Navigating security trade-offs in embedded Linux systems

Olivier Benjamin
olivier.benjamin@bootlin.com

Embedded Linux Conference Europe 2025

Olivier Benjamin

- Embedded Linux engineer at **Bootlin**
  - Development, consulting and training about **embedded Linux**
  - Open-source focus
- **Linux kernel** device driver developer
- Bootloaders, Buildroot and Yocto integration
- Open-source contributor
- Living in **Lyon**, France

Embedded systems security

Security mindset

Security is an ever-increasing concern in embedded systems.

- compliance: legislation (CRA), insurance
- reputational risk
- security is part of the features customers are now expecting

"I want the system to be secure"

- Security is not a binary state
- We aim to make it harder for the adversary to compromise the system

The cost of security

- Security measures have a cost:
  - time (e.g. for implementation)
  - dedicated hardware
  - bootup time
  - complexity
- Going for **maximum** security might not be the right call.
- Going for **minimum** security is most likely the wrong one, though.
- Where to place the cursor is our topic.

Threat modeling

What parts of the system should we pay most attention to in order to thwart most of our adversaries?

This depends on:

- the design of the system
- the adversaries we expect
- the constraints we can afford to put on our users
- the level of security we want to achieve

Only some of these factors are technical.

Threat model

A full-fledged threat model is very complex, scaling with the complexity of the system.

Usually means describing your system's assets:

- customer data
- cryptographic material
- intellectual property

Then your system's various boundaries:

- network ports
- physical ports
- privilege levels (Exception Levels, sandboxes, RCE vs LCE, …)

And your adversaries

Security Measures

Security Measures

They are the blockers between your adversaries and your assets, or between different privilege levels.

Any compromise will come from either:

- an unidentified transition
- an unintended use of an identified transition
- a missing security measure

Filesystem encryption

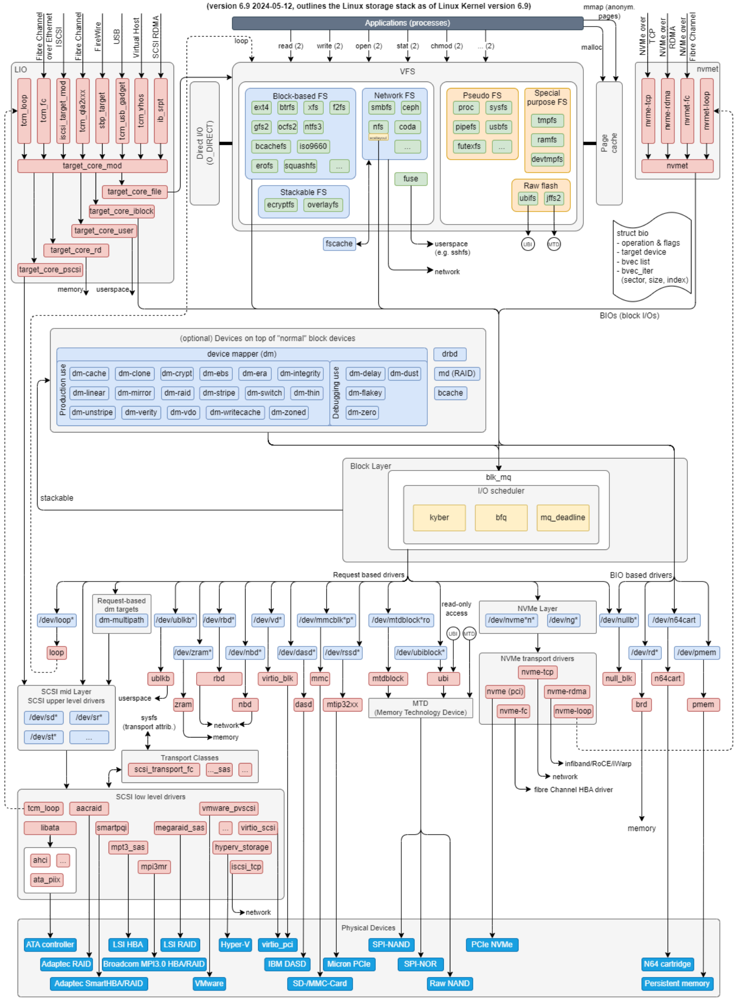
Filesystem encryption

- Linux makes it look like a normal filesystem
- It is never stored unencrypted **on the disk**
- The key is usually either
  - derived from a given password
  - stored encrypted in a header (possibly multiple times) and decrypted at rutime

Filesystem encryption

The Linux Storage Stack Diagram (Linux Kernel 6.9)

(version 6.9 2024-05-12, outlines the Linux storage stack as of Linux Kernel version 6.9)



The Linux Storage Stack Diagram
https://www.thomas-krenn.com/en/wiki/Linux_Storage_Stack_Diagram
Design Werner Fischer, support by Christoph Hellwig, Richard Weinberger et al.

