Embedded Linux Security training

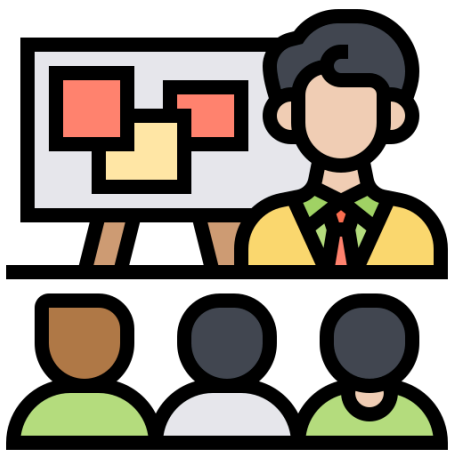
Latest update: June 18, 2026.

Document updates and training details:
https://bootlin.com/training/security

Corrections, suggestions, contributions and translations are welcome!
Send them to feedback@bootlin.com

Embedded Linux Security training

- These slides are the training materials for Bootlin's *Embedded Linux Security* training course.
- If you are interested in following this course with an experienced Bootlin trainer, we offer:
  - **Public online sessions**, opened to individual registration. Dates announced on our site, registration directly online.
  - **Dedicated online sessions**, organized for a team of engineers from the same company at a date/time chosen by our customer.
  - **Dedicated on-site sessions**, organized for a team of engineers from the same company, we send a Bootlin trainer on-site to deliver the training.
- Details and registrations: https://bootlin.com/training/security
- Contact: `training@bootlin.com`



Icon by Eucalyp, Flaticon

About Bootlin

Bootlin introduction

- Engineering company
  - In business since 2004
  - Before 2018: *Free Electrons*
- Team based in France and Italy
- Serving **customers worldwide**
- **Highly focused and recognized expertise**
  - Embedded Linux
  - Linux kernel
  - Embedded Linux build systems
- **Strong open-source** contributor
- Activities
  - **Engineering** services
  - **Training** courses
- https://bootlin.com

bootlin

Bootlin engineering services

Bootloader / firmware development

U-Boot, Barebox, OP-TEE, TF-A, .../

Linux kernel porting and driver development

Linux BSP development, maintenance and upgrade

Embedded Linux build systems

Yocto, OpenEmbedded, Buildroot, ...

Embedded Linux integration

Boot time, real-time, security, multimedia, networking

Open-source upstreaming

Get code integrated in upstream Linux, U-Boot, Yocto, Buildroot, ...

Bootlin training courses

Embedded Linux system development On-site: 4 or 5 days Online: 7 * 4 hours	Linux kernel driver development On-site: 5 days Online: 7 * 4 hours	Yocto Project system development On-site: 3 days Online: 4 * 4 hours	Buildroot system development On-site: 3 days Online: 5 * 4 hours	Embedded Linux networking On-site: 3 days Online: 4 * 4 hours
Understanding the Linux graphics stack On-site: 2 days Online: 4 * 4 hours	Embedded Linux audio On-site: 2 days Online: 4 * 4 hours	Embedded Linux Security On-site: 3 days Online: 4 * 4 hours	Linux debugging, tracing, profiling and performance analysis On-site: 3 days Online: 4 * 4 hours	Real-Time Linux with PREEMPT_RT On-site: 2 days Online: 3 * 4 hours

Bootlin, an open-source contributor

- Strong contributor to the **Linux** kernel
  - In the top 30 of companies contributing to Linux worldwide
  - Contributions in most areas related to hardware support
  - Several engineers maintainers of subsystems/platforms
  - 9000 patches contributed
  - https://bootlin.com/community/contributions/kernel-contributions/
- Contributor to **Yocto Project**
  - Maintainer of the official documentation
  - Core participant to the QA effort
- Contributor to **Buildroot**
  - Co-maintainer
  - 6000 patches contributed
- Significant contributions to U-Boot, OP-TEE, Barebox, etc.
- Fully **open-source training materials**

Bootlin on-line resources

- Website with a technical blog:
  https://bootlin.com
- Engineering services:
  https://bootlin.com/engineering
- Training services:
  https://bootlin.com/training
- LinkedIn:
  https://www.linkedin.com/company/bootlin
- Elixir - browse Linux kernel sources on-line:
  https://elixir.bootlin.com

Icon by Freepik, Flaticon

Generic course information

Corrections, suggestions, contributions and translations are welcome!

Training quiz and certificate

- To get your training certificate you must
  1. Attend all sessions of this training course
  2. Achieve more than 50% of correct answers at our final quiz
     - The final quiz questions are identical to the pre-training quiz
     - The final quiz must be completed within two weeks of the session end's date
- The training certificate will be sent to you two weeks after the session end's date.

Participate!

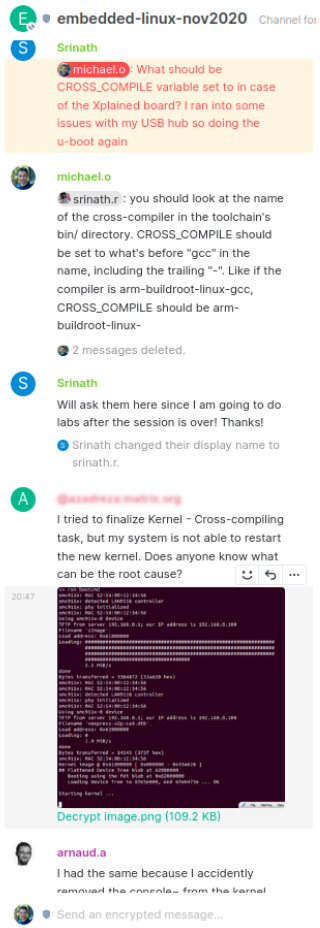
During the lectures...

- Don't hesitate to ask questions. Other people in the audience may have similar questions too.
- Don't hesitate to share your experience too, for example to compare Linux with other operating systems you know.
- Your point of view is most valuable, because it can be similar to your colleagues' and different from the trainer's.
- In on-line sessions
  - Please always keep your camera on!
  - Also make sure your name is properly filled.
  - You can also use the "Raise your hand" button when you wish to ask a question but don't want to interrupt.
- All this helps the trainer to engage with participants, see when something needs clarifying and make the session more interactive, enjoyable and useful for everyone.

Collaborate!

As in the Free Software and Open Source community, collaboration between participants is valuable in this training session:

- Use the dedicated Matrix channel for this session to add questions.
- If your session offers practical labs, you can also report issues, share screenshots and command output there.
- Don't hesitate to share your own answers and to help others especially when the trainer is unavailable.
- The Matrix channel is also a good place to ask questions outside of training hours, and after the course is over.



IMX93 FRDM shopping list

- NXP i.MX93 11x11 FRDM board (76 EUR + VAT on Mouser)
  - NXP i.MX93 (Dual ARM Cortex-A55 + Cortex-M33)
  - 2 GB LPDDR4
  - 32 GB of on-board eMMC storage
  - Plenty of peripherals: I2C, SPI, UART, USB...
- 2 USB-C cable for the power supply and the serial console
- RJ45 cable for networking

Fundamental concepts

Threat modeling

What is threat modeling?

- Threat modeling is the process of listing and organizing potential threats and appropriate responses
- Helps to:
  - Identify potential threats
  - Prioritize them based on the risk
  - Identify possible countermeasures
- Provides an analysis of what affects the security and which measures need to be implemented
- A large number of frameworks and methodologies exist.

Security properties: CIA triad

- **Confidentiality**: Can an unauthorized entity gain access to information?
- **Integrity**: Can there be unauthorized modification of information?
- **Availability**: Can authorized access to information be impeded?

Confidentiality

- The main property one wants out of a secure information system
- Typical adversary: **passive** MITM
- Protecting it is **encryption**'s main role
- Can make it harder to ensure **Availability**

Integrity

- Aims to ensure that information is not modified
- Typical adversary is an **active** MITM
- The most basic form of protection are checksums

Availability

- Aims to ensure that information is accessible
- Typical adversaries are DoS attacks
- Defense usually involves:
  - early detection of threats
  - redundancy and failover
- Maintaining availability can lead to compromising on confidentiality, this is what some scams rely on.

In cybersecurity: the five pillars

- **Confidentiality**: Can an unauthorized entity gain access to information?
- **Integrity**: Can there be unauthorized modification of information?
- **Availability**: Can authorized access to information be impeded?
- **Authenticity**: Can an unauthorized entity insert undistinguishable information?
- **Non-repudiation**: Can an authorized entity deny some information's authenticity?

The **Parkerian hexad** introduces

- **Utility**: Is the information useful?

and trades **Non-repudiation** for

- **Control**: Can authorized users access information?

Authenticity

- Aims to ensure that information (e.g.) a message came from a source
- This is what digital signatures aim to protect
- This is not the same as integrity:
  - No protection against **replaying** a signed message

Non-repudiation

- Ensures that actions or messages cannot be disavowed after the fact
- Properly implemented digital signature schemes can protect it
- Precise context must be included in the signature to be effective

Threat modeling frameworks: STRIDE

Flip side of the security properties:

- **Spoofing**: Unauthorized use of credentials
- **Tampering**: Unauthorized modification of information
- **Repudiation**: Performing unauthorized actions that cannot be detected
- **Information disclosure**: Unauthorized access to information
- **Denial of Service**: Disruption of authorized access to a resource
- **Elevation of privilege**: Execution of unauthorized actions

Threat modeling frameworks: Attack tree

- Popularized by Bruce Schneier
- Start with a goal, e.g. “run a program as root”, this is your root node
- Children are ways to achieve this goal
  - recover root password
  - elevate local user privilege
  - make a sudoer run your program as root
  - replace a setuid binary on the filesystem
- Children can then have children themselves
- Attribute a cost to leaves, the cost of the parent is the min

Threat modeling frameworks: LINDDUN

- Privacy-focused threat model
- Developed by researchers at KU Leuven
- Suitable for GDPR/HIPAA
  - **Linkability** Can an adversary link actions or data to a person?
  - **Identifiability** Can an adversary leak a person's identity?
  - **Non-repudiation** Can an adversary attribute a claim to an person?
  - **Detectability** Can an adversary detect a person's involvement?
  - **Disclosure of information** Can an adversary access personal data?
  - **Unawareness** Do persons know how their data is being processed?
  - **Non-compliance** Does the system comply with standards and regulation?

Threat modeling frameworks: PASTA

- Process for Attack Simulation and Threat Analysis
- Rather complex
- Works in 7 stages:
  - **Definition of objectives**
  - **Definition of technical scope**
  - **System decomposition**
  - **Threat analysis**
  - **Vulnerability analysis**
  - **Attack modeling**
  - **Impact analysis**

Which methodology to choose?

- It is not necessary to follow one
- Any Threat Model is better than no Threat Model
- These frameworks are mnemonic devices:
  - A Threat Model helps to be exhaustive
  - The one forgotten threat might be the one that is exploited
  - But they also help define scope
- Try one, and adapt it to your needs

Cryptography basics

Cryptography basics

- Various types of algorithms can be used to secure sensitive data
  - Cryptographic hash functions produce a message digest
  - Encryption algorithms allow the transformation of plain-text data into unintelligible data
    - Symmetric cryptography uses the same key to encrypt and decrypt data
    - Asymmetric cryptography uses a different key to encrypt and decrypt data
- Real world protocols will combine different algorithms from several types

Cryptographic hash function

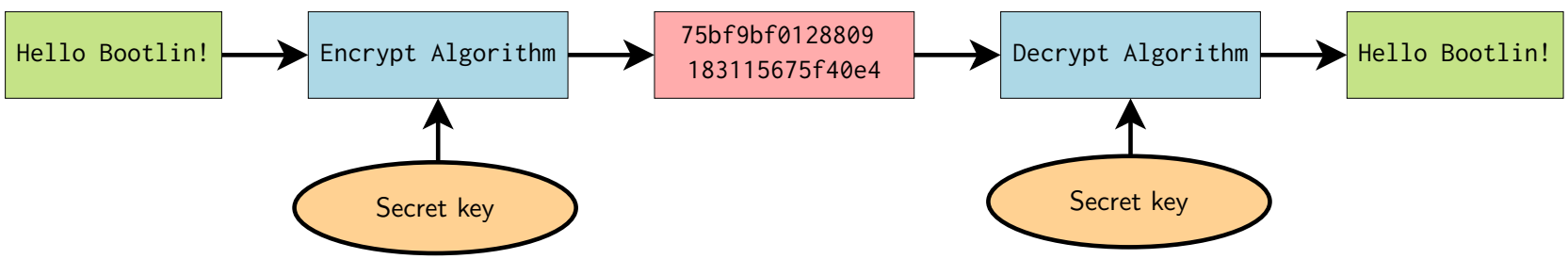
Cryptographic hash function

- Hash functions map data of arbitrary size to a known-size digest
- Cryptographic hash functions are suited for cryptographic operations:
  - Generally used for message authentication, digital signatures or password hashing
- Specific requirements on cryptographic hash functions:
  - Finding an input message that generates a given hash is unfeasible
  - For random input data, each possible hash is equally probable
  - Avalanche effect: small changes on the input produce a completely different output
- Popular algorithms: MD5, SHA-1, Whirlpool, SHA-2, SHA-3 (Keccak).
- SHA-2 and SHA-3 are the most suited for new products
  - TLS 1.3 mostly uses SHA-256 and SHA-384

Symmetric encryption

Symmetric encryption

- Simplest form of encryption
- Encryption and decryption algorithms use the same key
- Encryption and decryption algorithm may differ
- Historical algorithms: Caesar cipher, Enigma machine, ROT13, XOR
- Previously popular algorithms: DES, RC4, Blowfish, and Twofish
- Nowadays, AES is the most widespread algorithm

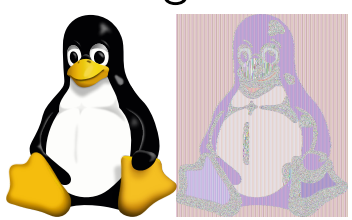


AES: Advanced Encryption Standard

- NIST specification from 2001
- Based on Rijndael algorithm, developed by Joan Daemen and Vincent Rijmen
- Block sizes 128 bits
- Key sizes of 128, 192, or 256 bits
- Used in a virtually all up-to-date protocols relying on symmetric encryption
- Low RAM and CPU requirements, can easily be hardware accelerated

Block Cipher Modes of Operation

- Most symmetric encryption algorithms operate on blocks of a fixed size
  - To encrypt longer data, the data must first be split in as many blocks as needed
  - When encrypting multiple blocks with the same key, some randomness must be introduced, otherwise input patterns might be identifiable in the output.



An image and its encryption in ECB mode

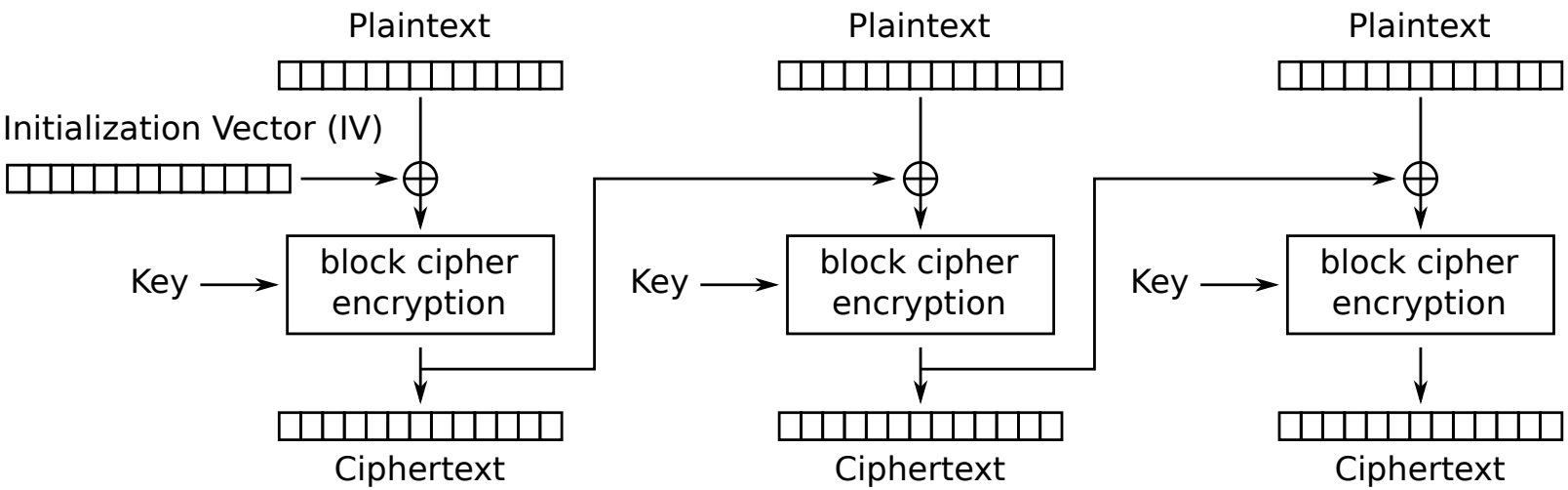
Larry Ewing, Simon Budig, Garrett LeSage, CC0: https://en.wikipedia.org/wiki/File:Tux.svg

RFL890, CC0: https://en.wikipedia.org/wiki/File:Tux_encrypted_ecb.png

- Additionally, we often want protection from data modification by third parties and replay attacks
- Modes of operation allow to securely chain blocks together

Block Cipher Modes of Operation Examples (1)

- Modes ensuring data confidentiality:
  - CBC: Cipher block chaining
    - Input of each block is XORed with the output of the previous block
    - The first plaintext block is XORed with an initialization vector either deterministic or random, shared along the encrypted data.

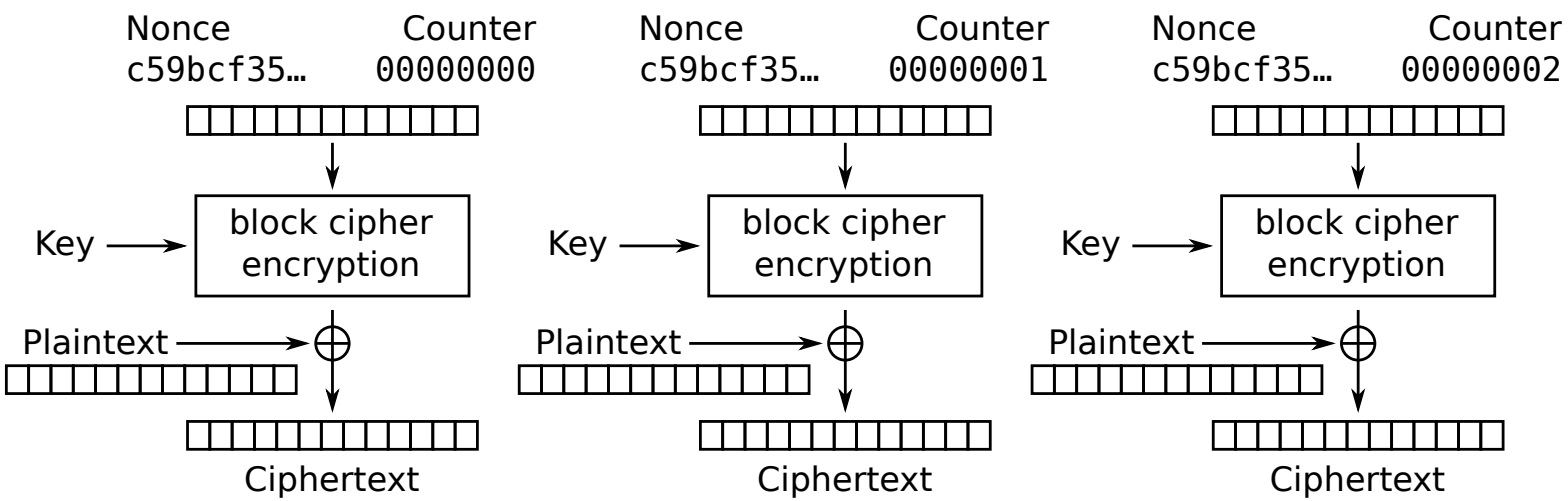


Cipher Block Chaining (CBC) mode encryption

WhiteTimberwolf, Public domain:
https://commons.wikimedia.org/wiki/File:CBC_encryption.svg

Block Cipher Modes of Operation Examples (2)

- Modes ensuring data confidentiality:
  - CTR: Counter
    - An incrementing value is encrypted, then XORed with the input block to generate encrypted data
    - No dependency between blocks: encryption can be parallelized, random blocks can be modified
    - The counter is initially derived from a nonce, which should be unique per message-key combination.

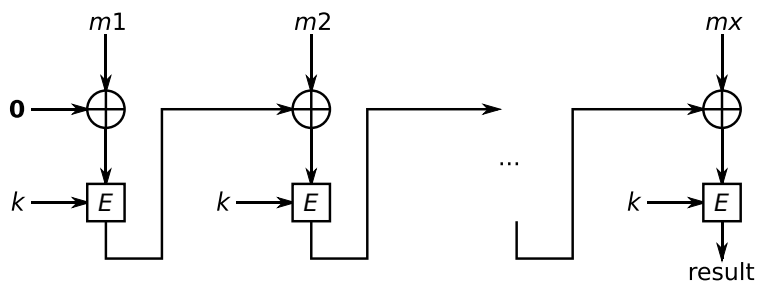


Counter (CTR) mode encryption

WhiteTimberwolf, Public domain:
https://commons.wikimedia.org/wiki/File:CTR_encryption_2.svg

Block Cipher Modes of Operation Examples (3)

- Modes ensuring data authentication:
  - CBC-MAC: Cipher block chaining message authentication code
    - Reuses the CBC mechanism
    - All blocks are chained: the output of the last block depends on all preceding blocks, the key, and the initialization vector
    - This last block output is the CBC-MAC value
    - CBC-MAC can be sent with the message, allowing receiver to verify data integrity



Benjamin D. Esham, Public domain: https://commons.wikimedia.org/wiki/File:CBC-MAC_structure_(en).svg

  - In most situations, keys should not be reused between confidentiality and authentication modes, at the risk of leaking part of it.

