-in code assertions, or potential memory leaks.
- A lot of free and open source fuzzing tools are available, including **AFL** (american fuzzing loop) and **syzkaller** (Linux kernel fuzzer).

PERMISSIONS

- One way to mitigate vulnerabilities in programs is not running them with root (superuser) privileges!
- Also, design a system where you don't need a user logged as root, and disable root login.
- If you have an external connection to the system, use public-key authentication (if possible) and a strong password if needed.
- Always design with the principle of least privilege in mind!

ACCESS CONTROL

- But the problem is that sometimes we need "root powers" to execute some privileged operation like set the system clock or use RAW sockets.

- And then we need to run our program as root, right?

- Wrong! We have some options for fine-grained control over processes permissions.

- One of the solutions is called **Linux capabilities**.

LINUX CAPABILITIES

- Linux divides the privileges associated with superuser into distinct units, known as capabilities, which can be independently enabled and disabled.

- So the idea is to write a program that will run as root, but enable just the capabilities it needs to do its job.

```
$ getcap /usr/bin/ping
/usr/bin/ping = cap_net_raw+ep
```

- While capabilities provide a subset of the available root privileges to a process, it is not so flexible.

- If you need more control over permissions, you should think about using a type of access control called MAC (**Mandatory Access Control**).

DAC vs MAC

- Linux traditionally supports Discretionary Access Control (DAC).
- DAC is a type of access control where the access to objects is restricted based on the identity of subjects and/or groups to which they belong ("user" and "group" flags).
- Another type of access control is called Mandatory Access Control (MAC).
- MAC refers to a type of access control where the operating system constrains the ability of a subject to access or perform some sort of operation on an object.
- MAC is implemented in the kernel via Linux Security Modules (LSM).

LINUX SECURITY MODULES

- LSM is a framework that allows the Linux kernel to support a variety of computer security models.
- The most know Linux security modules that implement MAC are AppArmor, SELinux, Smack and TOMOYO.
  - **SELinux** is one of the most popular (and complex) MAC implementation, developed initially by NSA and today used in bigger projects like Android and Fedora.
  - **AppArmor** is also a popular and more user-friendly MAC implementation, supported by Canonical and used in some Linux distributions like Ubuntu and Debian.
- Although our objective isn’t to go over the details on SELinux or AppArmor, you should think about using a MAC if you need fine-grained control over process permissions.

APPLICATION SANDBOXING

- Sometimes, restricting permissions is not enough to protect the system from a vulnerable application, and to improve security, **sandboxing** could be used to isolate applications from the rest of the system.

- Possibly the oldest sandboxing tool available in Linux is **chroot**, but it’s not very useful in terms of security because it will only isolate the filesystem.

- Virtualization is another form of application sandboxing in Linux, but it is too costly, especially in embedded systems.