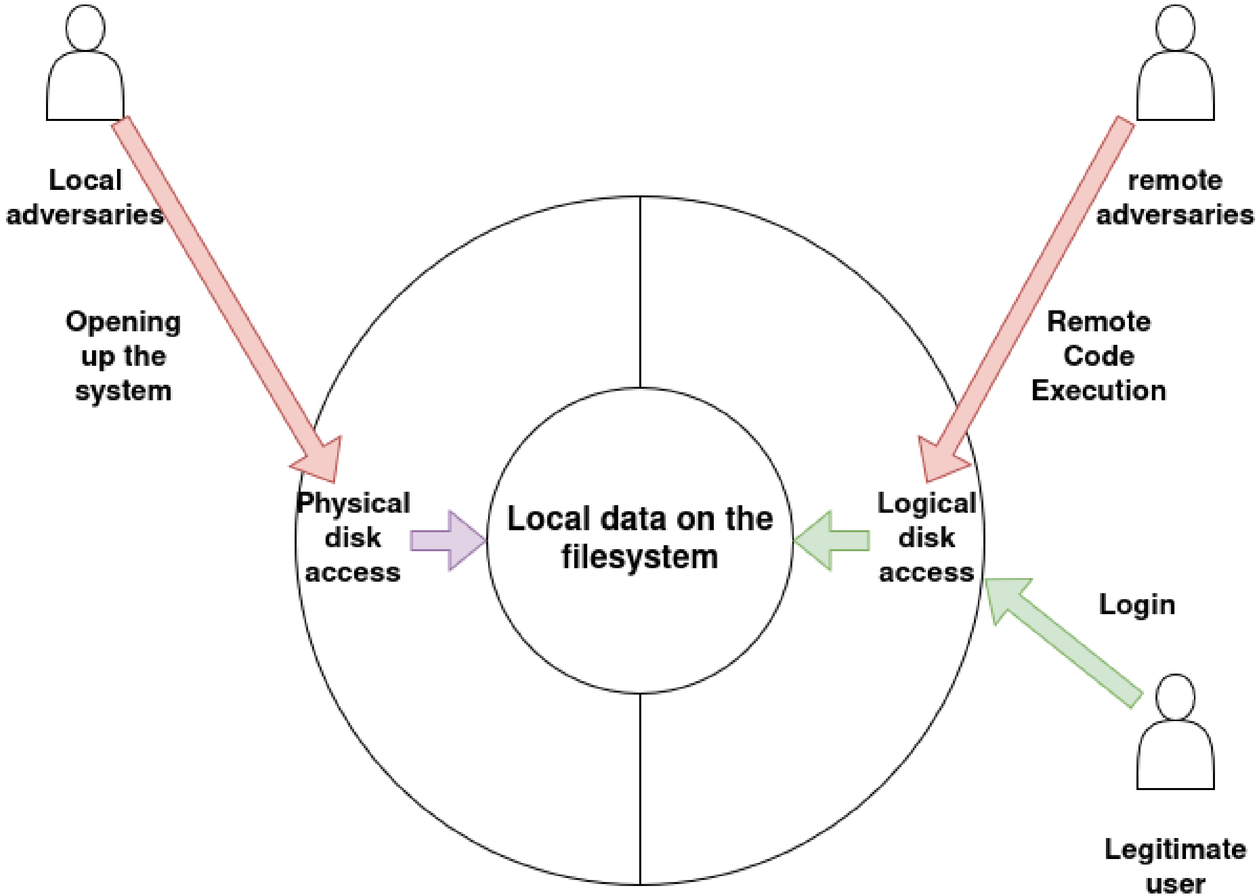
Filesystem encryption

- Will mitigate:
  - read-only offline attack on the hardware (“evil maid”)
- Will **not** mitigate:
  - essentially anything else

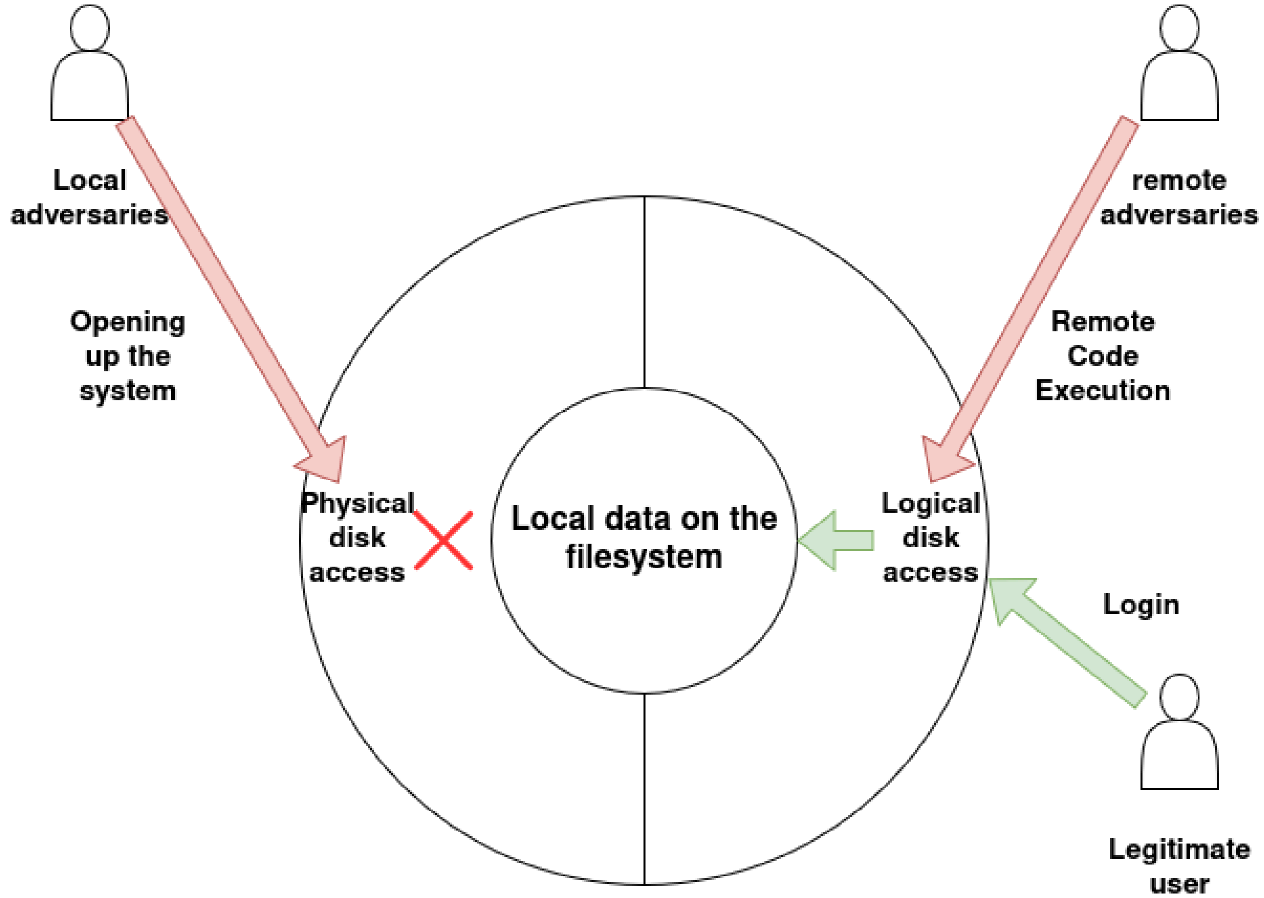
Filesystem encryption: the cost

- small performance overhead
- implementation
- key provisioning & storage
- risk of potential data loss if keys are mismanaged