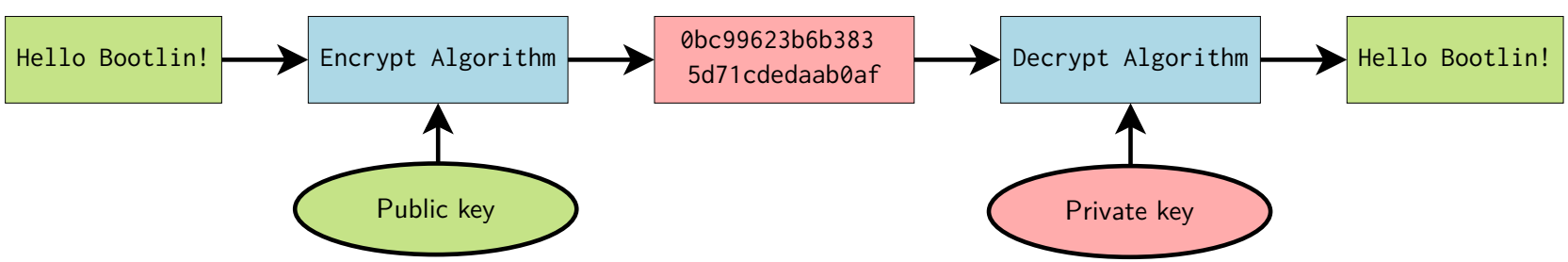
Block Cipher Modes of Operation Examples (4)

- Modes ensuring data authentication and confidentiality:
  - CCM: Counter with cipher block chaining message authentication code
    - Relies on both CTR and CBC-MAC
    - Allows both encryption and authentication in one operation, with a single key and a single nonce.
  - GCM: Galois/Counter Mode
    - Relies on both CTR and Galois field multiplication
    - Also allows both encryption and authentication in one operation, with a single key and a single initialization vector.

Asymmetric cryptography

Asymmetric cryptography

- Encryption and decryption use a pair of distinct but related keys
- Relies on one-way mathematical functions
- A public key is used to encrypt data, a private key is used to decrypt them
- Can also be used to authenticate data:
  - A message signature can be generated using a data digest and the private key
  - The data can be verified using this signature and the public key by comparing the obtained digest with the data
  - If both digests match, we know the signature emitter had access to the private key
- This allows two parties to communicate without first sharing a secret key
- Popular encryption algorithms: RSA
- Popular signature algorithms: ECDSA, EdDSA



RSA

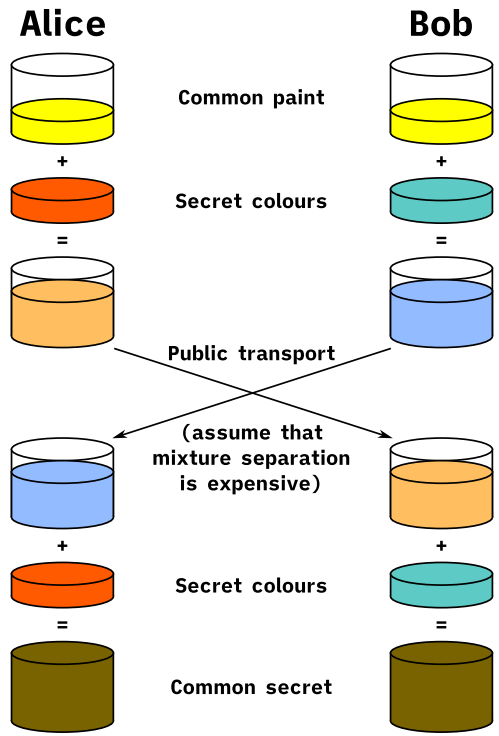
- Created by Ron Rivest, Adi Shamir, Leonard Adleman in 1977
- Variable key size, typically 3072 or 4096 bits for new products
- Relies on the difficulty to factorize the product of two prime numbers
- Can be used both for encryption and signature generation
- Slower than symmetric encryption
- Hardware implementation is complex but possible

ECDSA, EdDSA

- Elliptic Curve Cryptography is based on another mathematical concept
  - Allows smaller keys with similar security level
  - Needs a bit less CPU and memory resources
- Elliptic Curve Digital Signature Algorithm
  - Designed in 1999
  - A variant of the DSA algorithm that uses elliptic-curve cryptography
  - Typical key size of 256 or 384 bits
  - Needs a randomly generated nonce
- Edwards-curve Digital Signature Algorithm
  - Designed in 2011
  - Based on twisted Edwards curves, a family of elliptic curves
  - Ed25519 relies on SHA-512, 256-bit keys
  - Ed448 relies on SHAKE256, 456 bits keys
  - Use a deterministically generated nonce

Diffie-Hellman

- Published by Whitfield Diffie and Martin Hellman in 1976
- A key exchange algorithm that allows two parties to jointly generate a key
  - Both parties will generate part of the key
  - A third party eavesdropping during the key generation would not be able to recreate this key
- Provides forward secrecy
  - Communication sessions remain secret over the long term, even if a long-term secret key is leaked



Diffie-Hellman key exchange analogy

A.J. Vinck, Public domain: https://en.wikipedia.org/wiki/File:Diffie-Hellman_Key_Exchange.svg

Real-world use cases

Symmetric vs asymmetric cryptographic algorithms

- Symmetric and asymmetric cryptographic algorithms tend to have opposite advantages:
  - Symmetric algorithms:
    - Low RAM and CPU requirements
    - Can easily be hardware accelerated
    - Requires a previously and secretly exchanged key between each pair of peers
  - Asymmetric algorithms:
    - Hardware resources hungry
    - Each peer needs to publish only one public key.
- In practice, most protocols will rely on a combination of symmetric and asymmetric algorithms.

Typical use cases

- Some use cases will only need to sign data:
  - RSA, ECDSA, or EdDSA can be used
  - Typical example: secure boot
- Some use cases will only need to encrypt data:
  - AES can be used with an appropriate block cipher mode of operation
  - Typical example: disk encryption
- Secure communication protocol will use a mix of all of these:
  - RSA, ECDSA, or EdDSA for authentication
  - DH of ECDH for key exchange
  - AES for communication once the secure connection is established
  - Typical protocols: TLS, SSH, GPG...

Understanding a TLS handshake

The Transport Layer Security Protocol

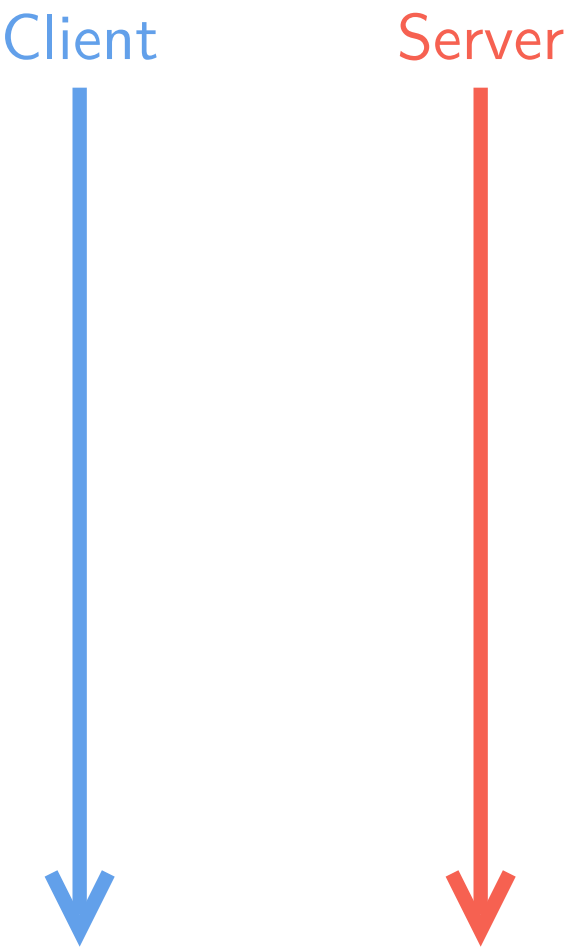
- First proposed in 1999 by the IETF as the Secure Sockets Layer (SSL) protocol
- Allows servers and clients to communicate in a secure way:
  - Authenticate the peers, either server-only or both ends
  - Encrypt connection to prevent eavesdropping
- It relies on several cryptographic mechanisms, including public-key and symmetric encryption.
- Relies on public key certificate to verify the remote public key

Understanding the TLS protocol

- TLS protocol is based on *records* containing either protocol management data or application data.
- Two main phases:
  - A handshake phase:
    - The algorithms to use during the session are selected
    - Peers are authenticated
    - Session keys are exchanged
  - A data exchange or "Application" phase
- This is a highly simplified view: application records can be interleaved with connection management records, such as key renegotiation or alerts.

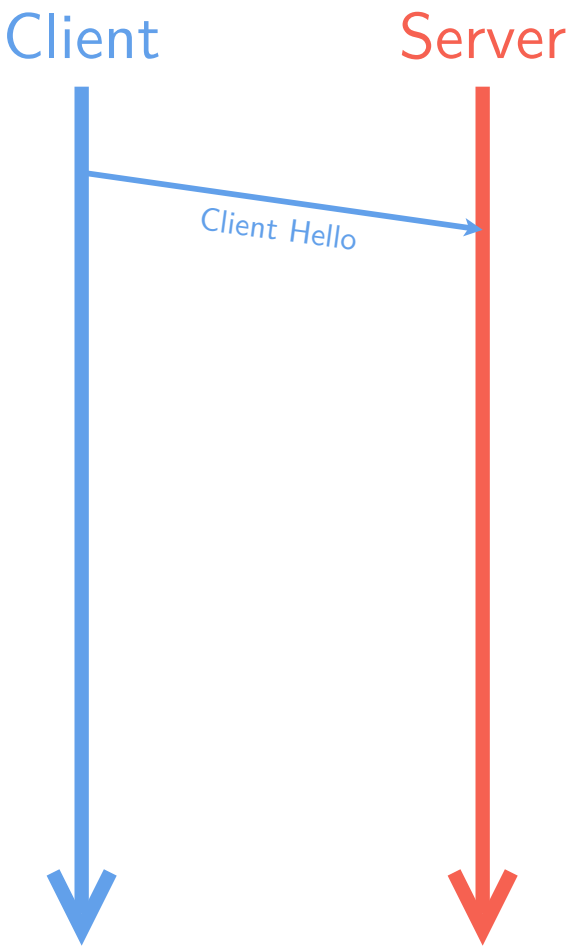
TLS handshake with server authentication only

- Assumptions
  - Only the server is authenticated
  - This is the first time the peers connect
  - TLS version 1.3: https://www.rfc-editor.org/rfc/rfc8446



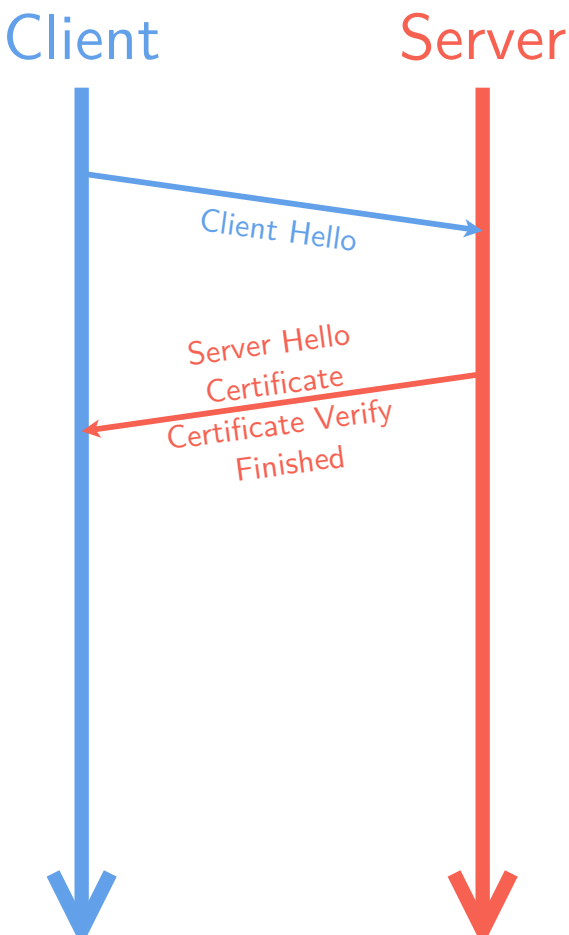
TLS handshake with server authentication only

- Assumptions
- Client: Client Hello
  - Very first message, sent by the client willing to open the connection
  - Contains:
    - The client protocol version
    - A 32 bytes random number
    - The list of supported symmetric cipher suites (e.g., `AES_128_CCM`, `CHACHA20_POLY1305`)
    - A list of supported mechanisms for key exchange and their associated values (e.g., Diffie-Hellman with client public parameters).



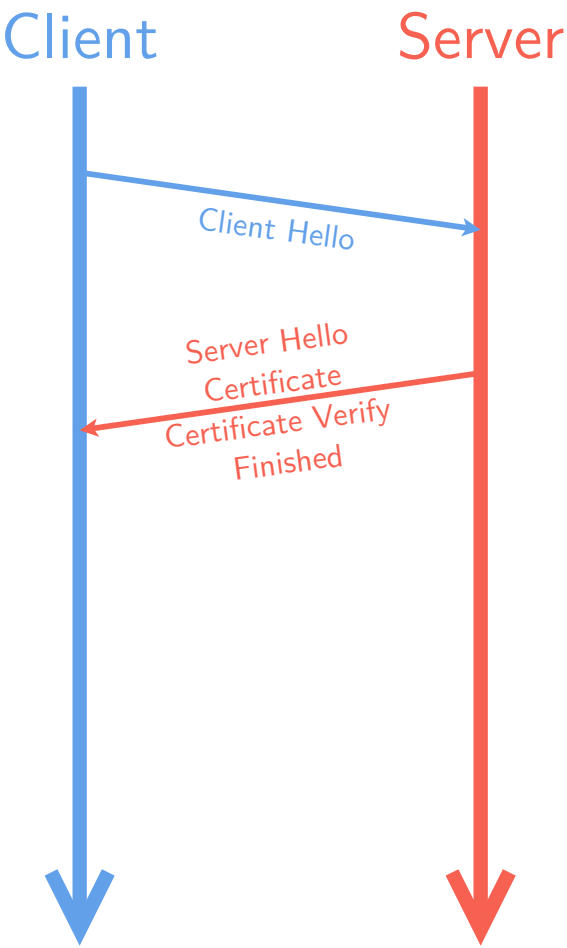
TLS handshake with server authentication only

- Assumptions
- Client: Client Hello
- Server: Server Hello
  - Server reply agreeing on the configuration to use
  - Contains:
    - The selected protocol version (e.g., TLS 1.3)
    - A 32 bytes random number
    - The selected symmetric cipher suites, e.g. `AES_128_CCM`
    - One of the key exchange mechanisms offered by the client and its associated values (e.g., Diffie-Hellman with server public parameters).



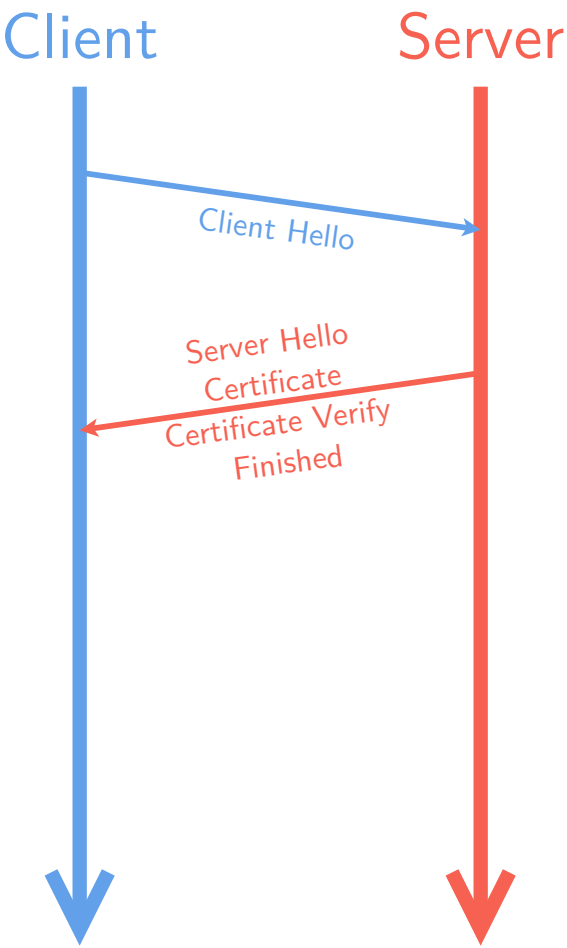
TLS handshake with server authentication only

- Assumptions
- Client: Client Hello
- Server: Server Hello
- Server: Certificate
  - Provides the server certificate
- Server: Certificate Verify
  - Provides a proof the server owns the corresponding private key
  - Contains:
    - A signature of all previous handshake messages
  - Algorithms depend on the certificate, e.g. ECDSA



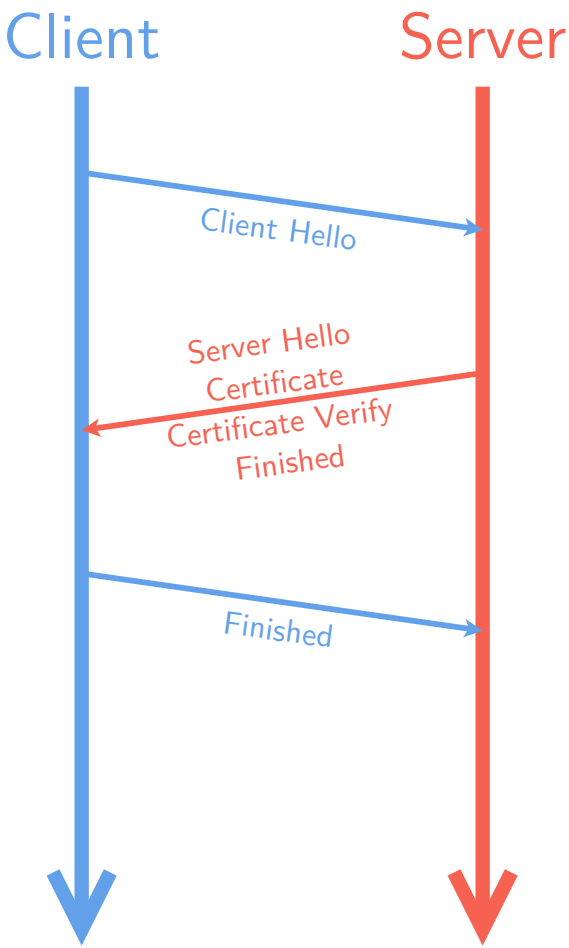
TLS handshake with server authentication only

- Assumptions
- Client: Client Hello
- Server: Server Hello
- Server: Certificate
- Server: Certificate Verify
- Server: Finished
  - Confirms the handshake step is done from the server side: the client can now send Application data
  - Contains:
    - A MAC covering the entire handshake (using an HMAC algorithm)



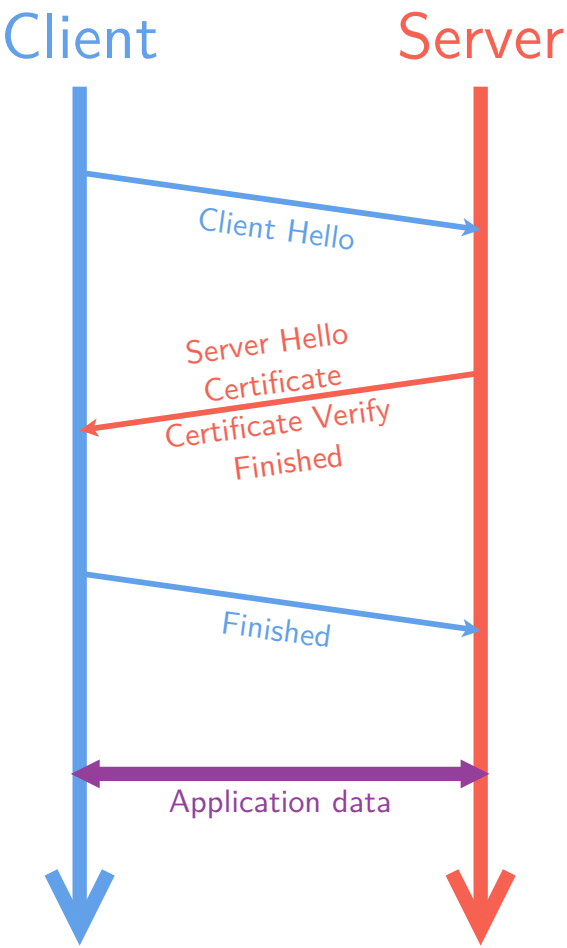
TLS handshake with server authentication only

- Assumptions
- Client: Client Hello
- Server: Server Hello
- Server: Certificate
- Server: Certificate Verify
- Server: Finished
- Client: Finished
  - Confirms the handshake step is done from the client side: the server can now send Application data
  - Contains:
    - A MAC covering the entire handshake (using an HMAC algorithm)



TLS handshake with server authentication only

- Assumptions
- Client: Client Hello
- Server: Server Hello
- Server: Certificate
- Server: Certificate Verify
- Server: Finished
- Client: Finished
- Application data can then be sent between peers
  - Algorithm and operation mode has been selected during the handshake, e.g. `AES_128_CCM`
  - Key has been constructed from the key exchange data during the handshake



Understanding Public Key Infrastructures

Public key certificate

- Certifies that a specific public key is assigned to a specific entity
- Is signed by a certificate authority (CA)
- Certificate signature can then be verified by the CA own certificate
  - This creates a chain of trust extending to a root certificate
  - Root certificates are not guaranteed by a third-party: they have to be known by the client
- The certificate has to follow a specific format, quite often ITU-T X.509 is used
- Certificates can generally be revoked: this information is not presented by the certificate itself but has to be retrieved by other means

Public key certificate content

- Content of the certificate will depend on its goal and the CA policy
- Common content:
  - A subject: which entities this certificate belongs to. This may be a machine, an individual, or an organization
  - A serial number, allowing one to uniquely identify each certificate
  - The period of validity
  - The purpose of the certificate: authenticate the entity for a specific service or a specific operation
  - The public key
  - The issuer (CA) and its signature

Public key infrastructure

- Certificates should be assigned to each entities, allowing them to authenticate to their peers
- These certificates should be signed by a certificate authority and issued to legitimate users
- The Public key infrastructure defines the organisation and processes governing these certificates
- PKI use case examples:
  - Establishing TLS connections, e.g. over HTTP
  - Authenticate and encrypt e-mail messages, e.g. with openPGP
  - Authenticate devices in a fleet, where the PKI is entirely managed by the device vendor

Practical lab - Experiment basic cryptographic tools

Time to apply some fundamental concepts!