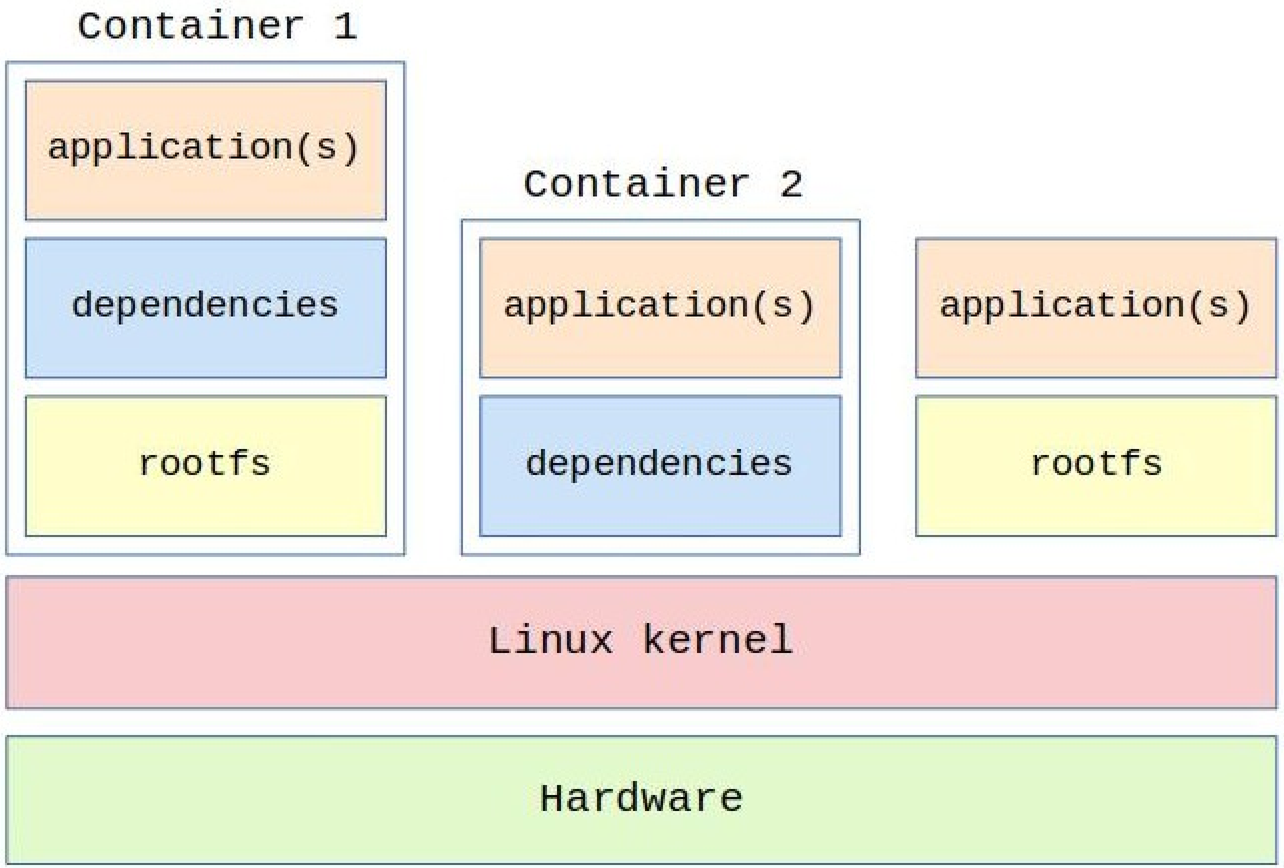
- Nowadays, two possible solutions to sandbox applications in embedded Linux are **Containers** and **Trusted Execution Environments** (TEE).

LINUX CONTAINERS

- A Linux container is a minimal filesystem with only the required software components to run a specific application or group of applications.

- Using some kernel features, the container will "run" completely isolated from the rest of the system (only the kernel is shared).
  - **namespaces** make it possible to isolate the execution of a process on Linux (PID, users, network connections, mount points, etc).
  - **cgroups** allows to partition system resources (CPU, memory, I/O) by process or group of processes.
  - **seccomp** allows to limit the system calls that a process can do.

LINUX CONTAINERS



CONTAINERS AND SECURITY

- Several tools are available to manage containers in Linux, including LXC, Systemd-nspawn, Podman and Docker.

- A container is not secure by itself, but if properly configured, we can limit the permissions of each process inside the container and control the communication between them, reducing the attack surface and improving the security of the product.

- Using in conjunction with a security module (e.g. AppArmor, SELinux), we can greatly enhance the security of the system.

TEE

- In a system based on containers, if the kernel is compromised, all the operating system is at risk. A Trusted Execution Environment could prevent that.

- A Trusted Execution Environment (TEE) is an environment where the code executed and the data accessed is isolated and protected in terms of confidentiality (no one have access to the data) and integrity (no one can change the code and its behavior).

- A lot of devices around us make use of a Trusted Execution Environment, including smartphones, set-top-boxes, videogame consoles and Smart TVs.

- TEE could be a good solution to store and manage encryption keys, store and manage credentials and sensitive data, and protecting digital copyrighted information.