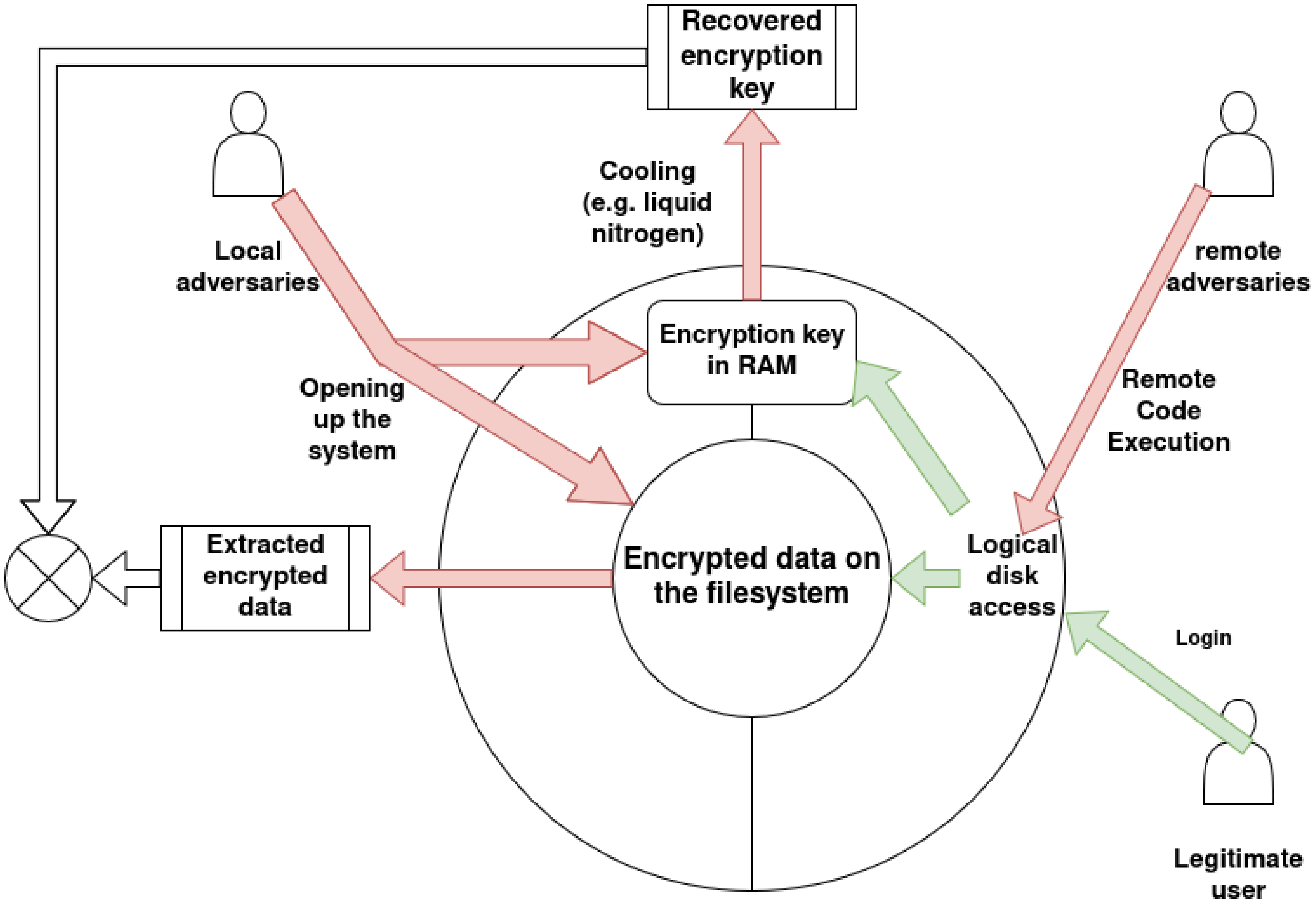
Simple Threat Model: no encryption



Simple Threat Model: encryption



Cold boot attack



Filesystem encryption

Systems that benefit:

- device exposed to untrusted actors without surveillance
- adversarial users (gaming consoles)

Systems that poorly benefit:

- devices not storing user or sensitive manufacturer data (routers for instance)
- devices under a lot of scrutiny: ATMs
- low compute power devices without crypto accelerators

Secure Boot

Secure Boot

- Chain of trusted software
- Root of trust
  - One or multiple hashes of cryptographic material
  - Often embedded in write-once hardware (e.g. fuses)
- Must be implemented in all software up to the kernel:
  - vendor-provided bootROM
  - all bootloader stages

Secure Boot

Will mitigate:

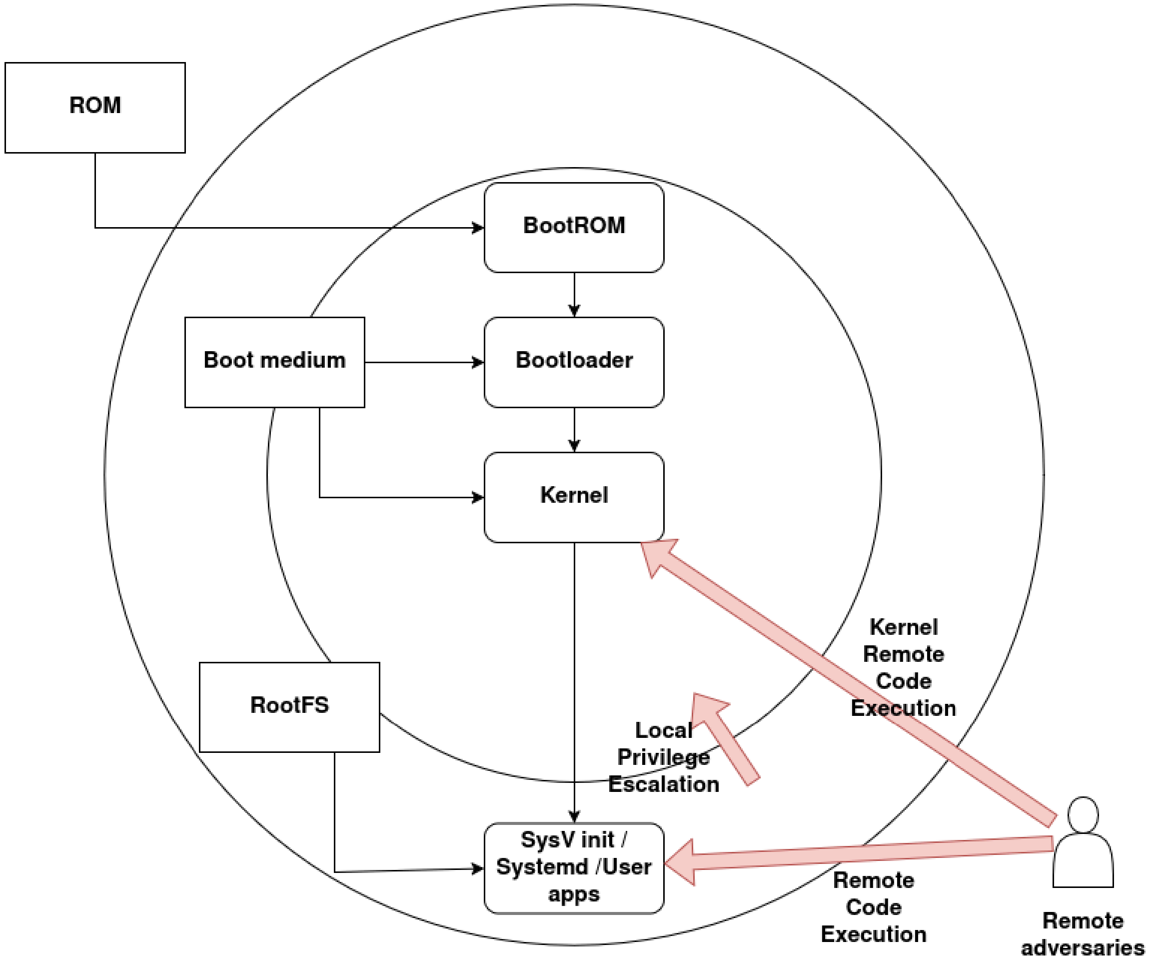
- offline attack from the hardware (“evil maid”)
- attempts at gaining access persistance across reboots/updates

if they target non-userland software

Will **not** mitigate:

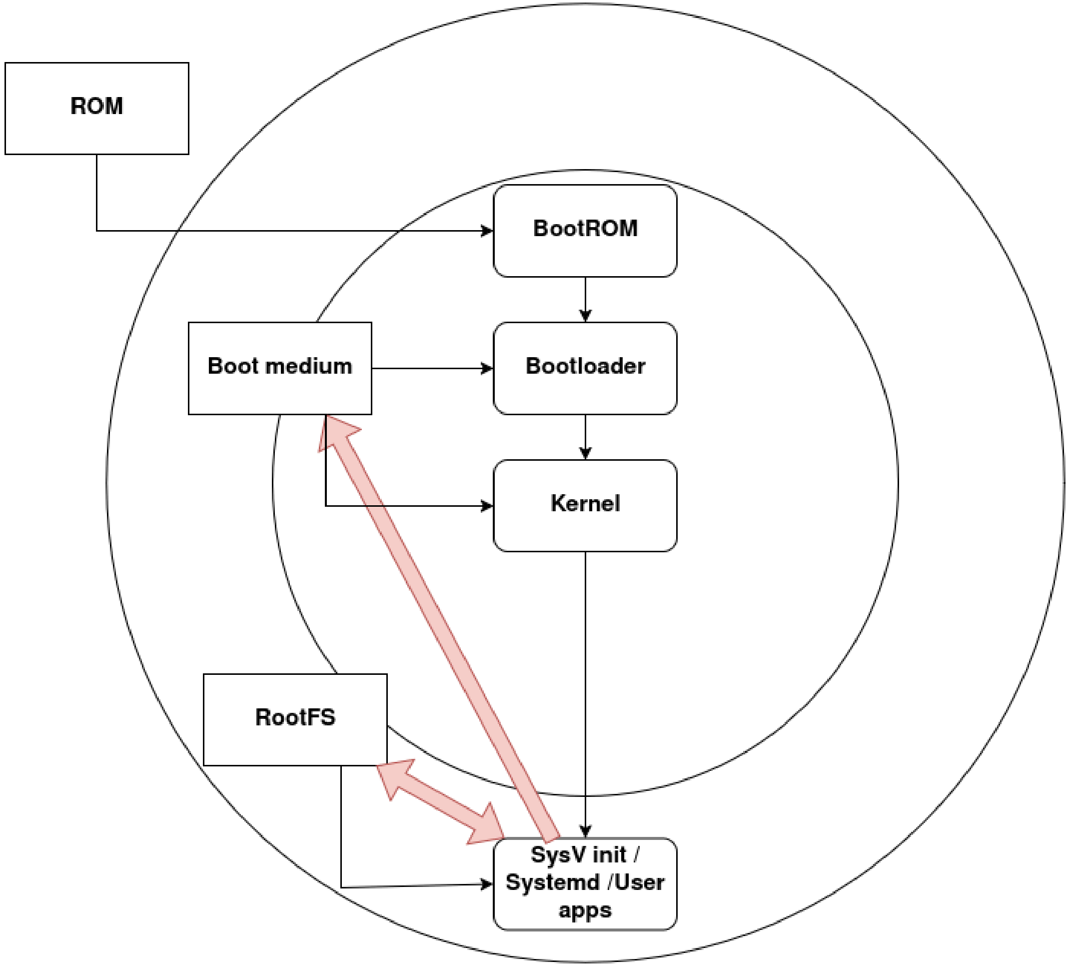
- runtime compromise of the system
- by itself, offline modification of the userland