- Using OpenSSL to generate a key hierarchy
- Exchange and verify encrypted and signed messages
- Experimenting with key revocation
- Performance comparison of symmetric and asymmetric encryption algorithms

Hardware-enforced security barriers

Privilege levels

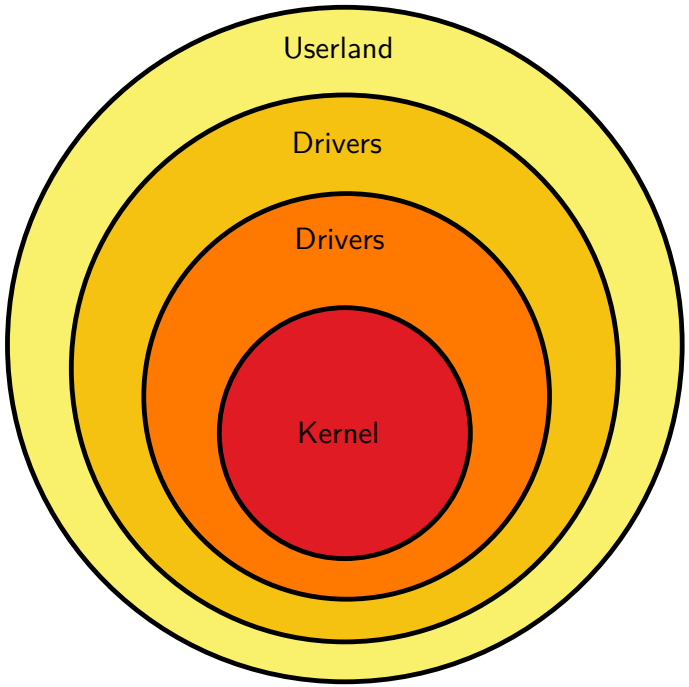
Kernel/Userland isolation

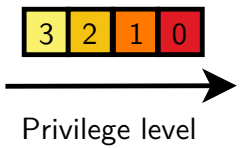
- We expect some security guarantees from modern systems:
  - Process isolation (resources, address space)
  - User isolation
  - Access control on files
- We need an entity in charge of enforcing those guarantees
- That entity is the operating system's kernel, this is one of its main roles

Kernel/Userland isolation

- The kernel has **privileged** access to the hardware
- On older CPUs, or some architectures, it even has **unfiltered** access
- Needs to be isolated from userland even though both run on the same hardware
- The key is the **privilege level**
  - x86 defines 4 “protection rings”, tracked in 2 bits of the Code Segment Selector register Only 2 are actually used: ring 0 for the kernel, ring 3 for userland
  - ARM has 4 “Exception Levels” (EL), with the kernel in EL1 and userland in EL0. The current EL is tracked in the CurrentEL register
- The privilege level is checked e.g. when accessing memory
- Some instructions can only be executed from elevated privilege levels

Protection rings





System Calls (1/2)

- A system call allows userspace to request services from the kernel by executing a special instruction that will switch to kernel mode (man 2 syscall)
- Executing functions provided by the libc (`read()`, `write()`, etc) often ends up executing a system call.
- System calls are identified by a numeric identifier that is passed via the registers.
  - The kernel exports some defines (in `unistd.h`) that are named `__NR_<syscall>` and defines the syscall identifiers.

```
#define __NR_read 63
#define __NR_write 64
```

System Calls (2/2)

- The kernel holds a table of function pointers which matches these identifiers and will invoke the correct handler after checking the validity of the syscall.
- System call parameters are passed via registers (up to 6).
- When executing this instruction the CPU will change its execution state and switch to kernel mode.
- Each architecture uses a specific hardware mechanism (man 2 syscall)

```
mov w8, #__NR_getpid
svc #0
tstne x0, x1
```

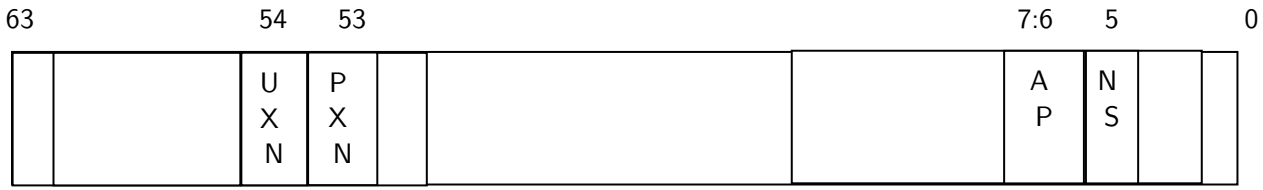
Virtual address translation

- When a core is running a **user program** it is **not** running the **kernel**
- The hardware itself must be able to enforce memory protection
  - this is where the MMU comes in
- The CPU accesses **virtual**, not **physical** addresses
- Each process has a different virtual address space
- Virtual memory is organized into a hierarchy of tables, the lowest level being pages.
  See mm/page_tables
- Table elements are descriptors, and contain several attributes
- Documentation:
  - ARMv8-A: Armv8-A Address Translation
  - x86: Intel Software Developer's Manual, Volume 3a, Chapter 5

Memory attributes

- Page descriptors use a hardware-specific format:
- PTE_USER() (ARM64) / _PAGE_BIT_USER() (x86) indicates userland
- An entry can be marked non-executable:
  - on x86, using the NX bit (_PAGE_BIT_NX())
  - on 64-bit ARM, split between EL0 (PTE_UXN()) and EL1+ (PTE_PXN())
- Userland (ring 3/EL0) execution from kernel (ring 0 /EL1) is conditional:
  - x86: depends on SMEP, see Intel SDM, Vol 3a, 5.6.1
  - 64-bit ARM: this is why **XN** is split into PTE_UXN() and PTE_PXN()
- On x86, userland (ring 3) access from kernel (ring 0) can be disabled using SMAP

Memory attributes (ARMv8)



- NS set to 1 means we are in normal world
- bit 7 (AP.1) encodes read-only
- bit 6 (AP.0) encodes unprivileged (EL0) access

AP		bit 7	bit 6	EL0	EL1+
00		0	0	No access	RW
01		0	1	RW	RW
10		1	0	No access	RO
11		1	1	RO	RO

Experimenting with memory attributes

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>

int main(int argc, char *argv[])
{
        unsigned char *ptr = mmap(
            NULL, 4096 * 2, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0
        );

        printf("ptr: %pn", ptr);                                    // 0x7XXXXXXXX000
        ptr[0] = 0xFF;
        printf("ptr[1025]: 0x%02xn", ptr[1025]);                    // 0x00
        printf("ptr[4097]: 0x%02xn", ptr[4097]);                    // 0x00

        unsigned int ret = mprotect(ptr, 1024,  PROT_NONE);