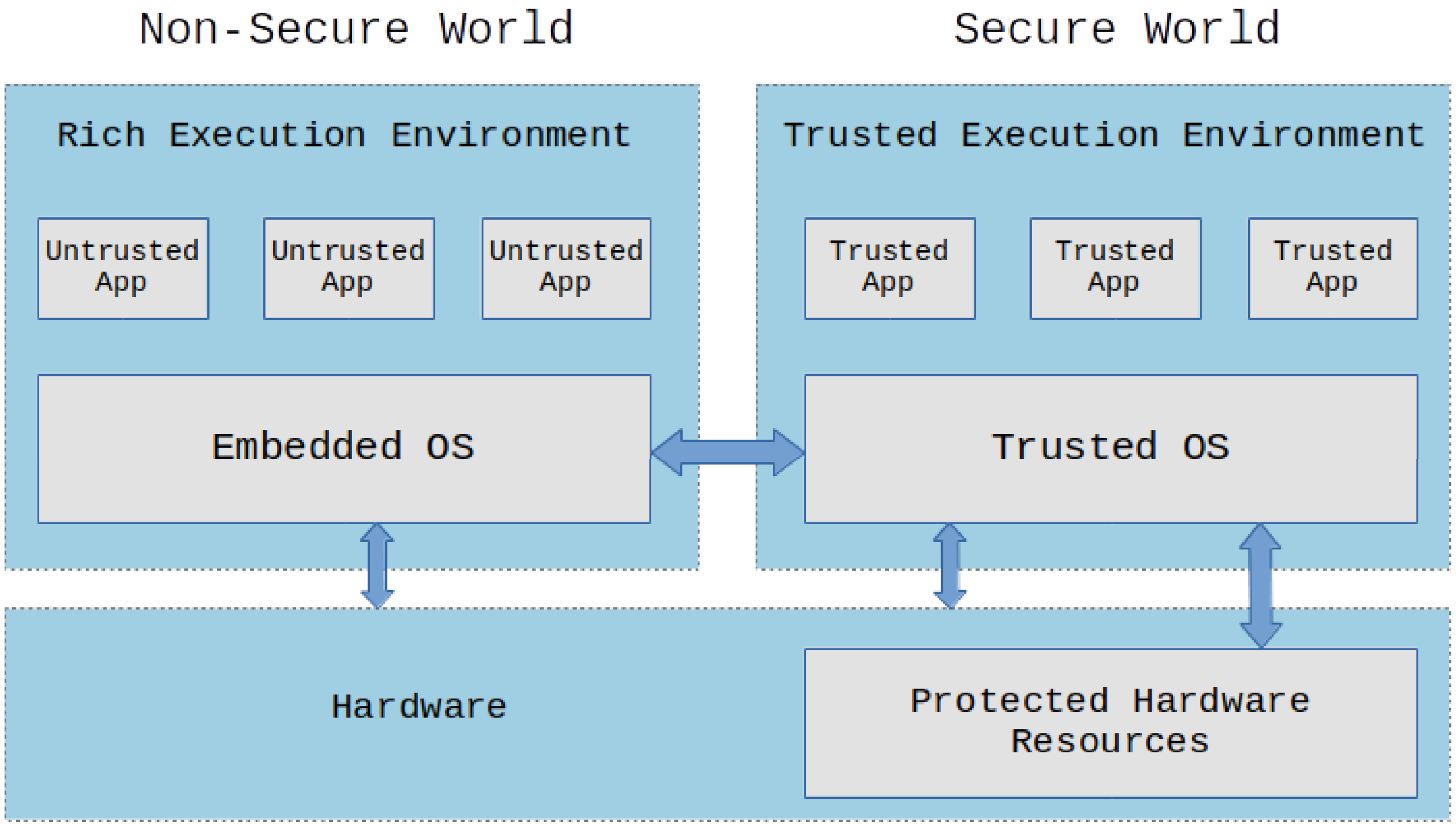
TEE

- In a system with a TEE, we have untrusted applications (UAs) running on a Rich Execution Environment (REE) and trusted applications (TAs) running on a Trusted Execution Environment (TEE).

- Only trusted applications running on a TEE (**Secure World**) have complete access to the main processor, peripherals and memory.

- Hardware isolation protects TAs from untrusted applications running on the main operating system (Non-Secure World).

TEE


Non-Secure World
Secure World
Rich Execution Environment
Trusted Execution Environment
Untrusted App
Untrusted App
Untrusted App
Trusted App
Trusted App
Trusted App
Embedded OS
Trusted OS
Hardware
Protected Hardware Resources

TEE IMPLEMENTATION

- We need hardware support to implement a TEE, so we can partition and isolate the hardware (busses, peripherals, memory regions, interrupts, etc) to prevent untrusted applications from accessing protected resources.
- Most modern processors have this feature build-in (e.g. ARM's TrustZone, RISC-V's MultiZone, Intel SGX).
- There are some commercial TEE implementations, including Kinibi, QSEE and iTrustee.
- We have also some open source implementations like Trusty and OP-TEE.