Normal Boot

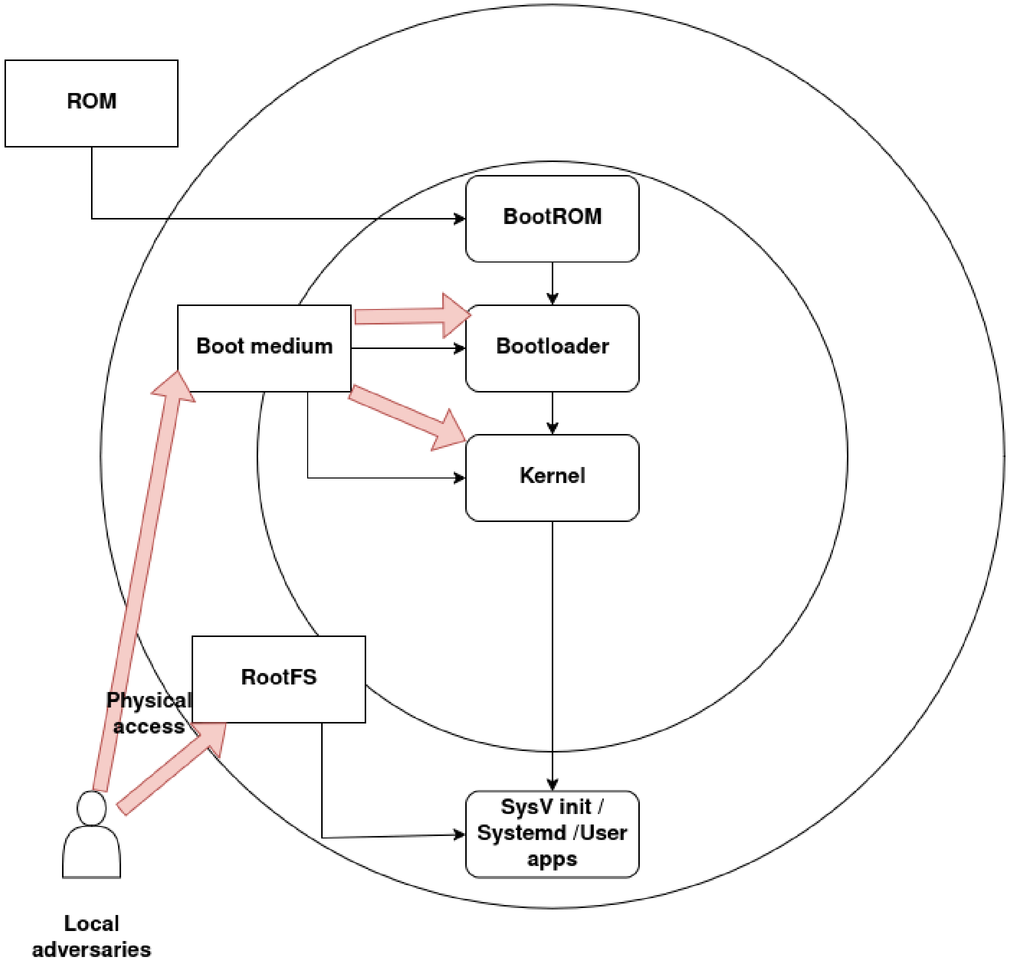


ROM
BootROM
Boot medium
Bootloader
Kernel
RootFS
SysV init / Systemd /User apps
Kernel Remote Code Execution
Local Privilege Escalation
Remote Code Execution
Remote adversaries