        printf("ptr[4097]: 0x%02xn", ptr[4097]);                    // 0x00
        printf("ptr[1025]: 0x%02xn", ptr[1025]);                    // SIGSEGV
}
```

Limits to this model

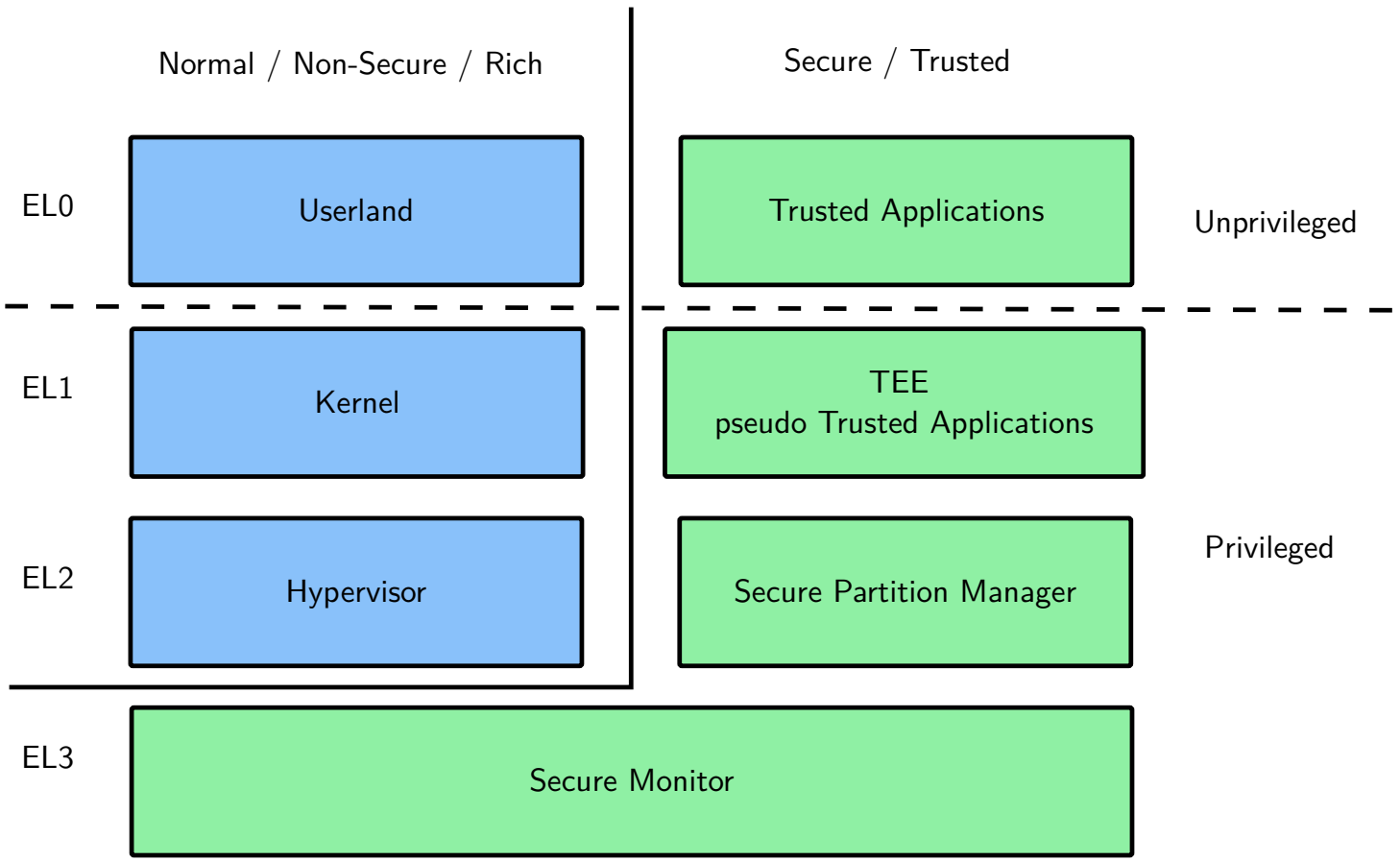
- Although this model is widespread, it has limitations
- Sometimes, the kernel is untrusted:
  - Host virtual machine kernel
  - “Custom ROMs”
- Or it is not trusted **enough**:
  - Large attack surface
  - Not Invented Here
  - Money (not just the user’s) is at stake
- This gave rise to multiple hardware isolation technologies

ARM TrustZone

TrustZone / Secure World

- TrustZone is an ARM-only hardware isolation mechanism
- It is implemented on the Application(A)-profiles of ARMv7-9, this is what we discuss
- Also implemented on the ARMv7-M (Microcontroller) profile, but differently
- Splits the system into 2 **worlds**: Secure and non-Secure
- They are also called **Trusted** / **Rich** Execution Environment (TEE/REE)
- This is **orthogonal** to Exception Levels:
  - EL3 (Secure Monitor) only exists in **Secure** world
  - EL2, if it exists, can exist in **Secure**, **Non-Secure**, or both
  - EL1 and EL0 must exist in **both** security states

TrustZone / Secure World



World transitions: Secure to Non-Secure

- The CPU starts executing at the **highest** EL: EL3, so in Secure World
- It then sets up the rest of Secure World, then Normal World, in stages
- Similar to the EL, the CPU keeps track of the security state in a register
  - SCR is the *Secure Configuration Register (SCR)*, bit 0 is the *NS* bit
  - SCR can only be accessed in Secure EL1-3
  - Going from Secure World back to Non-Secure is simply setting *NS* to 1
  - The Security model expects only the Secure Monitor (running at EL3) to do this transition
- The ARM spec leaves non-EL3 access to SCR as *undefined*
  - In practice, accessing SCR from non-secure or from S-EL0 will fault.

World transitions: Back to Secure

- The systems transitions from Non-Secure to Secure via exceptions, either:
  - asynchronous, via some *FIQ* interrupts, depending on *SCR.FIQ*
  - synchronous, generated by an *smc* (*secure monitor call*) instruction
- *smc* cannot be called from EL0, so userland would first need to issue a *svc*
- The authority is the *Secure Configuration Register (SCR)*'s NS bit

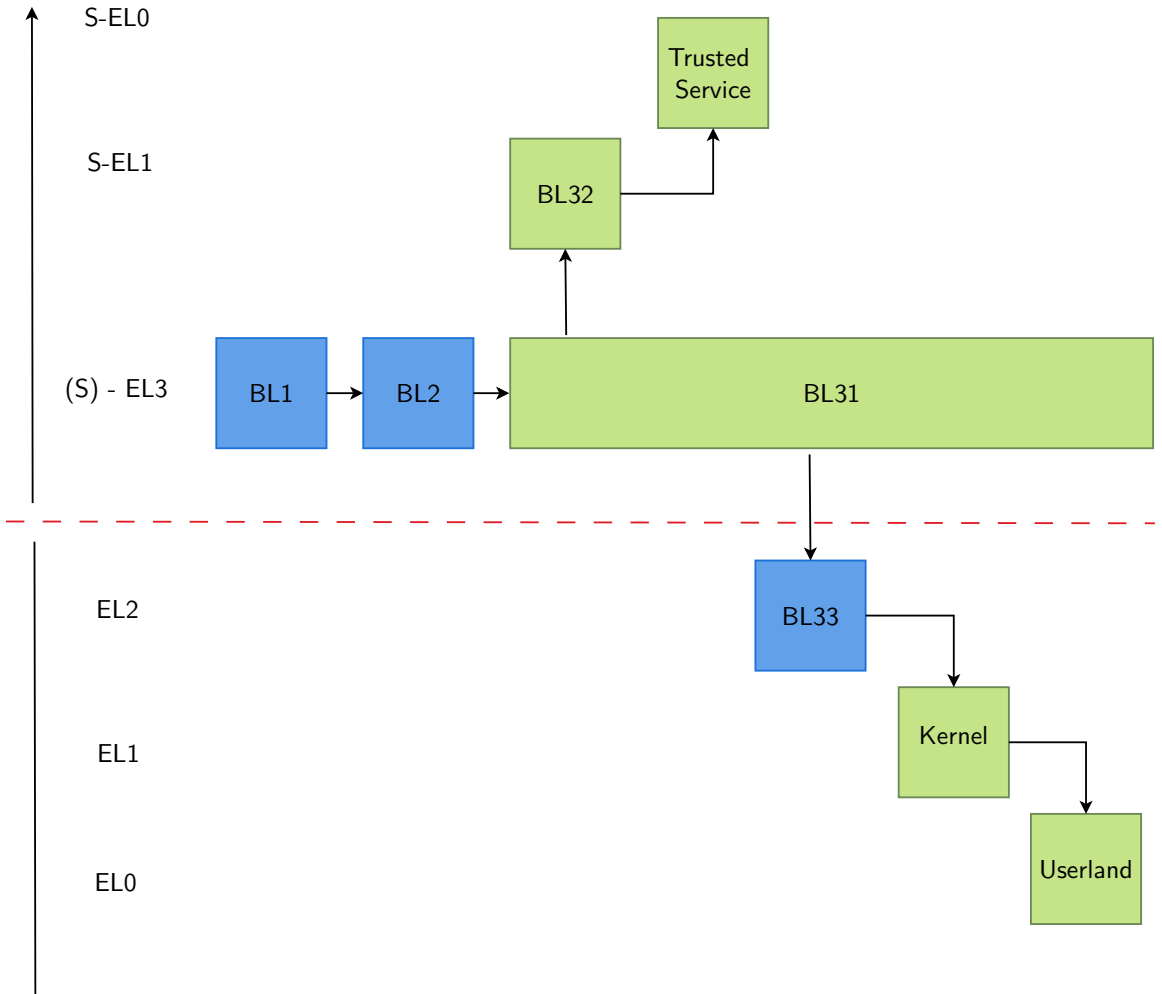
Arm Trusted Firmware

- Trusted Firmware is an open-source reference Secure World implementation
- It is the default Secure World for most ARM platforms
- Comes in two flavours:
  - Trusted Firmware A (for the Application profile), or TF-A
  - Trusted Firmware M (for the Microcontroler profile) TF-M
- There is now a Rust implementation of TF-A called Rusted Firmware A (RF-A)
- Most importantly, TF-A implements the Secure Monitor
- Uses a specific nomenclature to designate boot components

Trusted Firmware A (TF-A)

- TF-A boot is split into 5 steps:
  - BootLoader 1, or BL1: the SoC vendor's ROM code
  - BootLoader 2, or BL2: "Trusted Boot firmware"
  - BootLoader 3-1, or BL31: EL3 Runtime
  - BootLoader 3-2, or BL32 (optional): Secure-EL1 payload (Trusted OS, Trusted kernel, TEE)
  - BootLoader 3-3, or BL33: Non-trusted firmware, the "normal" bootloader
- Only BL31 and BL32 are resident after boot is complete:
  - BL31 is responsible for routing Secure exceptions (and smcs) to the Secure World
  - BL32 handles some of these exceptions and smcs, potentially on behalf of TAs

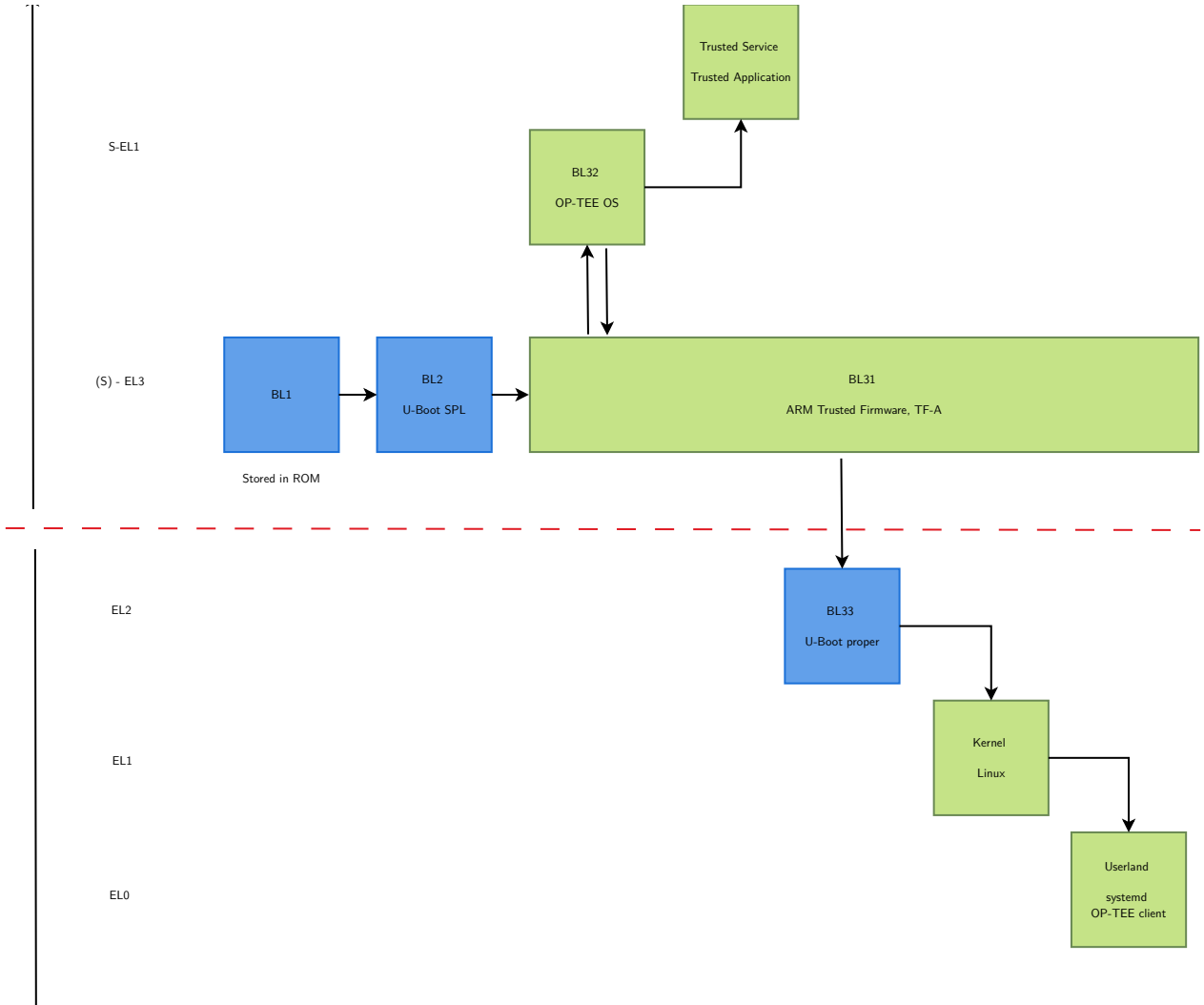
ARMv8 boot sequence: TF-A names



Trusted Firmware A (TF-A)

- In a “standard” boot, where U-Boot is used as boot loader:
  - BL1: the vendor bootROM, non-resident
  - BL2: is implemented by the Second Program Loader (SPL)
  - BL31: the runtime component of TF-A resident in EL3
  - BL32: OP-TEE
  - BL33: U-Boot proper

ARMv8 boot sequence



Rationale

- The aim of TrustZone is to create a Trusted Execution Environment (TEE)
- The isolation is backed by the hardware
- Some hardware resources are only accessible to the TEE
- Sounds familiar, why is it not simply the kernel?
  - attack surface: the TEE should be as small as possible
  - usage: can you ensure attackers do not get *root*
  - virtualization: offer services securely to kernels you don't trust
- The key concept is defense in depth

Trusted Execution Environments

- The TEE is the part of Secure World that does things for the user
- This is an A profile concept, the M profile equivalent is Secure Processing Environment (SPE)
- There are different implementations of TEEs:
  - Trustsonic's Kinibi
  - Samsung's Knox
  - Qualcomm's QTEE
  - Huawei's iTrustee
- Some are even Open Source:
  - Android's Trusty, based on Little Kernel (LK)
  - Nvidia's Trusted Little Kernel (TLK), also based on LK: `git://nv-tegra.nvidia.com/3rdparty/ote_partner/tlk.git`
  - OP-TEE, the reference Open Source TEE

Trusted Execution Environments

- The TEE is an optional component
- It does not normally offer service to Normal World directly
- It does provide HW access to Trusted Applications
- The TEE is the Secure World equivalent of the kernel, running in S-EL1
- For further isolation, ARMv8 introduces Secure Partitions
  - Essentially run different instances of TEEs in isolated HW
  - A Secure Partition Manager runs at S-EL2 (SPMC) and (S-)EL3 (SPMD)
- ARMv9 pushed this even further with the Realms extension

Trusted Applications

- Trusted Applications are the secure world “userland”
- OP-TEE has 2 main types of Trusted Apps:
  - Pseudo-TAs, statically compiled into the OP-TEE core binary
  - Usermode TAs are dynamically loaded by OP-TEE and run at S-EL0.
- Usermode TAs can be further subdivided into:
  - Early TAs: still stored in OP-TEE core, in a `data` section
  - REE FS TAs: loaded by the REE via `tee-supplicant`.
- Because REE FS TAs are loaded from the REE, the TEE needs to authenticate them
  - OP-TEE expects REE FS TAs to be signed
  - the OP-TEE core embarks the public part of `default_ta.pem`
  - do **NOT** ship OP-TEE with that key

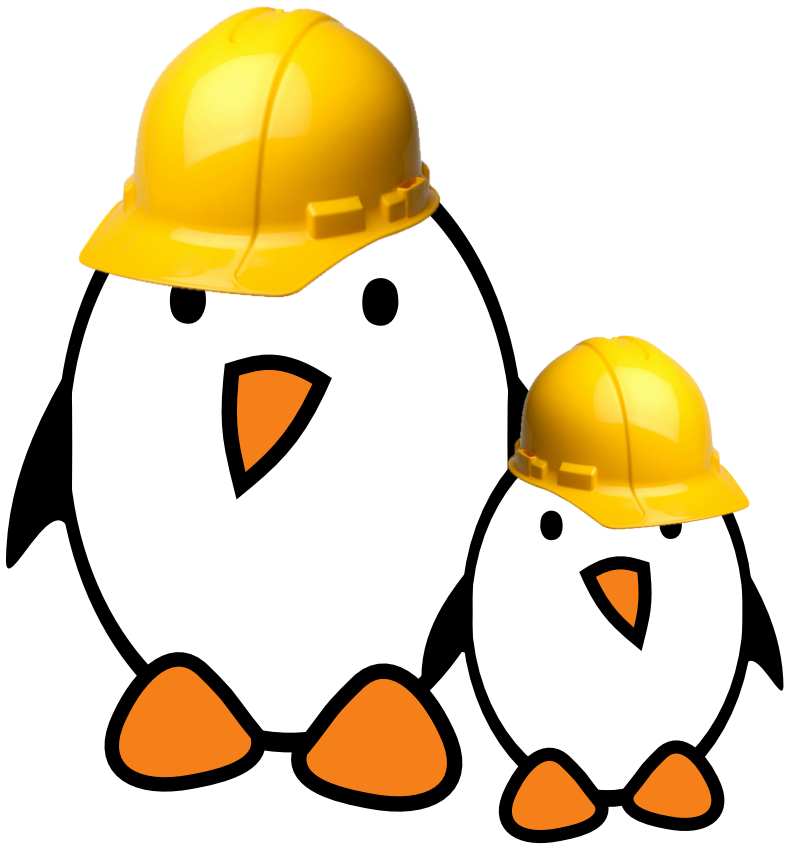
Usecases

- "Software HSM": this is why OP-TEE has a PKCS#11 TA
- fTPM: OP-TEE
- Payment processing
- DRM
- Biometrics (fingerprint unlock)
- Hardware-isolated detection of non-secure malware

Issues

- TrustZone is, ultimately, another security layer
- The security guarantees it offers are only as strong as the implementation
- Although it is hardware-backed, it is only useful if software is involved
- Like all software, Secure World software can have bugs
  - Arbitrary code execution in a Samsung TEE
  - Other TEEGris vulnerabilities
  - S-EL0 stack buffer overflow in Qualcomm's TEE
  - EL3 execution via Samsung Kinibi
- Secure world configuration can be very HW-dependent

Practical lab - Exploring the secure world



Time to use hardware security barriers!

- Building secure world software (TF-A, OP-TEE, TA)
- Adding logs to observe Exception level transitions
- Adding logs to observe world transitions
- Writing a small Trusted Application in OP-TEE
- Interacting with our Trusted Application from userland

Secure Boot

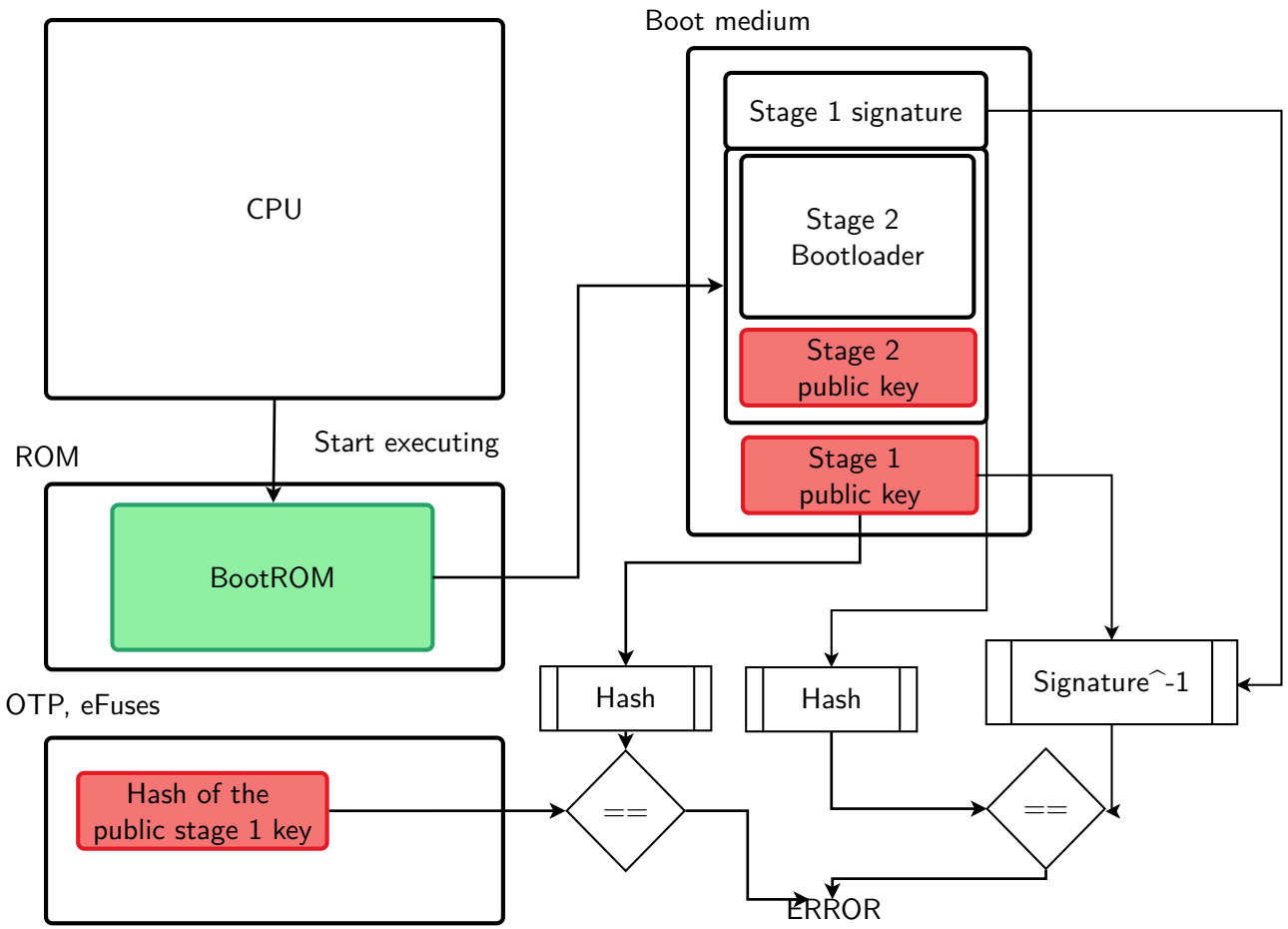
Purpose

- Start from the following premises:
  - the system's software was installed securely
  - the system enters production and might be exposed to threats
  - some of those threats will know how to install software
- The question is: how do I make sure the software has not been altered?
- The system must be able to assess autonomously

Concept

- Chain of verification of software
  - the manufacturer's ROM code verifies the first stage bootloader
  - the last stage bootloader (e.g. U-Boot) verifies the OS kernel
  - the kernel, optionally, verifies the userland
- Compute the hash of the next stage binary
- Use the embedded **public** key to retrieve the signed hash
- Compare re-computed hash to extracted hash
- Refuse to hand over control flow if the hashes do not match

Initial bootloader verification

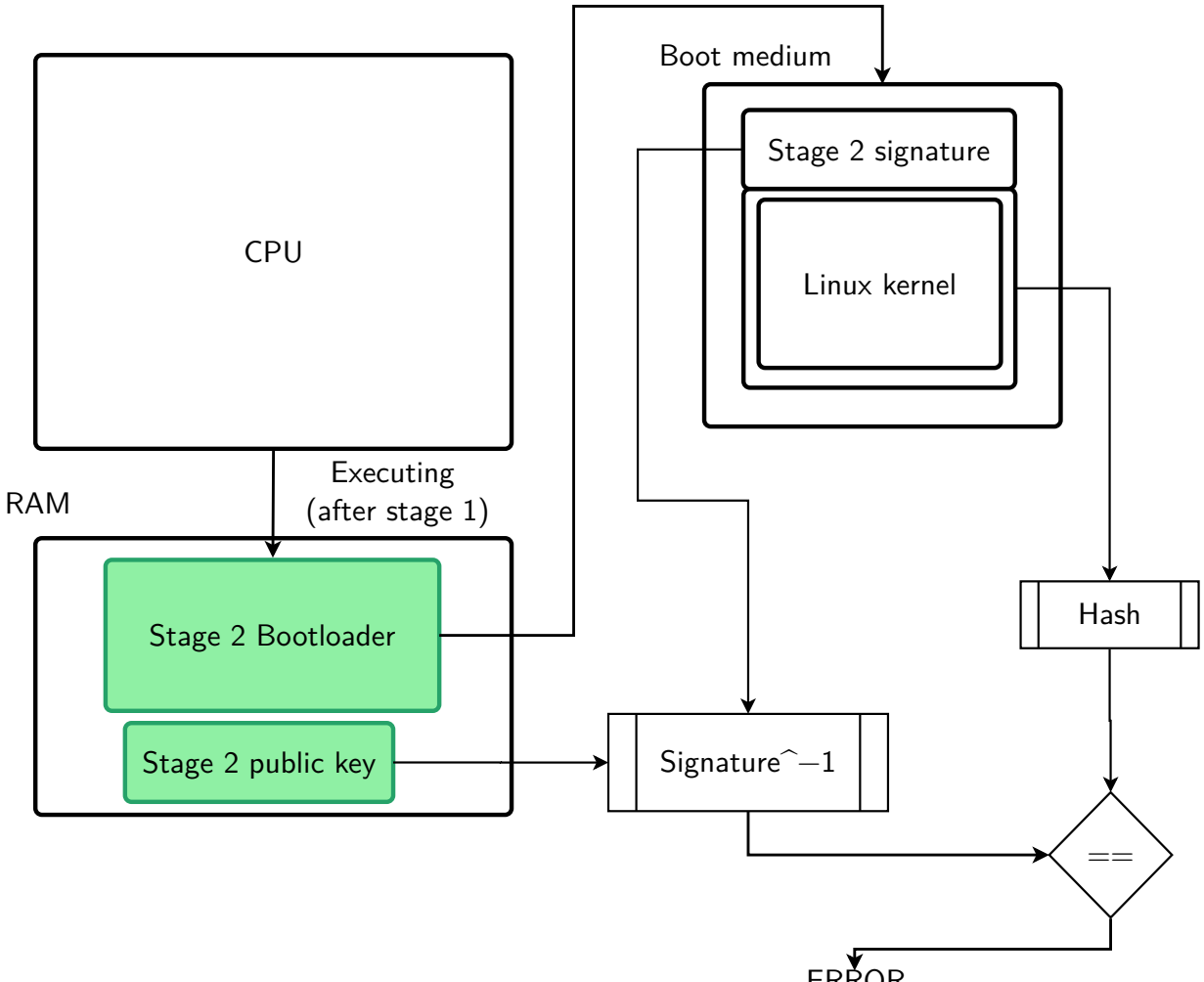


- The public key's hash is stored in a once-writable memory
- The BootROM uses it to validate the embedded key
- It uses that key to recompute the hash from the signature

Initial bootloader verification

- Storing the hash instead of the full public key means the storage is independent of the signature algorithm. It only depends on the hash algorithm.
- The BootROM must be trusted. It cannot be modified, as it is stored in a read-only memory.
- The key hash is stored in a memory that can only be written once. This is usually done by burning electrical fuses.
- In some implementations, one more fuse must be burned once the implementation has been verified to be correct to enable secure boot.

Kernel verification



- End of the "traditional" secure boot chain
- Verify the kernel before execution
- Must be supported by the bootloader

Kernel verification

- This stage no longer depends on hardware
- The root of trust for this stage is the combined:
  - last stage bootloader (e.g. U-Boot proper)
  - last stage signature public key
- Both should have been verified in the previous stage
- This ensures that the kernel that has been started is the one that the possessor of the last stage private key expects.
- This leaves some unanswered questions, which we'll answer in the next part:
  - What happens once the kernel starts *init*?
  - What about the DTB?
  - What about the command line parameters?

Examples

- In the x86/PC world:
  - usually implemented via UEFI.
  - Pushed by Microsoft, Windows 11 makes it a requirement
- On Android: Android Verified Boot (AVB)
  This is why "bootloader unlocks" are necessary
- In U-Boot, FIT (Flat Image Tree) images can be signed.

Threat Model

- Secure boot is designed to protect against unauthorized modification of software. This includes:
  - Offline modification of the software by reflashing the boot medium.
    This could be how an attacker gain access.
  - Runtime rewriting of the software (persistence).
    This is defense-in-depth against an attacker who already has access.
- It is not designed to protect against:
  - Runtime compromise of the system via a vulnerability
- It can be designed to protect against:
  - Leaking of some of the cryptographic material
- Secure boot is a chain, so its security is a consequence of:
  - The security of the root/anchor.
  - The security of each link, guaranteed by the signature scheme.

Threat Model: the root

- The root is made of the elements that are initially trusted at boot:
  - The hash(es) of the secure boot public key(s)
  - The bootROM
  - The CPU
  - Optionally, hardware implementations of cryptography
- Of these, only the hash(es) are in an integrator's control
- The SoC vendor is very much part of the trusted perimeter.

Example: UEFI secure boot

Example: UEFI secure boot

- Secure boot is part of the UEFI spec
- Theoretically support x86-64, ARM32, aarch64, loongarch and RISC-V
- Mostly used on x86
- UEFI uses a modified version of Microsoft's Portable Executable (PE) file format
- Must be authenticated by the CPU first
  - Intel Boot Guard
  - AMD Platform Secure Boot (PSB)

Example: Raspberry Pi secure boot

Raspberry Pi Secure boot support

- Since Raspberry Pi4, RPis support secure boot
- Raspberry Pi Ltd has 2 whitepapers on the topic:
  - Boot security
  - Secure Boot
- The BootROM holds 4 RSA 2048 public keys
- These keys are held by the RPi foundation, the first one is a dev key
- Broadcom's BCM2711 SoC embeds a one-time programmable (OTP) memory. It holds:
  - revocation status of the ROM keys
  - Configuration (e.g. boot mode)
  - 1 slot for a SHA256 hash of an OEM key
  - 256-bit device unique private key (readable to root in cleartext)

RPi: Secure boot process

- The RPi BootROM first:
  - checks the OTP revocation bits
  - Verifies `bootsys`' signature from EEPROM against a non-revoked key
  - starts `bootsys` (if verification succeeded)
- `bootsys` then:
  - verifies the OEM key in EEPROM against the OTP hash
  - verifies `boot.conf` (in EEPROM) against `boot.sig` with the OEM key
  - verifies `bootmain` (in EEPROM) against the embedded hash (which was therefore signed with the RPi ROM key)
  - loads and executes `bootmain` if verification succeeded
- `bootmain` then
  - verifies `boot.img` against `boot.sig` (in boot medium) against the OEM key
  - loads the `boot.img` and starts `start.elf`
- Finally, `start.elf` loads the kernel from the `boot.img` ramdisk

RPi: Secure boot overview

- It is possible to test the `boot.img` signature

```
3.04 OTP boardrev b04170 bootrom a a
3.06 Customer key hash
8251a63a2edee9d8f710d63e9da5d639064929ce15a2238986a189ac6fcd3cee
3.13 VC-JTAG unlocked
3.36 RP1_BOOT chip ID: 0x20001927
3.41 bootconf.sig
3.41 hash: f71ede8fad8bea2f853bcff41173ffedde48c5b76ed46bc38fa057ce46e5d58b
3.47 rsa2048:
3f215305d5aff620219da94f6f1294787e3a407102a507da96c28e9195d3ccb2f144cac66919f9d86ba9f54
3.94 RSA verify
3.10 rsa-verify pass (0x0)
```

- On RPi4 , it is also possible to test the EEPROM signature
- This is no longer possible on RPi5:
  - The EEPROM image must be counter-signed using the OEM key
  - If the OTP has not been flashed, boot will not proceed

Example: AHAB on NXP i.MX93

Detailed example: i.MX93

- i.MX SoCs implement a version of secure boot called High Assurance Boot (HAB)
- Starting from i.MX8, moved to Advanced HAB (AHAB)
- Uses a table of 4 asymmetric keys called Secure Root Keys (SRKs)
- The hash of the SRK table is stored in the SoC's fuses
- AHAB actions can be performed using NXP's Secure Provisioning SDK (SPSDK)

The EdgeLock (Secure) Enclave (ELE)

- The ELE is the security subsystem on the i.MX9 family
- Replaces the SEcurity COntroller (SECO) of the i.MX8 family
- Also called “Sentinel”
- Based on a dedicated RISC-V core
- Authenticates all firmware loaded at boot
- ELE firmware implements AHAB. Signed using ELE SRKs provided as a binary blob by NXP.

The EdgeLock (Secure) Enclave (ELE)

- Described in the ELE API Reference Guide (IMX93ELEAPI) (NXP account required for download)
- HSM capability, API described in RM00284
  - Key generation
  - Key storage (no internal NVM, this is important)
  - Encryption/Decryption
  - Signature (and verification)
- Using the ELE to wrap keys for filesystem encryption is possible
- No ELE support for Trusted Keys
  - LUKS will not be supported, only "naked" cryptsetup
  - An initramfs will most likely be necessary
  - Will require some provisioning to initially wrap the key
- API accessible from userland via a messaging unit
- NXP has a library for this: imx-secure-enclave

Detailed example: i.MX93 - AHAB container

- AHAB containers are custom structured binary files
- They can be built using SPSDK's nxpimage
  - Generate a template: `nxpimage ahab get-template`
- They will include one or several binaries
- The `signer` property will indicate how to sign the binaries. It can point to:
  - A PEM file containing the private key
  - A signature provider (e.g. a PKCS#11 URI, for instance pointing to an HSM)

Detailed example: i.MX93 - AHAB container template

```
# Description: NXP chip family identifier.
family: mimx9352
# --------------===== MCU revision [Optional] =====---------------------------------
# Description: Revision of silicon. The 'latest' name, means most current revision.
# Possible options: <a0, a1, latest>
revision: a1
# --------------===== Memory type [Required] =====---------------------------------
# Description: Specify type of memory used by bootable image description.
memory_type: serial_downloader
init_offset: 0
# --------------===== Primary Image Container Set [Optional] =====-----------------
# Container Set that is validated by ROM and usually contains DDR init and SPL.
# It could be used as pre-prepared binary form of AHAB and also YAML configuration
# file for AHAB. In case that YAML configuration file is used, the Bootable image tool
# builds the AHAB itself.
primary_image_container_set: ahab_primary_container.yaml
secondary_image_container_set: ahab_secondary_container.yaml
```

Detailed example: i.MX93 - AHAB primary container

```
family: mimx9352
revision: a1
target_memory: serial_downloader
output: ahab-primary-container-set.bin
containers:
  - binary_container:
      path: mx93a1-ahab-container.img   # ELE firmware, provided and signed by NXP
  - container:
      srk_set: oem
      used_srk_id: 0
      signer: Super_Root_Key_1.pem
      images:
        -
          lpddr_imem_1d: lpddr4_imem_1d_v202201.bin     # LPDDR memory FW in 1D mode
          lpddr_imem_2d: lpddr4_imem_2d_v202201.bin     # LPDDR memory FW in 2D mode
          lpddr_dmem_1d: lpddr4_dmem_1d_v202201.bin     # LPDDR memory data in 1D mode
          lpddr_dmem_2d: lpddr4_dmem_2d_v202201.bin     # LPDDR memory data in 2D mode
          spl_ddr: u-boot-spl.bin       # SPL
      srk_table:
        flag_ca: false
        hash_algorithm: default
        srk_array:
          - Super_Root_Key_1.pub    # 4 lines, one for each SRK
```

Detailed example: i.MX93 - AHAB secondary container