UPDATE SYSTEM AND SECURITY

- Despite all mitigations we have seen so far, an operating system with millions of lines of code will certainly have bugs and vulnerabilities!
- Having an update system in place is very important for embedded systems and connected devices where security is a key feature of the product.
- The update system should be designed in the early stages of the product development, with OTA features if possible.
- What is more costly: invest time to implement a good update system or recall all units to fix a bug in the software of your product?

UPDATE CHALLENGES

- Security (authenticity, confidentiality).
- Integrity.
- Atomic/power fail safe.
- Bandwidth.
- Speed/downtime.
- Rollback.

UPDATE STRATEGIES

- **Application-based**: not maintainable!
- **Package-based**: update images are small but the updates are non-atomic and dependencies could be a problem.
- **Image-based**: using the A/B mechanism is a very good solution, the problem could be bandwidth and storage size.
- **Container-based**: could be the best of both worlds. Makes it easier to implement an update system that is atomic, power fail safe, use less bandwidth, faster, with minimal downtime and rollback capable.

NETWORK SECURITY

- If you are doing OTA updates, your device has a network connection (Wi-Fi, Ethernet, etc).

- And if your device has a network connection, you should care about network security!

- The first rule is to decrease the attack surface. For example:
  - Close all ports not used/needed (tools like nmap can help).
  - Disable all protocols not used (e.g IPv6, PPP, etc).

NETWORK SECURITY

- Then make it harder to hack your device! For example:
  - Create firewall rules (prevent inbound/outbound connections, protect against DoS attacks, prevent port scanning, etc).
  - Use a network IDS such as **snort** for intrusion protection/detection.
  - Communicate with external devices using a secure connection (VPN, reverse SSH, TLS, HTTPS, etc).
  - Rate limit logins to services (ssh, web, etc) to prevent brute-force attacks.

THERE IS NO SILVER BULLET!

digital signatures
cryptographic
fscrypt
trustzone
secure coding
PKCS#11
secure element
owners
assets
syzkaller
threat agents
CAAM
static code analysis
selinux
ASLR
secure boot
torizon
root-of-trust
VPN
containers
TPM
fuzzing
vulnerabilities
AddressSanitizer
Attack vectors
DREAD
buffer overflow
firewall
SSH
authenticity
encryption key
IDS
ostree
seccomp
apparmor
integer overflow
dm-verity
hash function
CVE
threats
chain-of-trust
risks
device-mapper
dm-crypt
dm-integrity
capabilities
public-key cryptography
STRIDE
eCryptFS
TEE
mandatory access control
threat modeling
symmetric cryptography

SECURITY "GENERAL RULES"

- Security involves all levels of the system.
- Defense in depth: have always more than one layer or type of defense.
- Least privilege principle: do not give any more privileges than absolutely necessary to do the required job.
- Obfuscation or "obscurity" just doesn't work.
- There is no such thing as a system 100% secure.
  - Be aware that an attacker needs only to find one issue!

DESIGN FOR SECURITY

- Design with security in mind and be aware of the trade-offs (a system should be “secure enough”).

- Identify assets, threats, attack vectors and mitigate risks.

- Follow good security practices, know the techniques and tools available and use them when needed.

- Have a good update system, monitor software vulnerabilities (CVEs) and patch the system.

TORIZON

- Torizon is the solution we provide to our customers as a easy-to-use Industrial Linux Software Platform.

- We follow good practices to develop a product based on modern technologies with a good balance between ease-of-use and security.

- The Torizon platform is based on containers, providing the isolation needed for a secure system.

- The Torizon platform also provides a built-in OTA update system that is robust, fast and fail-safe.

Toradex

Swiss. Embedded. Computing.

Q&A

Toradex
Swiss. Embedded. Computing.

THANK YOU
FOR YOUR INTEREST

www.**toradex**.com
developer.toradex.com
community.toradex.com
labs.toradex.com