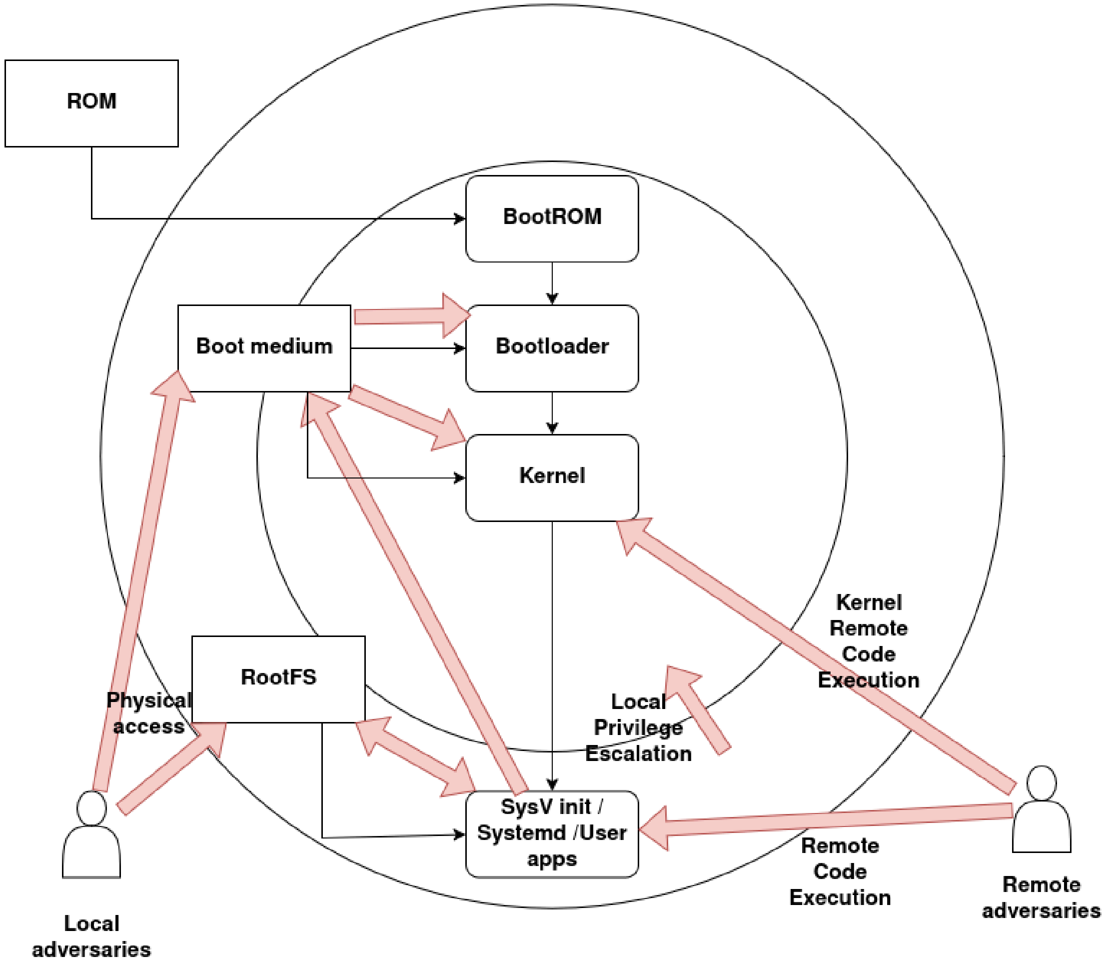
Normal Boot


ROM
BootROM
Boot medium
Bootloader
Kernel
RootFS
SysV init / Systemd /User apps

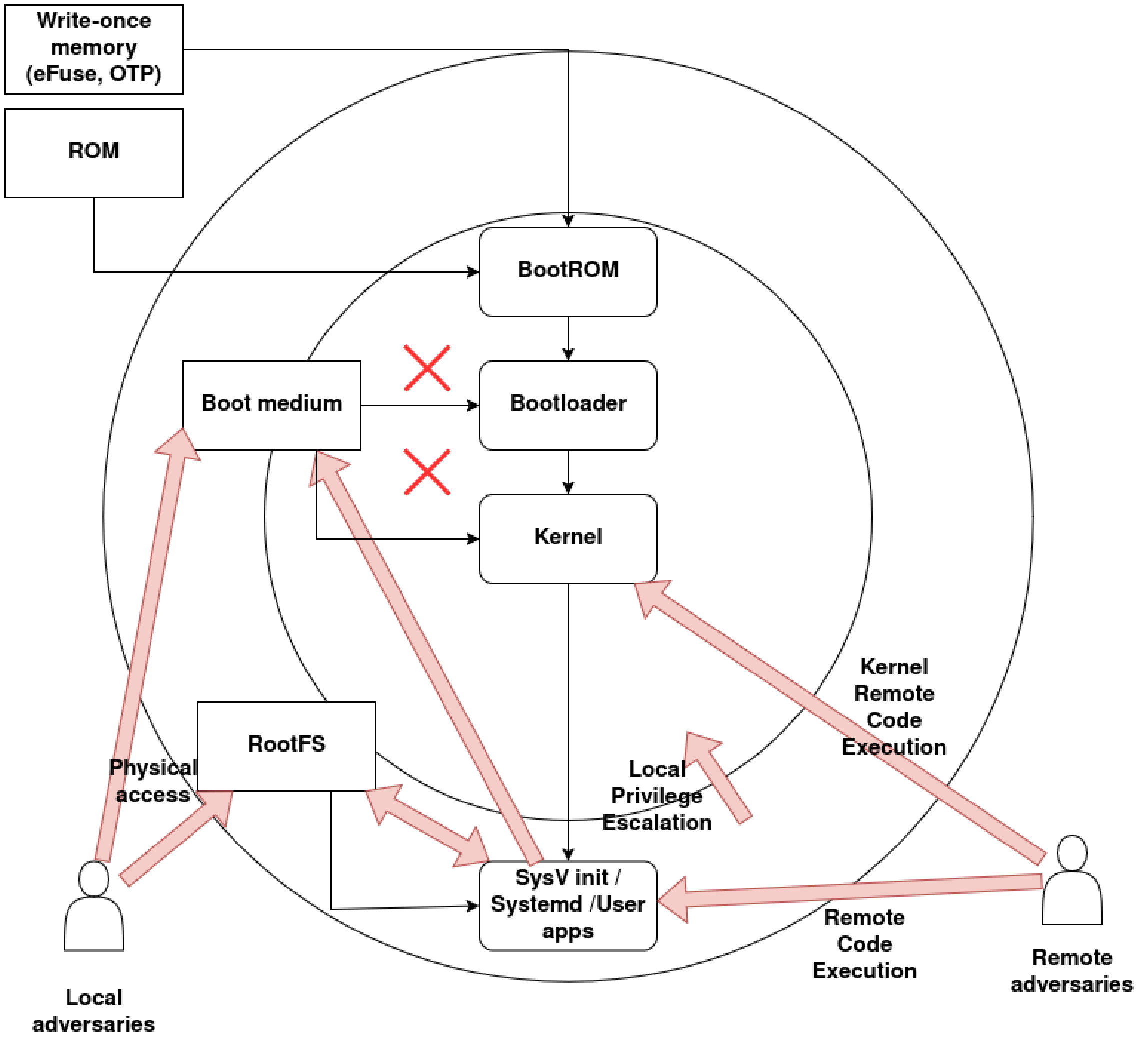
Normal Boot


ROM
BootROM
Boot medium
Bootloader
Kernel
RootFS
Physical access
SysV init / Systemd /User apps
Local adversaries

Normal Boot


ROM
BootROM
Boot medium
Bootloader
Kernel
RootFS
SysV init / Systemd /User apps
Physical access
Local Privilege Escalation
Kernel Remote Code Execution
Remote Code Execution
Local adversaries
Remote adversaries

Secure Boot



Write-once memory (eFuse, OTP)
ROM
BootROM
Boot medium
Bootloader
Kernel
RootFS
SysV init / Systemd /User apps
Physical access
Local Privilege Escalation
Kernel Remote Code Execution
Remote Code Execution
Local adversaries
Remote adversaries

RootFS verification (dm-verity)

RootFS verification (dm-verity)

This is the logical continuation of Secure Boot: how do we guarantee userland has not been altered?