```
[...]
containers:
  -
    container:
      srk_set: oem
      used_srk_id: 0
      signer: Super_Root_Key_1.pem
      images:
        - atf: bl31-imx93.bin-optee
        - uboot: u-boot.bin
        - tee: tee.bin
      srk_table:
        flag_ca: false
        hash_algorithm: default
        srk_array:
          - Super_Root_Key_1.pub
          - Super_Root_Key_2.pub
          - Super_Root_Key_3.pub
          - Super_Root_Key_4.pub
```

Detailed example: i.MX93 - AHAB

- As mentioned, the hashes of the SRKs must be flashed to the SoC's fuses.
- The SPSDK sources show that those fuses are index `0x80` to `0x87`
- On i.MX93, the hash function being used is SHA256, so the hash will be spread over 8 32-bit fuses.
- This hash is calculated over all 4 SRKs, they are not fully independent
- The fuses will be flashed by the ELE, via the usual message interface
  - U-Boot has an ele_message command
- This is a **sensitive** step, so SPSDK can generate a script to automate it

Example

```
# nxpele AHAB SRK fuses programming script
# Generated by SPSDK 3.0.0.dev68+g99003d2be
# Family: mimx9352, Revision: latest

# Value: 0xCB2CC774B2DCEC92C840ECA0646B78F8D3661D3A43ED265A490A13ACA75E190A
# Description: SHA256 hash digest of hash of four SRK keys
# Grouped register name: SRKH

# OTP ID: OEM_SRKH7, Value: 0x74C72CCB
write-fuse --index 128 --data 0x74C72CCB
# OTP ID: OEM_SRKH6, Value: 0x92ECDCB2
write-fuse --index 129 --data 0x92ECDCB2
# OTP ID: OEM_SRKH5, Value: 0xA0EC40C8
write-fuse --index 130 --data 0xA0EC40C8
# OTP ID: OEM_SRKH4, Value: 0xF8786B64
write-fuse --index 131 --data 0xF8786B64
# OTP ID: OEM_SRKH3, Value: 0x3A1D66D3
write-fuse --index 132 --data 0x3A1D66D3
# OTP ID: OEM_SRKH2, Value: 0x5A26ED43
write-fuse --index 133 --data 0x5A26ED43
# OTP ID: OEM_SRKH1, Value: 0xAC130A49
write-fuse --index 134 --data 0xAC130A49
# OTP ID: OEM_SRKH0, Value: 0x0A195EA7
write-fuse --index 135 --data 0xA195EA7
```

AHAB status

- The signature of an AHAB container is always verified
- Before the fuses are flashed, this will result in the following error:

```
> ahab_status
Lifecycle: 0x00000008, OEM Open

     0x0287fad6
     IPC = MU APD (0x2)
     CMD = ELE_OEM_CNTN_AUTH_REQ (0x87)
     IND = ELE_BAD_KEY_HASH_FAILURE_IND (0xFA)
     STA = ELE_SUCCESS_IND (0xD6)

     0x0287fad6
     IPC = MU APD (0x2)
     CMD = ELE_OEM_CNTN_AUTH_REQ (0x87)
     IND = ELE_BAD_KEY_HASH_FAILURE_IND (0xFA)
     STA = ELE_SUCCESS_IND (0xD6)
```

- Once they are, AHAB should report no events:

```
> ahab_status
Lifecycle: 0x00000008, OEM Open


    No Events Found!
```

- Then the board can be set to `OEM Closed`, and signatures will be enforced

Practical lab - Exploring Secure boot first stages

Time to setup AHAB on i.MX93!

- Building all software components for secure boot (AHAB) on i.MX93
- Creating a signed AHAB container using SPSDK
- Flashing the fuses on the i.MX93 to enable secure boot (AHAB)
- Testing that secure boot (AHAB) is correctly enabled

Secure Boot - Later stages

U-Boot components

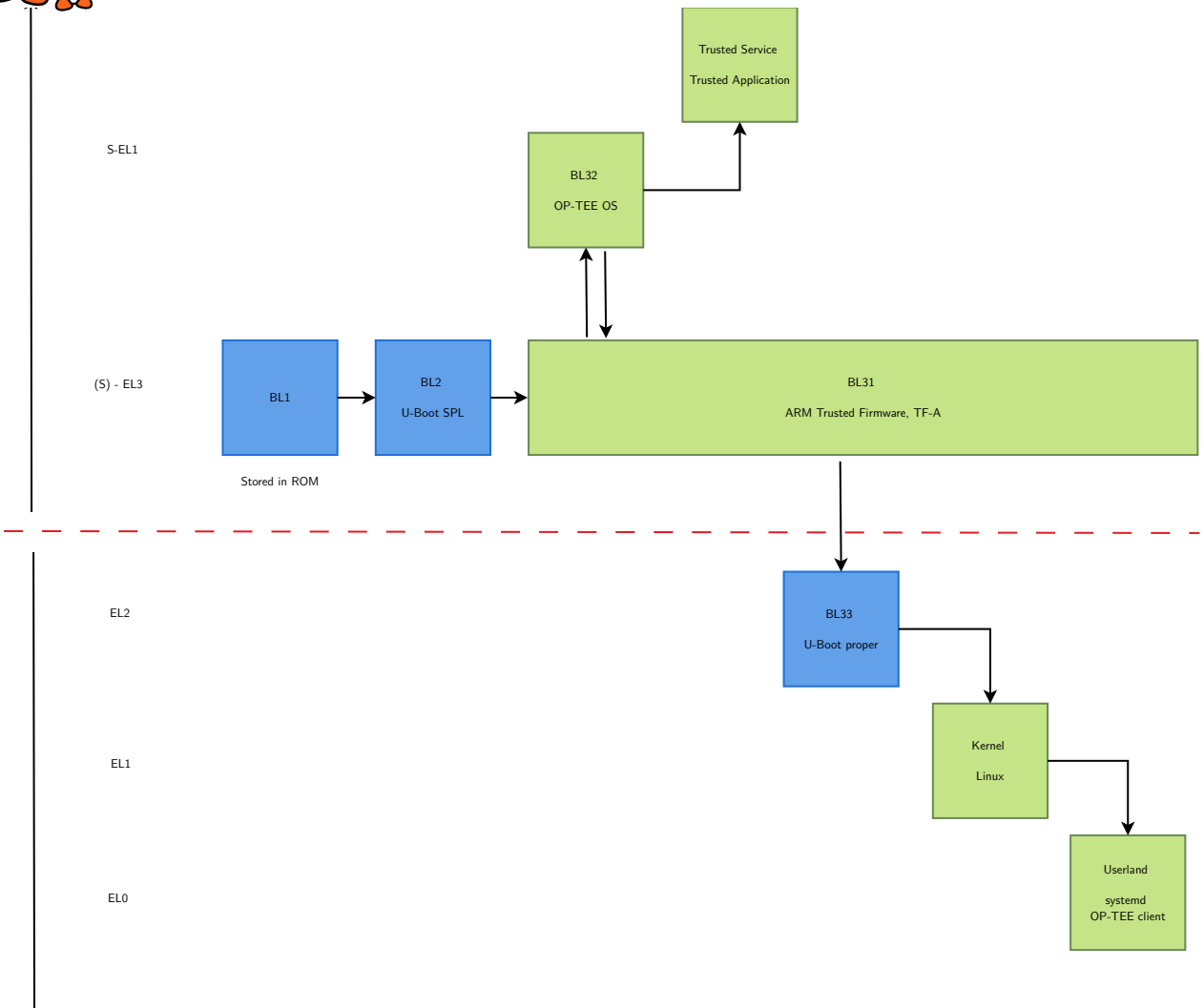
Early boot

- The system (most likely) has 2 type of RAM:
  - A small on-chip RAM (640 KB for i.MX93, 200KB for RK3399)
  - One or several big (GiB) **external** memory ((LP)DDR) bank(s)
- The on-chip RAM is **static** and requires little initialization, which can be done by the BootROM.
- The **external** RAM is **dynamic**, it and its controller needs to be initialized
- At startup, we must start running from the on-chip RAM, so at least the **external** RAM init logic must have a small memory footprint.
- The same goes for using NAND flash as a boot medium.
- Modern bootloaders have a lot of features, and therefore are too big. Our i.MX93 U-Boot is 850 kiB
- We need to split up the boot: a small loader will initialize the **external** RAM, and load the main bootloader there

Secondary Program Loader (SPL)

- Usually started by the bootROM, also at EL3
- In ARM terms, this is **BL2**
- Runs either from SRAM or directly from flash
- Initializes DRAM and DRAM controller, usually using firmware
- Loads the **proper** bootloader
- Can be stripped down to save space
- On TrustZone systems, actually loads TF-A, as well, which returns to U-Boot
- Some SoCs (e.g. Rockchip) split DRAM initialization into another loader, the TPL.

Boot sequence refresher



The “proper” bootloader

- On TrustZone platforms, started by TF-A, running at NS-EL2
- In ARM terms, this is **BL32**
- Runs from **external** DRAM
- More complex, has support for more peripherals and filesystems
- Goal is to load and start the kernel with proper boot arguments
- In case of secure boot, verifies the kernel’s signature

(Das) U-Boot

U-Boot

- U-Boot is the de-facto default embedded systems bootloader
- Has extensive SoC and boards support
- Supports building a TPL and SPL

U-Boot image formats

- U-Boot needs to load several components:
  - the kernel
  - on ARM, the Device Tree Blob (DTB)
  - on ARM, SPL needs to load TF-A
  - on ARM, optionally OP-TEE
  - optionally, an external ramdisk
- We could install all those components separately, and have U-Boot load each
- Having one or two images bundling all components would be more practical
- This is the role of U-Boot's mkimage

U-boot image formats

- U-boot and mkimage support various image formats
- Some are SoC vendor-specific
- U-Boot can only boot 3:
  - legacy image, which is concatenated files with a simple header
  - FIT (Flattened Image Tree)
  - Android boot image, which uses this header
- FIT is the current format for non-Android devices

Flattened Image Tree (FIT)

- FIT images are Flattened Device Trees/ Device Tree Blobs that respect additional constraints.
- The FIT image specification was split off of U-Boot
- They are essentially containers for sub-images
- Each of these images can include a signature
- The FIT image can also list configurations, which list combinations of the images that can be used.
- Configurations can be signed. The `sign-images` property will list the images to include in the signature.
- Support must be enabled via CONFIG_FIT

FIT generation

- U-Boot uses tools/binman/ to generate images
- This relies on a “configuration file” in the form of a DTS
  - that file is named `<pattern>-u-boot.dtsi`
  - the file should be in `arch/<arch>/boot/dts`
  - as documented in tools/binman/binman.rst, binman will look for the following files, in order:
    - `<dts>-u-boot.dtsi` where is the base name of the .dts file
    - `<CONFIG_SYS_SOC>“-u-boot.dtsi`
    - `<CONFIG_SYS_CPU>“-u-boot.dtsi`
    - `<CONFIG_SYS_VENDOR>“-u-boot.dtsi`
    - `u-boot.dtsi`

Example: arch/arm/dts/imx93-u-boot.dtsi

```
/ { binman: binman { multiple-images; }; };
...
&binman {
        u-boot-spl-ddr {
                align = <4>;
                align-size = <4>;
                filename = "u-boot-spl-ddr.bin";
                pad-byte = <0xff>;
                u-boot-spl     { filename = "u-boot-spl.bin"; align-end = <4>; };
                ddr-1d-imem-fw { filename = "lpddr4_imem_1d_v202201.bin"; align-end = <4>; type = "blob-ext"; };
                ...
                ddr-2d-dmem-fw { filename = "lpddr4_dmem_2d_v202201.bin"; align-end = <4>; type = "blob-ext"; };
        };

        spl { filename = "spl.bin";
              mkimage { args = "-n spl/u-boot-spl.cfgout -T imx8image -e 0x2049A000"; blob { filename = "u-boot-spl-ddr.bin"; }; };
        };

        u-boot-container {
                filename = "u-boot-container.bin";
                mkimage { args = "-n u-boot-container.cfgout -T imx8image -e 0x0"; blob { filename = "u-boot.bin"; }; };
        };

        imx-boot {
                filename = "flash.bin";
                pad-byte = <0x00>;
                spl: blob-ext@1 { filename = "spl.bin"; offset = <0x0>; align-size = <0x400>; align = <0x400>; };
                uboot: blob-ext@2 { filename = "u-boot-container.bin"; };
        };
};
```

Configuring FIT verification

- The feature is documented in doc/usage/fit/signature.rst
- CONFIG_FIT_SIGNATURE must be set
  - this will disable CONFIG_LEGACY_IMAGE_FORMAT, as this format cannot be signed
  - it is not enough for signatures to be **required**
- The signing key itself will need to be loaded using a `.dtsi` file
  - This file can be generated using tools/key2dtsi.py
    `tools/key2dtsi.py --required-image fit_signing_key.pub fit_signing_key.dtsi`
- You can then either
  - `#include` your `fit_signing_key.dtsi` from your main device tree
  - Use CONFIG_DEVICE_TREE_INCLUDES to do it for you

fit_signing_key.dtsi

```
        / {
            signature {
                key-fit_signing_key {
                    key-name-hint = "fit_signing_key";
                    algo = "sha256,rsa4096";
                    rsa,num-bits = <4096>;
                    rsa,modulus = [bd 32 f6 a6 5d f7 9a ed
[...]
                                   7c 0b 2f 8e 8f d0 4d 95];
                    rsa,exponent = [00 00 00 00 00 01 00 01];
                    rsa,r-squared = [bc 5b f8 07 15 a2 36 92
[...]
                                     49 1f da e8 b9 74 07 3a];
                   rsa,n0-inverse = <0x7bec6a43>;
                   required = "image";
                };
            };
        };
```

Configuring FIT signing

- Now that U-Boot is configured to verify FIT signature we need to:
  - pack the kernel and DTB into a FIT
  - sign the FIT image
- Both can be done by mkimage
  `tools/mkimage -k keys_dir -f kernel_fit.its kernel_fit.itb -r`
- The `-r` option marks the key as required.
- The `.its` file will need to specify:
  - which files to pack into the FIT image
  - which nodes of the Device Tree to sign

fit.its

```
/dts-v1/;
/ {
        description = "Image for Linux Kernel";
        images {
                kernel {
                        description = "Linux Kernel";
                        data = /incbin/("Image.gz");
                        ...
                        compression = "gzip";
                        load =  <0xDEADBEEF>;
                        entry = <0xDEADBEEF>;
                        hash-1 { algo = "sha256";};
                };
                fdt {
                        description = "fdt";
                        data = /incbin/("platform.dtb");
                        ...
                };
        };
        configurations {
                default = "config-1";
                config-1 {
                        description = "Linux configuration";
                        kernel = "kernel";
                        fdt = "fdt";
                        signature-1 {
                                algo = "sha256,rsa2048";
                                key-name-hint = "fit_signing_key";
                                sign-images = "fdt", "kernel";
                        };

                };
        };
};
```

Last stage: rootFS verification

Root filesystem verification

- We have verified software up to the kernel, what about userland?
- We want to check the authenticity of the root filesystem
  - The system cannot modify the root filesystem, so it might as well be mounted read-only.
  - We need a hash of the rootFS but a rootFS is usually several orders of magnitude larger than the kernel.
- The hash will be the linchpin of the verification scheme, so it needs to be signed
  - We cannot put the signing key onto the system, so the signature must be generated off the system
- The kernel must verify the rootFS **before** it loads anything from it, so this will introduce a significant delay in the boot process.

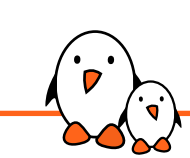
The device mapper (dm)

- The device mapper will wrap a block device and create a new block device
- It defines several types of wrapped devices, called “targets”. they include:
  - dm-crypt: admin-guide/device-mapper/dm-crypt
  - dm-integrity: admin-guide/device-mapper/dm-integrity
  - dm-verity: admin-guide/device-mapper/dm-verity

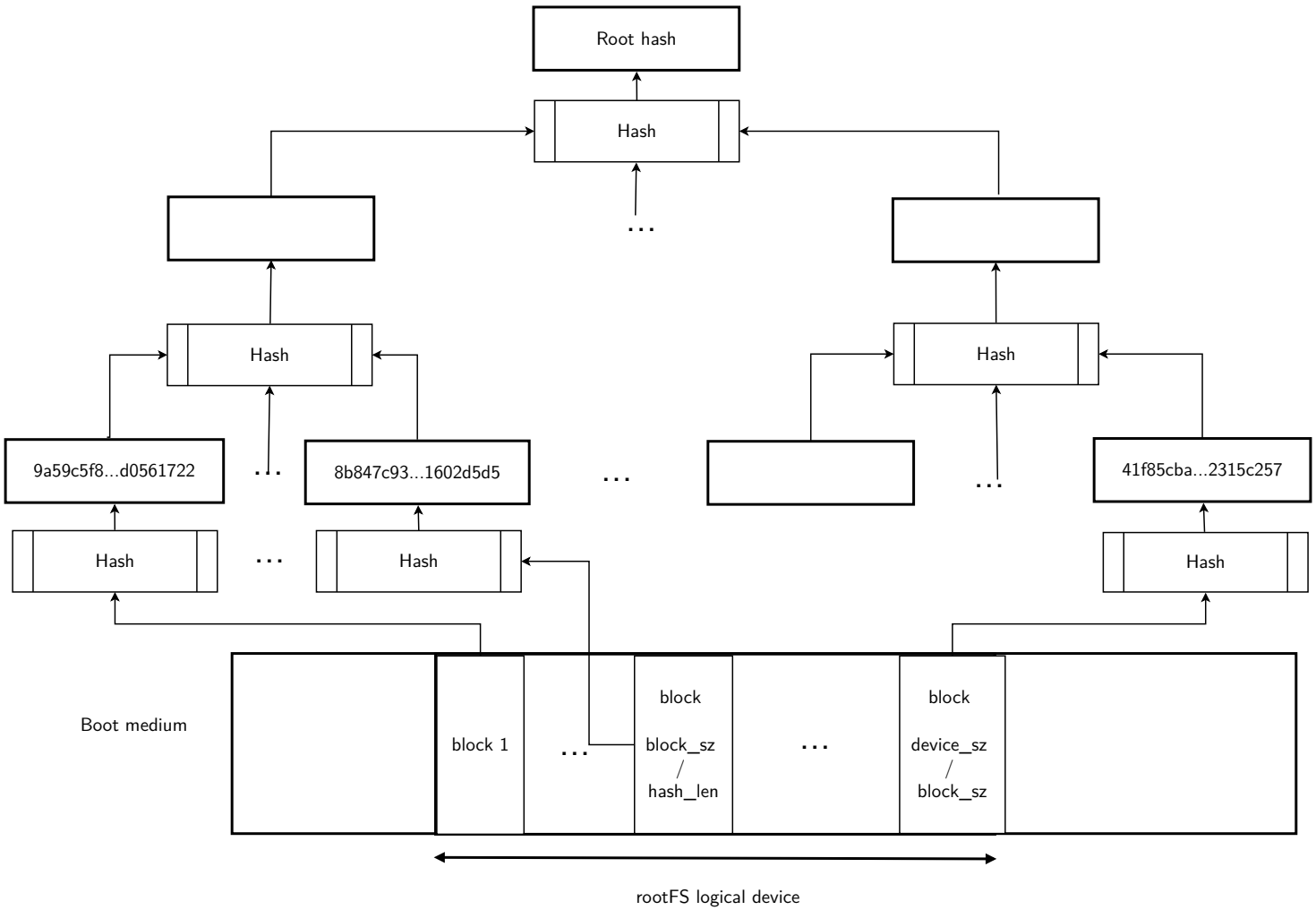
  among others
- These targets can be stacked on top of each other

dm-verity

- Read-only target of the device-mapper
- Needs the CONFIG_DM_VERITY option enabled
- Splits the block device into **blocks**
- Each block then gets hashed to give the leaves of a hash tree
- The rest of the tree is built iteratively by aggregating hashes into blocks of the data block size, and hashing them.
- After `math.ceil(math.log(num_blocks, block_size/hash_size))` iterations, we are left with one single hash: the **root hash**
- The tree can have maximum DM_VERITY_MAX_LEVELS (63 in 6.18) levels



dm-verity


Root hash
Hash
Hash
Hash
9a59c5f8...d0561722
8b847c93...1602d5d5
41f85cba...2315c257
Hash
Hash
Hash
Boot medium
block 1
block
block_sz
/
hash_len
block
device_sz
/
block_sz
rootFS logical device

dm-verity: verification

- On `read()`, dm-verity will perform a block I/O. Before completing it, it will call verity_verify_io()
- For each block of the I/O, it will
  - Lookup whether the block was already verified in the validated_blocks bitfield cache
  - If not, dm-verity will walk the hash tree down from the root, and for each level, recalculate the hashes.
  - It will then hash the **actual block data** and compare the hash to the hash from the tree that was used in the calculations.
  - The behaviour on error depends on the device configuration, but can be set to DM_VERITY_MODE_PANIC

dm-verity: setup

- To properly setup the `verity` device, the kernel will need the following parameters (described in admin-guide/device-mapper/verity):
  - `version`
  - `dev`
  - `hash_dev`
  - `data_block_size`
  - `hash_block_size`
  - `num_data_blocks`
  - `hash_start_block`
  - `algorithm`
  - `digest`
  - `salt`
- This assumes that the root hash has already been calculated, and the hash tree generated.
- These arguments can be passed on the kernel command line
- The value of the root hash is **sensitive**, but not **secret**
- Its **integrity** must be protected, not its **confidentiality**

Storing the root hash

- The parameters can be stored in an on-disk verity header
  - Modifying most parameters will be a functional problem, but not a security issue
  - This is clearly not true for the root hash
- The header needs to be signed, or at least the root hash
  - The kernel can do that, using CONFIG_DM_VERITY_VERIFY_ROOTHASH_SIG
  - It will use the kernel's trusted keyring (see security/keys/trusted-encrypted) via the sys_add_key() system call
  - The signature will be verified using:
    - the trusted keyring built into the kernel,
    - the secondary keyring if DM_VERITY_VERIFY_ROOTHASH_SIG_SECONDARY_KEYRING is enabled.
    - the platform keyring if DM_VERITY_VERIFY_ROOTHASH_SIG_PLATFORM_KEYRING is enabled

crypt/veritysetup

- Interacting with the device can be done using cryptesetup's veritysetup
  - `veritysetup format`
  - `veritysetup open`
  - `veritysetup close`
- The root hash signature can be passed using the `----root-hash-signature` option

Setup using an initramFS

- `veritysetup` is a userland tool, so it needs a userland to run
- We can use an initramFS, see filesystems/ramfs-rootfs-initramfs
- To avoid breaking our secure boot chain, it must be **signed**
- Fortunately, this is one of the images that can be included in the FIT: `FIT_RAMDISK_PROP`
- We can then even store the root hash as a file in the initramFS
- If we use kernel verification, the signature can be passed from a file

Setup at kernel init

- The kernel supports boot arguments to create mapped devices: `dm-mod.create`
- This must be enabled using CONFIG_DM_INIT
- The root hash and salt are then going to be passed as boot arguments
- Unfortunately, there is no boot argument to load the signature into the keyring, so we can no longer use CONFIG_DM_VERITY_VERIFY_ROOTHASH_SIG
- This means that the kernel command line is now security sensitive, and must be signed
  - The FIT spec include a `cmdline` property, but it is not implemented in U-Boot
  - We can use a U-Boot 'script `to set the` bootargs' environment variable. The script can be added to the FIT image as a script entry

Practical lab - Exploring Secure boot later stages

Time to setup kernel signature verification!

- Integrating the public key into the DTB
- Adding signature to the kernel FIT image
- Configuring U-Boot to enforce signature verification
- Optionally, configuring SPL to also enforce signature verification

Handling confidential information

Problem statement

- Most systems handle some type of confidential information
  - Customer/User data
  - Intellectual Property
  - Cryptographic material (private keys)
- The usual way to protect it is **encryption**
- Raises the issue of storing the **key**
- This is a hard problem on embedded systems
  - adversaries might have physical access to the system
  - unattended boot is incompatible with keys derived from user input

Hardware Security Modules (HSMs)

HSMs

- Dedicated Hardware for storing cryptographic secrets
- Implements cryptographic operations:
  - key generation
  - encryption
  - signing
- Prevent the key from leaving in cleartext form
- Some can be exported in **wrapped** (encrypted) form
- Separate usage of various keys based on PINs

HSMs: threat model

- HSMs split
  - **usage** of cryptographic material from
  - **knowledge** of the cryptographic material
- In case of compromise, this means recovery can happen without rotation
- HSMs do not necessarily prevent an adversary from **using** the key

HSMs: usage

- HSMs are usually used to store sensitive keys
  - Secure boot root keys
  - PKI root CA private keys
  - DNSSEC signature keys
  - Blockchain account private keys (cryptographic wallets)
- Some HSMs have an on-device interface (screen + buttons)
- All HSMs implement a command interface, and a PKCS#11 API

“Software HSMs”

- Strange concept at first glance
- This is the difference between “hot” and “cold” crypto wallets
- Much less secure than an actual hardware device
- Also less expensive, and potentially easier to use
- Examples:
  - SoftHSM
  - OP-TEE has a PKCS#11 TA

PKCS#11

PKCS

- Public Key Cryptography Standards
- Set of standards published by RSA Security
- Some are more relevant than others:
  - PKCS#1, or RFC 8017 is the RSA specification
  - PKCS#3 describes the Diffie-Hellman key agreement protocol
  - PKCS#7 or RFC 2315 describes the Cryptographic Message Syntax (CMS)
  - PKCS#10 is a common format for Certificate Signing Requests (CSRs)
  - PKCS#11, or “Cryptoki” is the Cryptographic Token Interface

PKCS#11

- CRYPtographic TOKen Interface
- v1.0 published by RSA Security in 1995
- Libraries implementing it are often called cryptoki
- Since 2013, overseen by an OASIS technical committee
- The specification includes C header files
- It is then up to token manufacturers to implement the API
  - Nitrokeys' nethsm-pkcs11
  - Yubico's PKCS11 module for YubiHSM
  - Thales' libCryptoki2, part of the Luna client

PKCS#11 functions

Below are some examples of functions declared in the `pkcs11f.h` header:

```
/* C_Initialize initializes the Cryptoki library. */
CK_PKCS11_FUNCTION_INFO(C_Initialize)

/* C_Finalize indicates that an application is done with the
 * Cryptoki library.
 */
CK_PKCS11_FUNCTION_INFO(C_Finalize)

/* C_GetMechanismList obtains a list of mechanism types
 * supported by a token.
 */
CK_PKCS11_FUNCTION_INFO(C_GetMechanismList)
```

PKCS#11 clients

- PKCS#11 is very specific (even implements a header)
- Token-specific implementations are usually shared objects
- So we can have a generic client compatible with all modules:
  - OpenSSL, via a provider
  - GnuTLS' p11tool

The OP-TEE PKCS#11 TA

- Implements the PKCS#11 interface over the kernel's interface to the TEE
- The "vendor" module is optee_client's `libckteec.so`
- Requires the TA to be loaded in OP-TEE
  - this a perfect job for the `C_Initialize` function.
  - Ultimately, `ckteec_invoke_init` is called:

```
CK_RV ckteec_invoke_init(void)
{
        TEEC_UUID uuid = PKCS11_TA_UUID;
...
        res = TEEC_InitializeContext(NULL, &ta_ctx.context);
        if (res != TEEC_SUCCESS) {
                EMSG("TEEC init context failed\n");
                rv = CKR_DEVICE_ERROR;
                goto out;
        }

        res = TEEC_OpenSession(&ta_ctx.context, &ta_ctx.session, &uuid,
                               login_method, login_data, NULL, &origin);
...
}
```

Cryptographic keys in the kernel

The Kernel Key Retention Service

- Also called “Kernel Key Ring Service”, or “Kernel keyring”
- Documented in security/keys/core
- Can essentially act as a software HSM
- The protection here is against adversaries that have gotten unprivileged code execution

Trusted & Encrypted Keys

- Specific key types in the Kernel keyring
- Stored in plaintext in kernel memory, only exportable wrapped
- The kernel sort of acts as an HSM for userland
- **Trusted key** encryption is hardware-backed
- **Encrypted key** encryption is backed by a kernel keyring key

Practical lab - Exploring secure key management

Time to handle confidential information!

- Provisioning keys into the i.MX93 ELE
- Using OP-TEE as software HSM
- Optionally, filesystem encryption using the ELE key
- Optionally, signing U-Boot FITs with HSM integration over PKCS#11

Userland security measures

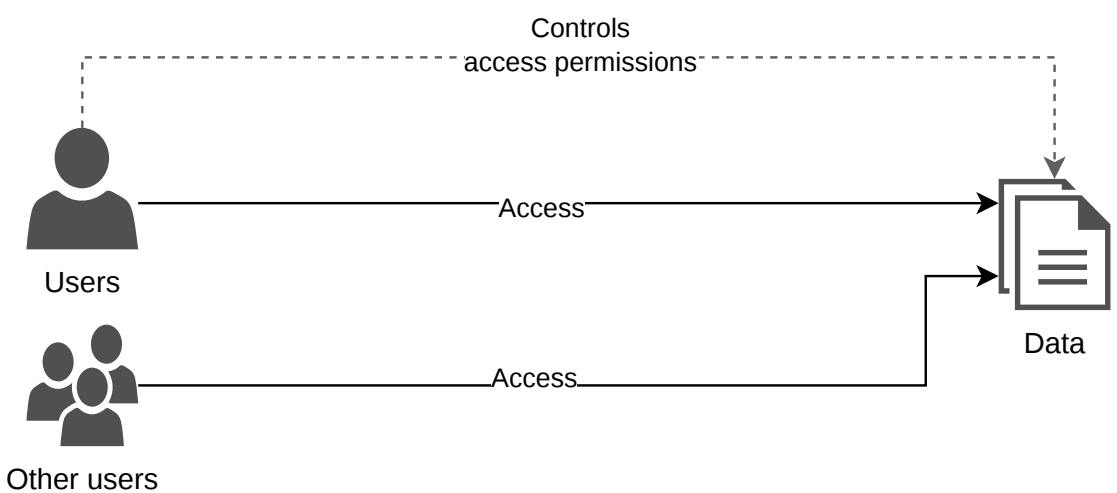
Access Control paradigms: MAC/DAC

Access control in computer security

- Access control defines relationships between two groups of entities:
  - Subjects: entities who can perform an action
  - Objects: resources that need to be controlled
- Access control makes sure only legitimate *subjects* can access an *object*
- Most systems use one of the following paradigms:
  - Discretionary access control (DAC), the most widespread on general purpose systems
  - Mandatory access control (MAC), on systems needing a more fine-grained access control

Discretionary access control

- Access control is determined by the object owner
  - Each object has an owner, e.g. the object creator.
  - The owner assigns permissions to the object, for themselves and others.
- Typical access model:
  - Access control list (ACL)-based: subjects appear in an authorization list linked with the object
  - Capability-based: subjects hold a *capability* that allows to manipulate an object



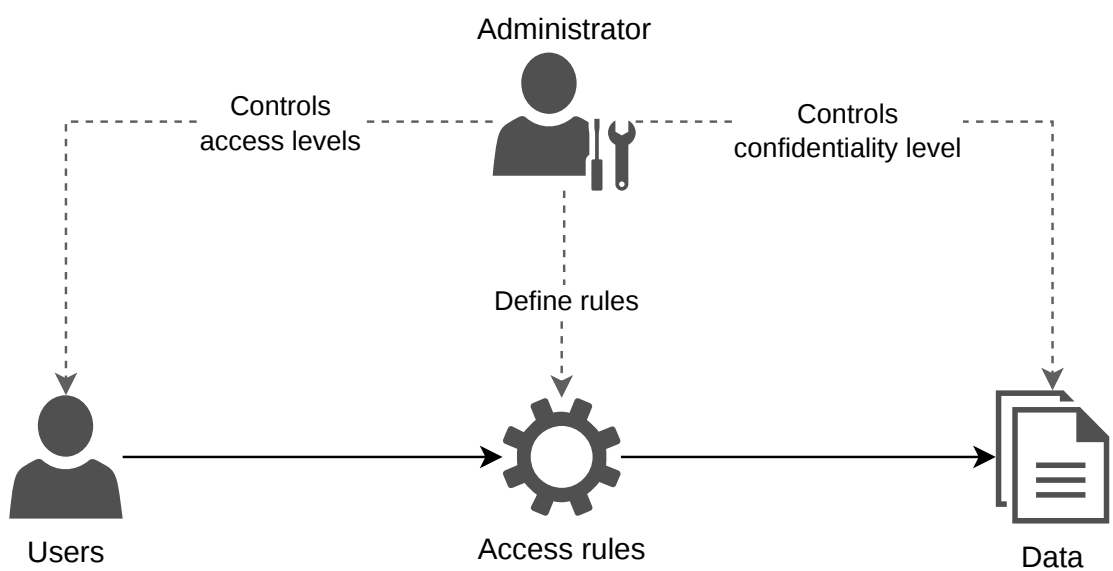
POSIX permissions

- POSIX permissions are a typical example of a DAC based on ACLs
- UNIX philosophy: everything is a file, so objects (including devices) are represented by files.
- Files metadata include:
  - Owning user and owning group
  - Permissions for each classes: owning user, owning group and others
    - Read: grants the ability to read file content
    - Write: grants the ability to write file content
    - Execute: grants the ability to execute a file, or read metadata of children files when applied on a folder.
  - Additional permission bits, such as `setuid` or `setgid`

```
$ ls -l /run/
drwxr-xr-x  2 root             root         60 Jan 22 15:51 blkid
-rw-r--r--  1 root             root          5 Jan 22 15:51 blkmapd.pid
drwxr-xr-x  3 root             lp          100 Feb  6 10:26 cups
-rw-r--r--  1 statd            nogroup       5 Jan 22 15:52 rpc.statd.pid
-rw-------  1 root             root          5 Jan 22 15:52 sm-notify.pid
```

Mandatory access control

- Access control is determined by rules
- Rules are controlled by the system administrator
- Every time a subject tries to access an object, the OS looks for a corresponding rule
- Objects can be accessed only if an existing rule allows this access
- Allows to enforce better rules, but more complex to maintain than DAC
- Three main implementations for Linux:
  - SELinux
  - AppArmor
  - TOMOYO Linux



Linux capabilities: usage and examples

Capability-based security

- Theoretical concepts:
  - Subjects have capabilities: unforgeable tokens of authority
  - Capabilities control if and how the subject can manipulate a given object
  - A given capability specifies access right on a given object.
  - Completely removes the need of ACL
  - Examples:
    - Subject *user1* can read object */etc/motd*
    - Subject *user2* can listen on *TCP port 22* object.
- POSIX standard comes with a capabilities specification, differing on various points:
  - Capabilities are not associated with objects
  - Capabilities are used in complement of ACL
  - Examples:
    - *CAP_NET_BIND_SERVICE* allows to listen on any privileged ports.

Linux capabilities

- Capabilities are a per-thread attribute, allowing to gain privileges traditionally associated with superuser
- Fully integrated since Linux 2.6.24
- Capabilities can be independently enabled and disabled
- Capabilities can be attached to executable files, pre-setting runtime capabilities
- Threads may voluntarily enable or disable a capability they are permitted to use, allowing to only use them in code paths needing it
- Threads may voluntarily preserve capabilities while executing another executable (`execve()`).

Linux capability examples

All capabilities are documented in the capabilities(7) manpage

- CAP_KILL: Bypass permission checks for sending signals
- CAP_NET_ADMIN: Perform various network-related operations
  - interface configuration, firewalling, routes, TOS, promiscuous mode, multicasting...
- CAP_NET_BIND_SERVICE: Bind a socket to Internet domain privileged ports (port numbers less than 1024).
- CAP_NET_RAW: Use RAW and PACKET sockets
- CAP_SETGID, CAP_SETUID: Make arbitrary manipulations of process GIDs/UIDs
- CAP_SETFCAP: Set arbitrary capabilities on a file.
- CAP_SYS_ADMIN Gives *a lot* of system administration related powers:
  - mounts, hostname, privileged log operations, monitoring, tracing...
- CAP_SYS_BOOT: Use reboot(2) and kexec_load(2)
- CAP_SYS_MODULE: Load and unload kernel modules
- CAP_SYS_NICE: Lower the process nice value and change the nice value for arbitrary processes, set real-time scheduling policies, set CPU affinity...
- ...

Thread capability sets

- Each thread has 5 different capability sets:
  - Controlling capabilities a thread is allowed to use:
    - **Permitted**: limits capabilities that might be in *Effective* and *Inheritable* sets. Thread can drop but never add capabilities to this set, except while using `execve()` on a file granting capabilities or with the set-uid bit.
  - Controlling capabilities effective at the moment:
    - **Effective**: used by the kernel to perform permission checks for the thread
  - Controlling capabilities present while executing a new program:
    - **Inheritable**: capabilities preserved across `execve()` on privileged process
    - **Ambient**: capabilities that are preserved across an `execve()` of a program that is not privileged. Can be used to add capabilities to an unprivileged program
    - **Bounding**: capabilities that might be gained on `execve()`. Can be used to limit capabilities that will be acquired from file capability sets

File capability sets

- Files can be associated with capabilities that will impact thread capabilities on `execve()`:
  - Permitted: capabilities automatically permitted to the thread, regardless of the thread's inheritable capabilities
  - Inheritable: capabilities that can be permitted, if they also appear in thread *Inheritable* set.
  - Effective: a single bit determining if *Permitted* set has to be copied into the thread *Effective* set.
- When executing a program with the *set-user-ID* mode bit set, file *Inheritable* and *Permitted* sets are ignored and are considered to be all ones.

Manipulating file capability sets

- `getcap` can be used to show file capabilities

```
$ getcap /usr/lib/x86_64-linux-gnu/gstreamer1.0/gstreamer-1.0/gst-ptp-helper
/usr/lib/x86_64-linux-gnu/gstreamer1.0/gstreamer-1.0/gst-ptp-helper cap_net_bind_service cap_net_admin,cap_sys_nice=ep
$ getcap -r /usr/bin
/usr/bin/clockdiff cap_net_raw,cap_sys_nice=ep
/usr/bin/dumpcap cap_net_admin,cap_net_raw=eip
```

- `gst-ptp-helper` has `CAP_NET_BIND_SERVICE`, `CAP_NET_ADMIN` and `CAP_SYS_NICE` in its *Permitted* and *Effective* sets but not in its *Inheritable* set
- `/usr/bin/clockdiff` has `CAP_NET_RAW` and `CAP_SYS_NICE` in its *Permitted* and *Effective* sets but not in its *Inheritable* set
- `/usr/bin/dumpcap` has `CAP_NET_ADMIN` and `CAP_NET_RAW` in all three sets
- `setcap` can be used to set file capabilities

```
# getcap /usr/bin/dumpcap
/usr/bin/dumpcap cap_net_admin,cap_net_raw=eip
# setcap cap_net_admin=eip /usr/bin/dumpcap
# getcap /usr/bin/dumpcap
/usr/bin/dumpcap cap_net_admin=eip
```

- Capabilities syntax is described in cap_from_text(3) manpage.

Manipulating thread capabilities I

- Capabilities can be manipulated through the `cap_t` structure.
- `cap_init()`, `cap_dup()` and `cap_free()` are used to allocate or free a `cap_t` instance.
- `cap_get_proc()` and `cap_set_proc()` allow to retrieve or apply the `cap_t` structure corresponding to the current thread.
- `cap_clear()`, `cap_set_flag()` and `cap_get_flag()` can be used to manipulate `cap_t` structure values.
- Additional functions and full behaviour are described in cap_get_proc(3) and cap_clear(3) manpages.

Manipulating thread capabilities II

```
...
cap_t caps;
const cap_value_t cap_list[2] = {CAP_FOWNER, CAP_SETFCAP};

if (!CAP_IS_SUPPORTED(CAP_SETFCAP))
     /* handle error */

caps = cap_get_proc(); if (caps == NULL)
     /* handle error */;

if (cap_set_flag(caps, CAP_EFFECTIVE, 2, cap_list, CAP_SET) == -1)
     /* handle error */;

if (cap_set_proc(caps) == -1)
     /* handle error */;

if (cap_free(caps) == -1)
     /* handle error */;
...
```

Example from cap_get_proc(3) manpage, adding `CAP_FOWNER` and `CAP_SETFCAP` effective capabilities to the calling thread.

setuid and capabilities

- The `setuid()` and `setgid()` syscalls set the effective user and group ids
  - Can be used by privileged processes to drop privileges or regain them later
- `setuid` and `setgid` flags can be added to file permissions to automatically execute them with owner user and group as effective IDs
- Securebits control the interactions of capabilities and processes with UID 0
  - `SECBIT_KEEP_CAPS`: controls if the process retains capabilities when switching to non-zero effective UID
  - `SECBIT_NO_SETUID_FIXUP`: controls if the kernel should adjust capabilities while transitioning between zero and non-zero effective UID
  - `SECBIT_NOROOT`: controls if the process is granted capabilities while executing a program with the `setuid` flag
  - `SECBIT_NO_CAP_AMBIENT_RAISE`: controls if the process can add new capabilities to the ambient set
  - Four corresponding `_LOCKED_` flags exist, preventing from ever removing the base flag

Capabilities transformation during execve()

- The capabilities(7) manpage provides a summary of capabilities transformations:

```
       P'(ambient)     = (file is privileged) ? 0 : P(ambient)
       P'(permitted)   = (P(inheritable) & F(inheritable)) |
                         (F(permitted) & P(bounding)) | P'(ambient)
       P'(effective)   = F(effective) ? P'(permitted) : P'(ambient)
       P'(inheritable) = P(inheritable)    [i.e., unchanged]
       P'(bounding)    = P(bounding)       [i.e., unchanged]

   where:
       P()    denotes the value of a thread capability set before the
              execve(2)
       P'()   denotes the value of a thread capability set after the
              execve(2)
       F()    denotes a file capability set
```

Process isolation: namespaces

Linux Namespaces

- A way to partition kernel resources
- Different groups of processes will see different resources
- In each namespaces, resources appear isolated from the other namespaces
- One fundamental principle behind software containers
- One namespace of each type is created at boot, processes can create or join different namespaces during their runtime
- Most features were introduced in Linux 3.8 (2013) or before, but some new features keep being added
- Namespace features are optional: they have to be enabled at kernel build time to be used

Linux Namespaces types I

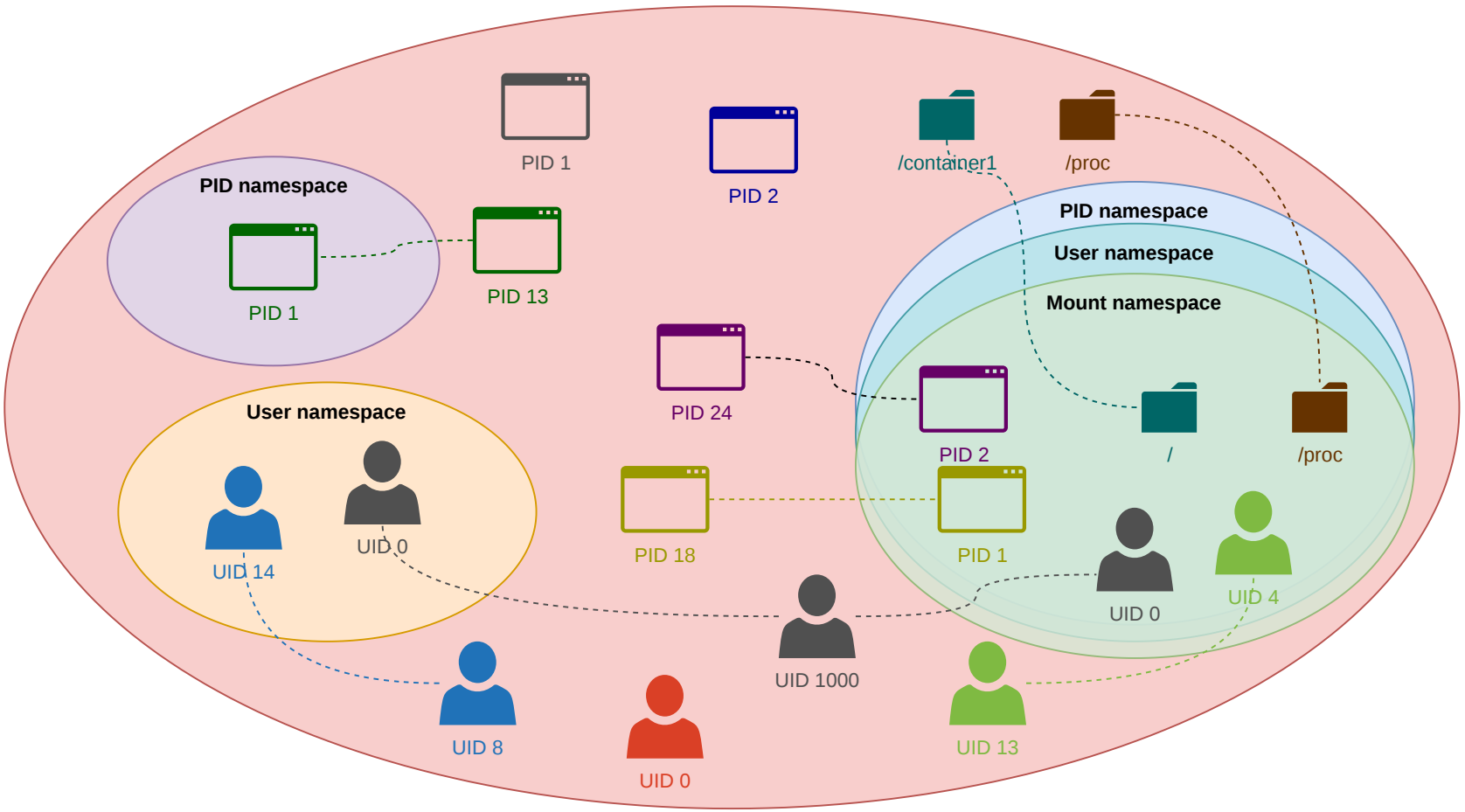
- Mount namespaces: allowing separation of the filesystem hierarchy. Different namespaces will see different mounted filesystems
- UTS namespaces: allowing separation of nodename and domainname
- IPC namespaces: allowing separation of some IPC resources, such as SysV `shmget()` IPC mechanism
- PID namespaces: allowing separation of PIDs. Each namespace can reuse the same PID for different processes, e.g. PID 1.
  The PID will still appear in the host namespace but with a different PID
- Network namespaces: allowing separation of networking resources. Each namespace had a different network configuration: devices, IP addresses, routing, firewalling...

Linux Namespaces types II

- User namespaces: allowing to isolate user ids and group ids. Ids inside and outside of a namespace can be different, allowing an unprivileged user to have the ID 0 inside of the namespace, i.e. being able to make privileged operations on the namespace resources.
- Control group namespaces (introduced in Linux 4.6): allowing cgroup hierarchies isolation
- Time namespace (introduced in Linux 5.6): allowing namespaces to see a different system time

Namespaces example

- Namespaces are often combined together



Namespace related syscalls (1)

- Processes can change namespace during their runtime
- Various syscalls exist to create or join namespaces
- They all take a flag to describe the types of namespaces to manipulate: `CLONE_NEWNS` (mount namespaces), `CLONE_NEWUTS`, `CLONE_NEWIPC`, `CLONE_NEWPID`, `CLONE_NEWNET`, `CLONE_NEWUSER`, `CLONE_NEWCGROUP`, `CLONE_NEWTIME`
- Namespaces are deleted automatically when no process is running in them and nobody is using a corresponding `proc/pid/ns/` link file

Namespace related syscalls (2)

- `clone()`:
  - Namespace related flags can be specified
  - The newly created child process will be in different namespace
- `unshare()`:
  - Create a new namespace without forking
  - Also wrapped by the `unshare` command-line tool
- `setns()`:
  - Join an existing namespace
  - Pointing to a specific namespace:
    - First argument: a link file in `proc/pid/ns/`
    - Second argument: a single flag describing the namespace type
  - Pointing to a running process:
    - First argument: a PID file descriptor obtained with `pidfd_open()`
    - Second argument: a bitmask of flags describing the namespaces to join
  - Also wrapped by the `nsenter` command-line tool

Process isolation: cgroups

Linux cgroups

- Control groups are a way to control resource usages by a group of processes
- Resources might include CPU time, RAM, disk...
- Such possibilities existed before cgroups, but were enforced per process
- Control groups allow to:
  - Limit resources allocated to a group: CPU time, CPU set, memory, number of file descriptors...
  - Prioritize one group over another one
  - Measure resource usage, without enforcing any particular limit
- cgroup v2 was introduced in Linux kernel 4.5 (2016) with breaking changes and a new configuration interface

Cgroup interface: creating cgroups

- cgroups are configured through a virtual filesystem, generally mounted on `/sys/fs/cgroup`
- A new cgroup can be created by creating a new folder in `/sys/fs/cgroup` or an already existing subfolder

```
mkdir /sys/fs/cgroup/mygroup
```

- Processes can be assigned to a particular cgroup by writing their PID to the `cgroup.procs` file:

```
echo 1234 > /sys/fs/cgroup/mygroup/cgroup.procs
```

- cgroups with no associated processes can be removed by removing the associated folder in `/sys/fs/cgroup`

```
rmdir /sys/fs/cgroup/mygroup
```

Cgroup interfaces: controllers I

- Enforcing limits or accessing statistics of a group can be done using files of the associated folder in `/sys/fs/cgroup`

```
cat /sys/fs/cgroup/mygroup/memory.current
45056
echo 10000 > /sys/fs/cgroup/mygroup/memory.max
```

- List of all supported controllers can be found in kernel documentation: Documentation/admin-guide/cgroup-v2.rst
- A few examples:
  - cpu.stat: read-only file of CPU usage statistics
  - cpu.max: maximum CPU bandwidth limit
  - memory.current: read-only file of current memory usage
  - memory.max: memory usage hard limit
  - io.stat: read-only file of I/O statistics
  - pids.max: Hard limit of number of processes
  - cpuset.cpus: CPUs to be used by tasks of this group

Cgroup interfaces: controllers II

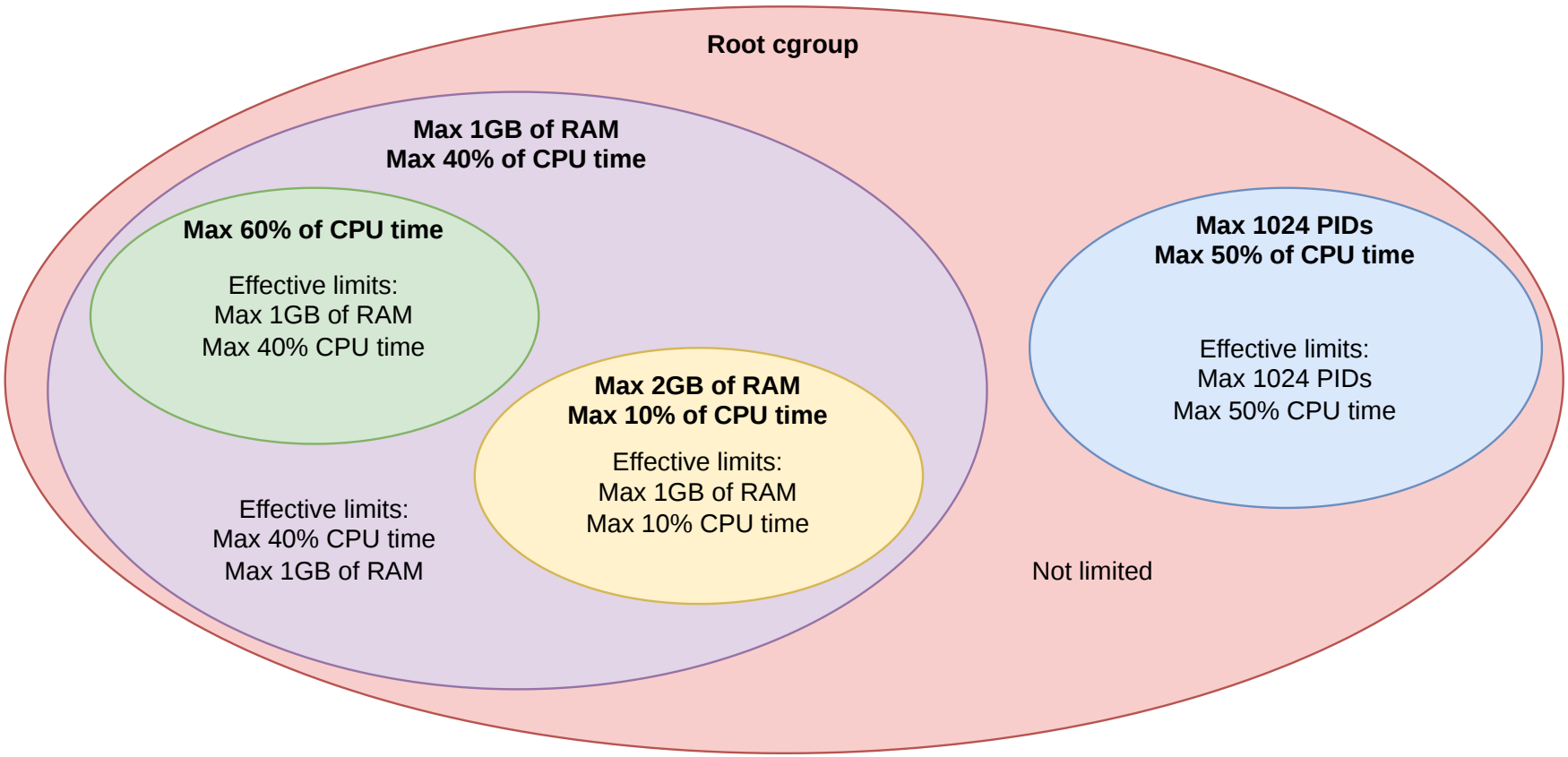
- Parent cgroups control whether controllers will be present in their childs through the `cgroup.subtree_control` control file:

```
# echo +cpu -memory -pids > /sys/fs/cgroup/mygroup/cgroup.subtree_control
# cat /sys/fs/cgroup/mygroup/cgroup.subtree_control
cpu
# cat /sys/fs/cgroup/mygroup/mysubgroup/cgroup.controllers
cpu
bootlin-mathieu# ls /sys/fs/cgroup/mygroup/mysubgroup/pids* | wc -l
0
```

- Some statistics files will still be present when the controller is disabled:

```
#ls -l /sys/fs/cgroup/mygroup/mysubgroup/memory.*
/sys/fs/cgroup/mygroup/mysubgroup/memory.pressure
```

Cgroups example



Process isolation: seccomp

Linux seccomp

- The linux kernel exposes hundreds of syscalls, but most applications only need a few of them
- Secure Computing (seccomp) is a kernel feature, allowing to restrict the syscalls that a process can use
- Processes can voluntarily use the seccomp syscall, restricting themselves to use only a few system calls:
  - `read()`
  - `write()`
  - `exit()`
  - `sigreturn()`
- If the process later tries to use another syscall, it will be killed by the kernel
- The seccomp syscall is not wrapped by the glibc: `syscall()` must be used

```
syscall(SYS_seccomp, SECCOMP_SET_MODE_STRICT, 0, NULL);
```

Seccomp filters

- Some programs might need more than just reading and writing already opened files
- Seccomp filters were introduced with Linux 3.5
- Processes can install a custom BPF program that will filter authorized syscalls

```
struct sock_fprog prog = { /* BPF program description */ };
syscall(SYS_seccomp, SECCOMP_SET_MODE_FILTER, 0, &prog);
```

- Supported filters are described in seccomp(2) manpage.
- This possibility is used by various applications: openSSH, Systemd, sudo, Firefox, QEMU...

Using Seccomp filters

- Some care must be used when selecting syscalls to filter:
  - Most syscalls are abstracted by the libc wrappers: the underlying syscall might be different than expected
  - The same wrapper might used different syscalls on different architectures or different libc versions
- Additionally, writing BPF code can be tricky
- The *libseccomp* library provides a higher level abstraction
  - Easier to use
  - Function based filtering
  - Platform independent

Going further on process isolation

- Namespaces:
  - A namespaces article series: https://lwn.net/Articles/531114/
- Control groups:
  - A control groups article series: https://lwn.net/Articles/604609/
- seccomp:
  - A seccomp overview: https://lwn.net/Articles/656307/
  - About seccomp caveats: https://lwn.net/Articles/738694/

Linux Security Modules: SELinux

Security-Enhanced Linux (SELinux)

- Originally developed by the National Security Agency of the United States
- Part of mainline Linux since 2.6.0 in 2003
- Based on the Linux Security Module (LSM) framework of the kernel
- Introduces mandatory access control for userspace components
- Additional access control: traditional access control list mechanism is still present
- SELinux can be used to precisely control which activities a system allows each user, process, and daemon
- Provided by all major distributions, Yocto and Buildroot
- Used on all Android devices

SELinux context

- Each object (files, users, processes) has a context composed by 3 or 4 fields:
  - user
    - An identifier, different from POSIX user
    - Determines which roles can be used
    - Each POSIX user is mapped to only one SELinux user
    - SELinux users can be shared among several POSIX users
    - Generally used to represent a class of users
    - E.g. `user_u`, `staff_u`, `sysadm_u`
    - Mapping between system users and SELinux users can be shown with `semanage login`:

```
# semanage login -l

Login Name           SELinux User         MLS/MCS Range        Service

__default__          unconfined_u         s0-s0:c0.c1023       *
root                 unconfined_u         s0-s0:c0.c1023       *
sddm                 xdm                  s0-s0                *
```

SELinux context

- Each object (files, users, processes) has a context composed by 3 or 4 fields:
  - user
    - An identifier, different from POSIX user
    - Determines which roles can be used
  - role
    - Determines what domains can be accessed
    - Each SELinux user can play a fixed set of roles
    - E.g. `system_r`, `staff_r`
    - List of possible roles for a user can be seen with `seinfo`:

```
# seinfo -uunconfined_u -x

Users: 1
    user unconfined_u roles { system_r unconfined_r } level s0 range s0 - s0:c0.c1023;
```

SELinux context

- Each object (files, users, processes) has a context composed by 3 or 4 fields:
  - user
    - An identifier, different from POSIX user
    - Determines which roles can be used
  - role
    - Determines what domains can be accessed
    - Each SELinux user can play a fixed set of roles
  - domain or type
    - Defines the security context
    - Most SELinux rules will rely on it
    - E.g. `bin_t`, `httpd_t`, `my_application_t`
    - List of types for a given role can be seen with `seinfo`

```
# seinfo -rstaff_r -x

Roles: 1
   role staff_r types { auditadm_screen_t bluetooth_helper_t chfn_t chkpwd_t
   chromium_naclhelper_t chromium_renderer_t chromium_sandbox_t chromium_t
   container_engine_t container_kvm_t container_t crio_t ddclient_t dirmngr_t
   dockerc_user_t dockerd_t dockerd_user_t evolution_alarm_t evolution_exchange_t
   evolution_server_t evolution_t evolution_webcal_t exim_t games_t gconfd_t gpg_agent_t
    ...
   };
```

SELinux context

- Each object (files, users, processes) has a context composed by 3 or 4 fields:
  - user
    - An identifier, different from POSIX user
    - Determines which roles can be used
  - role
    - Determines what domains can be accessed
    - Each SELinux user can play a fixed set of roles
  - domain or type
    - Defines the security context
    - Most SELinux rules will rely on it
  - range (optional)
    - Sometimes referred as *security level*
    - Can be used as part of *Multi-level security* or *Multi-category security*
  - This context is generally represented as a single string:
    - `user:role:type[:range]`

Multi-Level and Multi-Category Security

- Multi-Level Security (MLS)
  - Allows a hierarchical structure representing different level of sensitivity
  - Levels can be used to represent data classification
    - Unclassified, Restricted, Confidential, Secret...
- Multi-Category Security (MCS)
  - Allows to compartment data with different categories
  - Categories can be used to represent different departments
- SELinux allows to mix both levels and categories

SELinux Multi-Level and Multi-Category Security

- Context *range* can be composed by two parts:
  - The sensitivity level, as an integer
    - Order is defined by the *dominance*, s0 is generally the lowest
    - For MCS, only one level is used: typically s0
  - Optionally, the category set, as integers
- Ranges are represented as strings
  - `s0`: sensitivity level 0
  - `s1:c4,c7`: sensitivity level 1, categories 4 and 7
- Domains will be affected a clearance
  - Determines which level and categories can be accessed
  - `s2:c1.c4,c7`: sensitivity level 2, categories 1 to 4 and 7
  - *dominance* determinates other allowed security levels

SELinux file context

- On creation, files and directories will by default inherit the context of parent folder
- `ls -Z` can be used to show file context:

```
# ls -lZ /
lrwxrwxrwx.   1 root root system_u:object_r:bin_t:s0                  7 Feb 19 08:09 bin -> usr/bin
drwxr-xr-x.   3 root root system_u:object_r:boot_t:s0              4096 Feb 19 08:11 boot
drwxr-xr-x.  19 root root system_u:object_r:device_t:s0            3300 Feb 19 08:29 dev
drwxr-xr-x.  78 root root system_u:object_r:etc_t:s0               4096 Feb 19 08:28 etc
drwxr-xr-x.   3 root root system_u:object_r:home_root_t:s0         4096 Feb 19 08:11 home
...
```

- `chcon` can be used to change file context:

```
# ls -lZ hello.sh
-rwxr-xr-x. 1 root root unconfined_u:object_r:user_home_t:s0 23 Feb 19 08:29 hello.sh
# chcon system_u:object_r:bin_t:s0 hello.sh
# ls -lZ hello.sh
-rwxr-xr-x. 1 root root system_u:object_r:bin_t:s0 23 Feb 19 08:29 hello.sh
```

- `restorecon` can be used to set file context to policy default values

SELinux process context

- On creation, processes will by default inherit the context of parent process
- `id -Z` can be used to show own context in a shell:

```
# id -Z
unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
```

- `ps -Z` can be used to show processes contexts:

```
# ps -Z
LABEL                                PID TTY          TIME CMD
unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023 738 pts/0 00:00:00 bash
unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023 1608 pts/0 00:00:00 ps
```

- `runcon` can be used to start a process with a different context:
  - The transition must be allowed by SELinux policy rules

```
# id -Z
unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
# runcon -r system_r bash
# id -Z
unconfined_u:system_r:unconfined_t:s0-s0:c0.c1023
```

SELinux modes

- SELinux can be used in two different modes
  - Permissive: rules are not enforced, but all violations are logged
    - Can be useful during the configuration phase to ensure all needed rules have been set.
    - The `audit2allow` tool can then be used to generate some rules based on this log
  - Enforcing: rules are strictly enforced
- Mode can be configured in `/etc/selinux/config` or with `setenforce`
- Additionally, SELinux can be completely disabled
- Active mode can be seen with `sestatus`

```
# sestatus
SELinux status:                 enabled
SELinuxfs mount:                /sys/fs/selinux
SELinux root directory:         /etc/selinux
Loaded policy name:             default
Current mode:                   permissive
...
```

SELinux policies

- Policies will describe what is permitted on the system
  - By default, everything is forbidden
  - Rules will allow processes to use objects, based on the context of both the source and the target
- Policies can be dynamically loaded during runtime
- Policies will have to adapt to system needs:
  - They can only isolate a few processes
  - They can implement a full multi-level security system
  - Or implement anything in-between

SELinux rules

- SELinux rules are based on a *access vector* describing the access, composed by:
  - The source context
  - The target context
  - The class of the target
    - Describes the type of the resource
    - E.g. `file`, `socket`, `process`, `dbus`
  - the permission or activity
    - Describes the action the access tries to make
    - Possible values depend on the class
    - E.g. `create`, `read`, `write`, `execute`, `bind`, `connect`

SELinux rules examples

- Most SELinux rules rely on the type, but other context fields can also be used
- The most common rule is `allow`, allowing a specific access

```
allow user_t lib_t : file { execute };
```

- Allows processes from `user_t` domain
- To `execute files`
- Of the `lib_t` type
- Other types of rules exist, such as `type_transition`, allowing process transition to a different context:

```
type_transition init_t initrc_exec_t : process initrc_t;
```

- Processes running in `init_t` domain
- Executing a file with `initrc_exec_t` type
- Shall transition to the `initrc_t` domain

Listing SELinux rules

- The `sesearch` command can be used to query rules present on the system
  - Filters can be added on the type of rules:
    - `–allow`
    - `–role_transition`
    - ...
  - Filters can also be added on context fields:
    - `-s`: source context
    - `-t`: target context
    - `-c`: object class
    - ...