- The idea: generate a hash tree for the entire filesystem
  - That hash tree will be stored on a separate device
  - The root of the tree might be signed
  - Leaves are hashes of a data block
- On accessing any data, the kernel will
  - walk up the tree until it hits either a node that was already verified or the root
  - walk back down, verifying all children nodes on the way

See `fs/verity/verify.c`

RootFS verification (dm-verity)

Will mitigate:

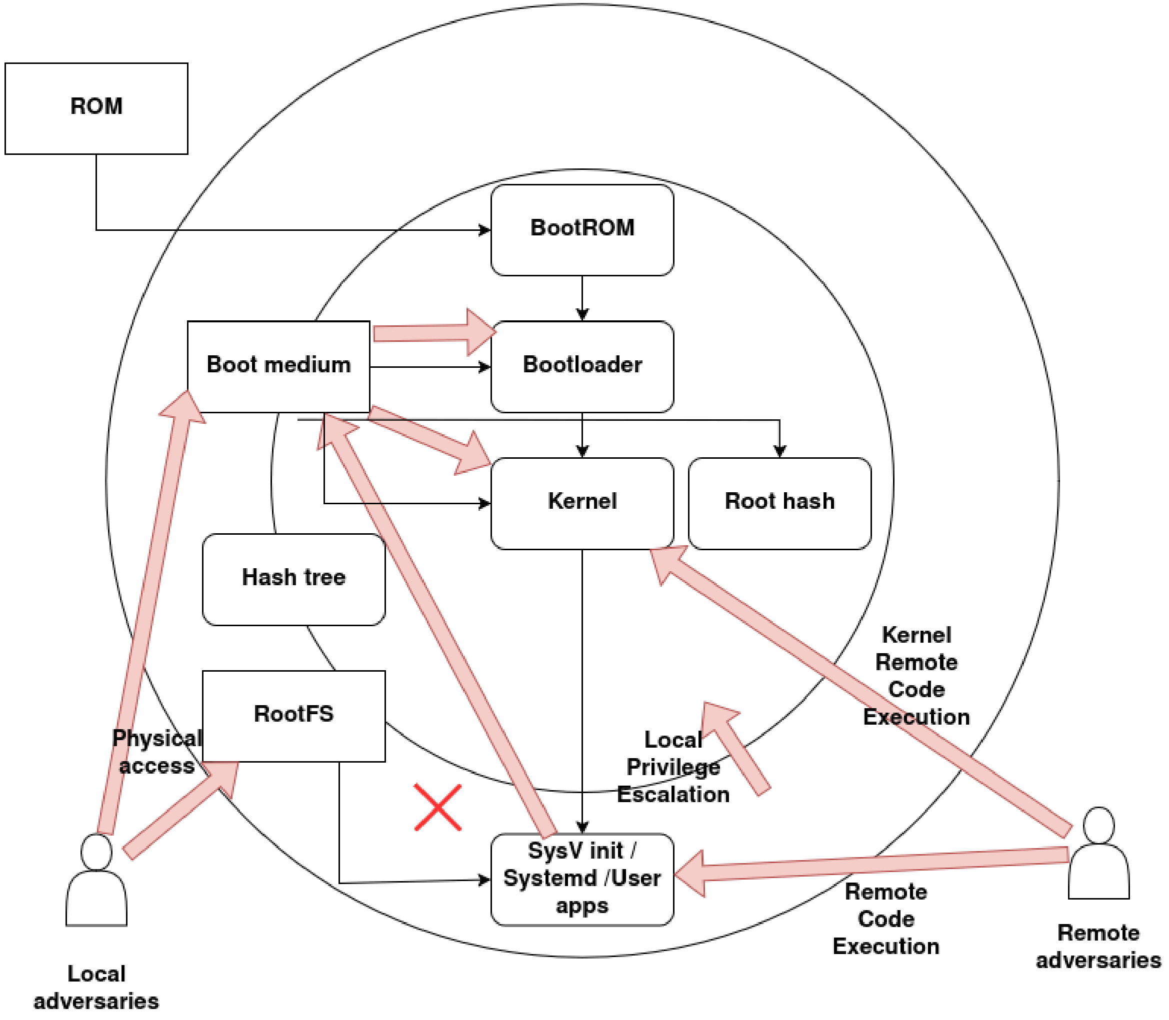
- persistence of userland-only code execution

if combined with a properly implemented Secure Boot:

- gaining userland code execution from physical access

An adversary that gains root privileges will defeat it

RootFS verification (dm-verity)



RootFS verification: pitfalls

The security of the entire scheme hinges on:

- the root hash
- the security of the hash function used (md5 might not be the best choice)
- the integrity of the kernel
- the kernel command line

To be effective, RootFS verification requires a properly implemented Secure Boot:

- Verification of the bootloader, including the kernel command line
- Verification of the kernel, including the co-located root hash

It requires a read-only filesystem.

RootFS verification (dm-verity): the cost

- Making the RootFS read-only
  - using a RO filesystem: EROFS, SquashFS
  - if that's not an option, mounting the rootFS RO
- Makes updates more complex
  - one can no longer update **only** the RootFS: at least the root hash must be updated too
  - if the system has a secure boot chain, that means updating the kernel signature as well
  - if using A/B updates, the bootloader must be able to keep track of the rootFS / root hash association

RootFS verification (dm-verity)

- Systems that benefit:
  - network-connected systems
  - systems where persistence across reboots has an impact
  - systems routinely targeted for botnet enrolment: e.g. SOHO routers, IP cameras
  - systems with a secure boot chain
- Systems that poorly benefit:
  - systems with partial updates (package distributions)
  - systems implementing stored user actions

Secure enclaves

Secure enclaves

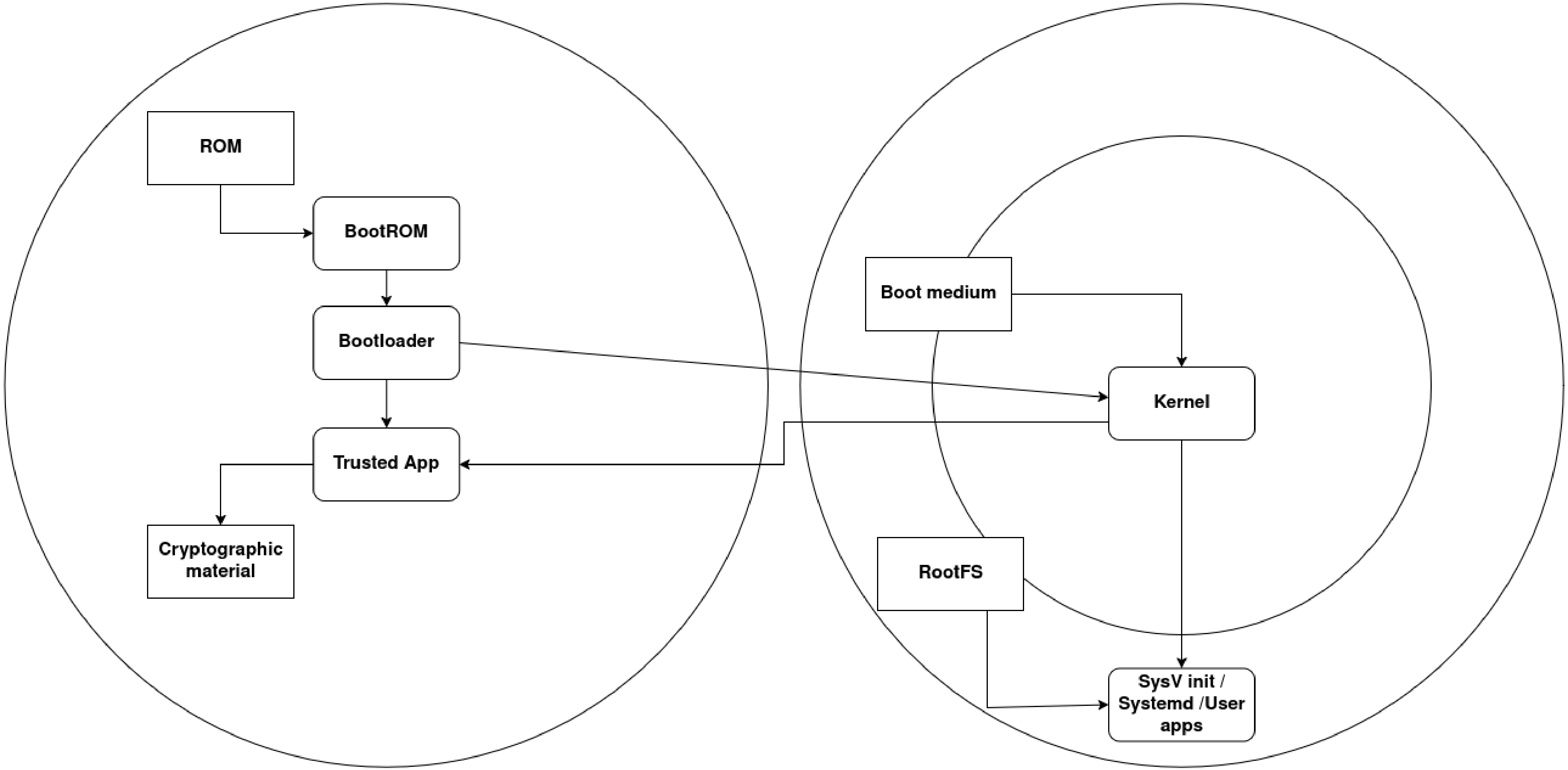
- Hardware-isolated units of computation on the system.
- The main technology for embedded devices is ARM's TrustZone
- Split the system into normal and secure worlds, isolated from each other.
- Essentially requires a further privilege escalation

Secure enclaves

This is useful in a defense in depth approach assuming an adversary with root privileges

- provision any secrets in Secure world (e.g. by reading memory only accessible in Secure world)
- only use those secrets within the Secure world
- offer an interface to the normal world OS

Secure enclaves



Secure enclaves: pitfalls

The Secure world is less versatile than the OS

- Development in secure world is harder

Secure enclaves are only an additional isolation mechanism

- Necessitates accrued collaboration from HW
- Trusted Applications can have vulnerabilities too
  - arbitrary code execution in Samsung's TEEGRIS
  - buffer overflow in a Trusted App in Qualcomm's QSEE
- Secure enclaves require **more** scrutiny to be effective

Overall, they are a significant increase in design, development and maintenance costs.

Secure enclaves

Will mitigate:

- Exfiltration of data/logic from the machine without physical access
- Modification of data/logic on the machine without physical access

Will not mitigate:

- Use of the data/logic by an adversary running on the machine

Secure enclaves

Systems that benefit:

- systems with global crypto secrets
- systems wanting to tie a secret to a physical machine (e.g. Licenses)
- systems part of large families, with long-term support
- systems shortly handling small sensitive info (voting machines, biometrics)
- adversarial users

Systems that poorly benefit:

- systems without a very security-aware userbase

Intrusion Detection System (IDS)

- OSSEC

Thank you!

Questions?

Olivier Benjamin

olivier.benjamin@bootlin.com

https://bootlin.com/pub/conferences/