```
# sesearch --allow --source wireshark_t --target proc_net_t
allow wireshark_t proc_net_t:dir { getattr ioctl lock open read search };
allow wireshark_t proc_net_t:file { getattr ioctl lock open read };
allow wireshark_t proc_net_t:lnk_file { getattr read };
```

SELinux policy modules

- SELinux comes with the concept of modules, providing rules
- Policy modules can be dynamically loaded or unloaded with `semodule`
  - `–list-modules`
  - `–enable`
  - `–disable`

```
# semodule --list-modules | head -5
accountsd
acct
afs
aide
alsa
# semodule --disable alsa
libsemanage.add_user: user sddm not in password file
root@setest:~# semodule --list-modules | grep alsa
```

- Generic policies might be provided by projects, distributions or the SELinux refpolicy project

Creating SELinux policies

- You will sometimes need to write your own policies
  - For your own custom context
  - For a project you are maintaining
- Typical policy modules will consist of:
  - A `.te` file containing policy rules for your application, such as `allow` or transition rules
  - A `.if` file defining the interfaces: policy macros used by other policy modules to interact with this policy
  - A `.fc` file defining application security contexts: instructions for labeling files related to the application

Creating SELinux policies: sepolicy generate

- The `sepolicy generate` command can be used to create module template files
- SELinux can be generated from the above files with `checkmodule` and `semodule_package` or more easily with some helper script

```
# mkdir myapp && cd myapp
# sepolicy generate --init -n myapp /bin/myapp
Failed to retrieve rpm info for selinux-policy
Created the following files:
/root/myapp/myapp.te # Type Enforcement file
/root/myapp/myapp.if # Interface file
/root/myapp/myapp.fc # File Contexts file
/root/myapp/myapp_selinux.spec # Spec file
/root/myapp/myapp.sh # Setup Script
# ... Define your custom rules ...
# ./myapp.sh
# semodule -l | grep myapp
myapp
```

- In most cases, audit2allow can help you to define those rules

Linux Security Modules: AppArmor

AppArmor

- Developed by Immunix in 1998
- Supported by Canonical since 2009
- Allows limiting program capabilities with profiles
- Provides an alternative to SELinux:
  - Allows to introduce MAC in Linux systems
  - Based on the Linux Security Module (LSM) framework of the kernel

AppArmor differences with SELinux

- Files are identified by their path instead of attaching *security labels* to inodes
  - Creating a hardlink to a restricted file might help to bypass restrictions
- As no data is stored in file inodes, the configuration is more centralized
- AppArmor only supports about 20 *Access Modes*:
  - SELinux supports hundreds of different permissions, depending on the type of object
  - Basic modes: `read`, `write`, `append`
  - Execution mode, with various variants
  - Link files and lock files management
- There is no support for multi-level security
- Overall, AppArmor tends to provide less advanced features but to be also easier to use

AppArmor profile files

- AppArmor relies on *Profiles* to describe applications confinement
- Simple text files stored in `/etc/apparmor.d`, one file per binary
  - Files are named to reflect binary path, `/` being replaced by `.`
  - E.g. `/etc/apparmor.d/bin.ping` for `/bin/ping`
- Profiles will contain rules:
  - Paths of files that can be accessed
  - Capabilities that can be used
- `aa-genprof` can be used to automatically create a new profile
  - Target application is launched, all actions are logged
  - The user is then prompted for actions that need to be allowed by profile rules
  - Similarly, `aa-logprof` can be used to interactively add rules from audit logs

AppArmor profile example

- `/etc/apparmor.d/bin.ping` content:

```
abi <abi/4.0>,

include <tunables/global>
profile ping /{usr/,}bin/{,iputils-}ping flags=(complain) {
  include <abstractions/base>
  include <abstractions/consoles>
  include <abstractions/nameservice>

  capability net_raw,
  capability setuid,
  network inet raw,
  network inet6 raw,

  /{,usr/}bin/{,iputils-}ping mixr,
  /etc/modules.conf r,
  @{PROC}/sys/net/ipv6/conf/all/disable_ipv6 r,
}
```

- This profile can be used by `ping` and `iputils-ping` binaries
- Allows to use `net_raw` and `setuid` capabilities
- Add m (executable mapping), ix (inherit execute mode) and r (read) access modes to the `ping` binary
- Add r (read) access mode to `/etc/modules.conf` and `/proc/sys/net/ipv6/conf/all/disable_ipv6` files

AppArmor base commands (1)

- `aa-status`
  - Shows current AppArmor status

```
# aa-status
apparmor module is loaded.
126 profiles are loaded.
6 profiles are in enforce mode.
  /usr/bin/man
  lsb_release
  ...
44 profiles are in complain mode.
  Xorg
  avahi-daemon
  dnsmasq
  ...
```

- `aa-complain` and `aa-enforce`
  - Either enter complain (audit) or enforce mode for a given profile

```
# aa-enforce /bin/ping
```

AppArmor base commands (2)

- `apparmor_parser`
  - Load or reload a profile file

```
# apparmor_parser -r /etc/apparmor.d/bin.ping
```

- `aa-exec`
  - Execute a program with non-default profile

```
# aa-exec -p unconfined -- ping bootlin.com
```

AppArmor audit log

- AppArmor log will list rules violations
  - We can test with a modified *ping* profile, with `net_raw` capability removed

```
# aa-enforce /bin/ping
# ping -N name bootlin.com
ping: socktype: SOCK_RAW
ping: socket: Permission denied
# grep success=no /var/log/audit/audit.log
type=SYSCALL msg=audit(1771853258.629:191): arch=c000003e syscall=41 success=no exit=-13 a0=2 a1=3 a2=1 a3=6 items=0 ppid=849 pid=869
auid=0 uid=0 gid=0 euid=0 suid=0 fsuid=0 egid=0 sgid=0 fsgid=0 tty=pts0 ses=1 comm="ping" exe="/usr/bin/ping" subj=ping
key=(null)ARCH=x86_64 SYSCALL=socket AUID="root" UID="root" GID="root" EUID="root" SUID="root" FSUID="root" EGID="root" SGID="root"
FSGID="root"
```

Application hardening via systemd

Systemd

- Modern *init* system used by almost all Linux desktop/server distributions
- Provides features such as
  - Parallel startup of services, taking into account dependencies
  - Monitoring of services
  - On-demand startup of services, through *socket activation*
  - Resource-management of services: CPU limits, memory limits
- Configuration based on *unit files*
  - Declarative language, instead of shell scripts used in other init systems

Capabilities-related settings

- Systemd execution units allow to specify capabilities related settings
  - `CapabilityBoundingSet=` controls which capabilities should be in the process bounding set
  - `AmbientCapabilities=` controls which capabilities should be in the process ambient set
  - `SecureBits=` controls which Securebits should be set
  - `NoNewPrivileges=` is a boolean value, controlling if `PR_SET_NO_NEW_PRIVS` flag should be applied with `prctl()`. If set, ensure no new privilege is ever gained through `execve()`: effective UID and GID are not affected by `setuid` and `setgid` bits, capabilities cannot be added.

SELinux and AppArmor control

- Systemd execution units allow to control SELinux context or AppArmor profile
  - `SELinuxContext=` Sets the SELinux security context, overriding the default transition. The transition must be allowed by the SELinux policy.
  - `AppArmorProfile=` Sets the AppArmor profile to use. The profile must already be loaded in the kernel.

Process sandboxing

- Systemd execution units allow to sandbox processes
  - Can be used to limit the system exposure
  - Sandboxing options relies on various kernel features: seccomp, namespaces...
  - Highly simplifies usage of various security features
- All options are described in the *SANDBOXING* section of the systemd.exec(5) manpage
- A good practice is to enable as much as possible of these options

Process sandboxing examples

- Data access:
  - Some directories can be made inaccessible, read-only, or replaced by a temporary empty folder
- Device access:
  - `/dev` can be replaced by a folder with only a few pseudo devices, such as `/dev/null` or `/dev/zero`
  - A separate network namespace or a completely isolated network can be used. Alternatively, communication can be restricted to some socket families
- System configuration:
  - Various parts of `/proc` can be made read-only
  - Kernel modules loading can be blocked
  - Realtime scheduling can be limited
- Dedicated namespaces can be used: IPC, PID, UTS, clocks...

Syscall filtering with Systemd

- Systemd allows to filter syscalls used by launched process
- Relies on `seccomp` kernel feature
- The `SystemCallFilter=` settings can be used to define the list of allowed or forbidden syscalls
- As for using seccomp directly, remember getting the correct list of syscalls might be tricky
- Syscalls are grouped in predefined sets that can be used instead of listing them individually:
  - `@basic-io`, system calls for basic I/O: `read()`, `write()` and related calls
  - `@mount`, mounting and unmounting of file system: `mount()`, `chroot()`, and related calls
  - `@reboot`, System calls for rebooting and reboot preparation: `reboot()`, `kexec()`, and related calls
  - ...

Systemd resource control

- Systemd allows to limit process resources, relying on Linux cgroups
  - Controlling CPU usage: `CPUAccounting=`, `CPUQuota=`, `AllowedCPUs=`...
  - Controlling memory usage: `MemoryAccounting=`, `MemoryMin=`, `MemoryHigh=`, `MemoryMax=`...
  - Controlling number of tasks: `TasksAccounting=`, `TasksMax=`
  - Controlling I/O throughput: `IOAccounting=`, `IOReadBandwidthMax=`, `IOReadIOPSMax=`...
  - Controlling network usage: `IPAccounting=`, `SocketBindAllow=`, `RestrictNetworkInterfaces=`...
- All settings are described in systemd.resource-control(5) manpage.

Systemd service example: openvpn@.service

```
[Unit]
Description=OpenVPN connection to %i
...

[Service]
Type=notify
PrivateTmp=true
WorkingDirectory=/etc/openvpn
ExecStart=/usr/sbin/openvpn --daemon ovpn-%i --status /run/openvpn/%i.status 10 --cd /etc/openvpn --config /etc/openvpn/%i.conf
    --writepid /run/openvpn/%i.pid
PIDFile=/run/openvpn/%i.pid
KillMode=process
CapabilityBoundingSet=CAP_IPC_LOCK CAP_NET_ADMIN CAP_NET_BIND_SERVICE CAP_NET_RAW CAP_SETGID CAP_SETUID CAP_SETPCAP CAP_SYS_CHROOT
    CAP_DAC_OVERRIDE CAP_AUDIT_WRITE
TasksMax=10
DeviceAllow=/dev/null rw
DeviceAllow=/dev/net/tun rw
ProtectSystem=true
ProtectHome=true
RestartSec=5s
Restart=on-failure
```

Practical lab - Exploring userland security measures

Time to restrict userland applications!

- Preventing a simple application from executing shellcode using **SECCOMP**
- Manipulating **SELinux** contexts
- Using **systemd** to restrict a daemon's access to resources

Measured boot

Concept

Measured boot

- Another way of establishing **trust**
- Idea: measure characteristics of the system and record the measurement
- One major target for measurement is the software being run
- Other targets include:
  - configuration (e.g. U-Boot environment)
  - boot parameters (e.g. kernel command line)
- Also called **trusted** boot != secure/verified boot

Differences from Secure Boot

- The software is not **authenticated**, only **measured**
- To be useful, the measurement needs to be used to make some decision
- This is a different step called **attestation** or **appraisal**
- It can only consist of recording the measurement discrepancy, or more complex actions

Measurements

- A measurement is a hash of the data we are trying to measure
- We need to use a cryptographically secure hash function
- In a later step, we'll want to compare this hash to a known-good value
- We need to store these hashes

Measured boot support

- A lot of the software components we've already covered support Measured boot:
  - TF-A
  - U-Boot
- systemd supports measured boot using systemd-measure
  - this feature is experimental
  - currently only supports UKI, which are mostly used in UEFI boots

Trusted Platform Module (TPM)

Trusted Platform Module

- Dedicated and isolated crypto component
- Following the Trusted Computing Group (TCG)'s specification:
  - TPM 1.2, originally published in 2003
  - TPM 2.0 Library, also known as ISO/IEC 11889
- In 1.2, the TPM was explicitly specified as a hardware component including a crypto co-processor among others.
- The current 2.0 spec explicitly drops this requirement

```
Another reasonable implementation of a TPM is to have the code run on
  the host processor while the processor is in a special execution mode
```

TPM 2.0

- Current version of the specification
- This is e.g. the one that is a Windows 11 requirement
- Specifies the TPM as a *library*
- The TCG's Brief overview recognizes 5 types of TPM:
  - Discrete TPMs, or dTPM: dedicated hardened chip
    example: the ST33KTPM2X
  - Integrated TPMs: co-processor. Examples:
    - Microsoft's Pluton
    - The AMD Secure Processor (ASP, or PSP)
  - Firmware TPMs, or fTPMs: hardware-isolated software. Examples:
    - the OP-TEE fTPM
  - Software TPMs: only useful for testing
    example: this reference implementation from Microsoft.
  - Virtual TPMs, or vTPMs: basically an fTPM running in a hypervisor
    Google implemented one for GCP

Using the TPM

- Key storage and generation
- Platform Configuration Registers (PCRs)
  - "Shielded Locations", meaning hardware protected by the TPM
  - Meant to contain a log of measurements
  - Some persist across reboot, most should be reset to initial value
  - Initial value must be all 0s or all 1s, except for PCR[0], which can be used as a locality indicator
  - Cannot be set, only **reset** or **extended**
    - This means hashing the current value of the PCR appended with the measurement
    - The order of measurement therefore impacts the final PCR value

Additional references

- The Open Security Training 2 “Trusted Computing” track: TC1101, TC1102, TC2202
- The Open Access Practical Guide to TPM 2.0

IMA/EVM

Integrity Measurement Architecture (IMA)

- IMA is a Linux kernel subsystem, found under `security/integrity/ima`
- Uses extended attributes (filesystems/ext4/attributes.html) to store measurement of files at runtime.
- Enabled via CONFIG_IMA
- Continues the measurement process after the kernel takes over
- Focuses on the integrity of file **contents**

Integrity Measurement Architecture Appraisal

- IMA appraisal validates the integrity of file contents
- File content is verified either with:
  - A hash: file content is protected against offline modifications
  - A signature: file content is protected against both offline and online modifications
- Only the file content is validated: nothing prevent from renaming a file or replacing it with another valid file
- IMA appraisal behaviour is controlled by the `ima_appraise` kernel command line parameter:
  - `enforce` appraisal is fully enabled: access to file with invalid or missing hashes of signatures is denied
  - `log` access to file with invalid or missing hashes or signature is allowed, but logged
  - `fix` hashes of files covered by the policy are updated, when a file is accessed
  - `off`: appraisal is completely disabled: hashes or signature are neither generated nor validated

Integrity Measurement Architecture Policies

- Policies defines what is measured
- Controlled by the `ima_policy` kernel command line parameter
- Several policies can be combined
- IMA comes with predefined policies:
  - Controls what is measured
  - `tcb`: measures all executed programs, files mmap'd for execution, and all files read with uid or effective uid set to 0
  - `appraise_tcb` appraises the integrity of all files owned by root
  - `secure_boot` appraises the integrity of files based on file signatures
  - `fail_securely` always force file signature verification
  - `critical_data` measures kernel integrity critical data
- Custom policies can be defined
  - Only well-known stable files should be measured
    - Binaries, libraries, configuration...
  - Files with frequent modifications should not be measured
    - Logs, databases...

Extended Verification Module (EVM)

- EVM is also a Linux kernel subsystem, found under `security/integrity/evm`
- Uses extended attributes (filesystems/ext4/attributes.html) to store measurement of files at runtime.
- Enabled via CONFIG_EVM
- Continues the measurement process after the kernel takes over
- Focuses on the integrity of file **metadata**

Extended Verification Module (EVM)

- Similarly to IMA, EVM is split into:
  - EVM HMAC: allows to protect against offline modifications
  - EVM Signature: also protects against runtime modifications
- EVM can be enabled with the `securityfs` pseudo-filesystem: `/sys/kernel/security/evm`:
  - Value is a bit mask of enabled configuration, bits can only transition from 0 to 1
  - bit 0: Enable HMAC validation and creation
  - bit 1: Enable digital signature validation
  - bit 31: Disable further runtime modification of EVM policy
- As for IMA, nothing prevent from renaming the file: filename is not part of the metadata

IMA/EVM vs dm-verity

- Both aim to enforce userland integrity
- IMA/EVM enables:
  - remote attestation
  - auditing
- This comes at the cost of a more complicated setup
- IMA policies must be crafted very carefully
- Neither is very useful without a full secure boot chain

Maintenance, regulation and compliance

Rationale

- “Cybersecurity” has been a rapidly growing concern
- The stakes are high:
  - Reputation
  - Downtime
  - Money (blackmail, ransomware)
- You can actually buy cybersecurity insurance
- Some vulnerabilities have wide-reaching consequences for everyone
- Predictably, this has led to produce standards and legislation

Examples

- standards
  - ISO 27001
  - NIST standards, such as
    - the Federal Information Processing Standard (FIPS)
    - the Advanced Encryption Standard (AES)
  - Common Criteria
- laws
  - Federal Information Security Modernization Act (FISMA)
  - The Cyber Resilience Act (CRA)

Vulnerability frameworks

Vulnerability taxonomy

- A Vulnerability is a rather high-level concept
- Very different phenomena can be deemed a vulnerability:
  - Having a deprecated feature still present in a UI
  - Using HTTP
  - Use-after-frees
  - etc...
- The MITRE corporation has a categorization of vulnerabilities by category: the Common Weakness Enumeration (CWE)
- CWE uses different categories to organize vulnerabilities

Vulnerability databases

- Idea: central repository listing published vulnerabilities
- The main problem here is maintenance
- The reference was the U.S National Vulnerability Database (NVD)
  - Funding issues have fragilized NVD as a reliable source
- China has 2 databases: CNVD and CNNVD
  - Both require accounts and can be difficult to navigate
- The EU has started the EUVD in 2025
  - Uses UUIDs for vendors and products
  - Version ranges are not super trivial to parse

The CVE program

- CVE stands for “Common Vulnerabilities and Exposures”
- Has been operated by MITRE with US government funding
- This funding has had ups and downs in 2025, which has led to data quality issues
- CVE Numbering Authorities (CNAs) can reserve and assign CVE numbers within their scope
- The Linux kernel team became a CNA in early 2024
- This is the program that serves as a base for the NVD
- Usually contains mostly rudimentary information at first
- Database is cached into a github repo

Common Vulnerability Scoring System (CVSS)

- Open Framework published by FIRST (specification)
- Current version is 4 since November 2023
- Usually, CVE records will use 1 or 2 versions of CVSS, depending on publication date
- CVSS results in:
  - A "vector" representing the values of a discrete set of metrics
  - A score between 0 and 10, which is a numerical conversion of the vector
  - both are therefore fully equivalent
- The score is the result of a complex-ish computation (online calculator)

Exploitation Predictability Scoring System (EPSS)

- Relatively recent (2021) effort
- Backed by FIRST
- Gives an estimated probability of the vulnerability being exploited within the next 30 days
- Uses machine learning to correlate exploit activity to vulnerabilities
- Relies on closed-source data from industry partners re: exploitation
- One big question here is how big the blind spot due to the incompleteness of the exploitation data is

Common Platform Enumerations (CPEs)

- CPEs are unambiguous identifiers for a specific product
- Can be very specific, or use wildcards (such as "*") to identify a range of products
- CPE is a MITRE trademark
- An authoritative dictionary of CPEs is maintained by the NIST
- The specification is also published by the NIST
- Structure of a CPE:

```
cpe:<version>:<part>:<vendor>:<product>:<version>:<update>:<edition>:<language> \\
     :<sw_edition>:<target_sw>:<target_hw>:<other>
```

- Example for CVE-2026-22174:

```
cpe:2.3:a:openclaw:openclaw:*:*:*:*:*:node.js:*:
```

Summary

- *CWE*s (Common Weakness Enumerations) classify vulneraibilities into types.
- *CVE*s (Common Vulnerabilities and Exposures) inventory individual vulnerabilities
  CVE-2012-5109 and CVE-2025-29834 are both examples of CWE-125:
  Out-of-bounds Read
- *CVSS* scores give a "grade" from 0 to 10 (critical) to the vulnerability
  - broken down into a vector along metrics (privilege, physical/local/network, user interaction…)
- *EPSS* is a score trying to predict the likelyhood of the vulnerability being exploited
- *CPE*s (Common Plaftorm Enumerations) are identifiers for the target (software or hardware)
  - They can be specific e.g. to a version, or a patch level

Regulation

The Cyber Resilience Act CRA

- European Law adopted on October 10 2024
  - Most dispositions start applying on December 11 2027
  - Manufacturer's reporting obligations to ENISA start on September 11 2026
- Applies to products placed on the European market
  - "Products" in this case include software
- Enforces obligations from different actors regarding cybersecurity
- Introduces a minimum amount of time these obligations must be carried out: the support period

CRA: support period

- It is defined in Article 13, paragraph 8
- Determined by the manufacturer, taking into account:
  - EU law (other than CRA) if existing
  - ADCO (ADministrative COoperation group) guidance
  - comparable products' support period
  - the "availability of the operating environment"
  - the support period of critical components
- Minimum is 5 years unless the product cannot reasonably be expected to last that long

CRA: useful resources

- The law
- The european comission has an implementation FAQ
- Draft candidates for future ETSI standards

Software Bill of Materials

SBoM

- CRA definition:

  ```
  a formal record containing details and supply chain relationships of
  components included in the software elements of a product with
  digital elements; a commonly used and machine-readable format covering
  at the very least the top-level dependencies of the products
  ```

- High-level explanation
- SBoMs are supposed to give end-users a good overview of their dependencies
- Without them, answering “are we using component X?” can be hard

SBom standards

- No law imposes a specific format for SBoMs
- Two main standards are competing:
  - CycloneDX
  - SPDX

CycloneDX

- Standard published by ECMA (ECMA-424 as of December 2025)
- Supports the following serialization formats:
  - JSON (JavaScript Object Notation)
  - XML (eXtensible Markup Language)
  - Protocol buffers
- Tends to be supported by commercial tools
  - SBoM Studio (Cybeats)
  - SBoM Manager (Keysight)

SPDX

- Software Package Data eXchange
- Open Source project hosted by the Linux Foundation
- Standard published by ISO
- Two versions coexist:
  - 2.3 is widely supported, and the current ISO standard
  - 3.0 tooling is trailing a bit, and the next ISO standard (draft)
- Tends to be supported by open-source tools

SPDX3: Example


```
{
  "@context": "https://spdx.org/rdf/3.0.1/spdx-context.jsonld",
  "@graph": [
    {
      "type": "CreationInfo",
      "@id": "_:CreationInfo1",
      "created": "2011-04-05T23:00:00Z",
      "createdBy": ["http://spdx.org/spdxdocs/bitbake-addba517-4804-5ae3-87c2-0c3a1a5812ba/bitbake/agent/OpenEmbedded"],
      "createdUsing": ["http://spdx.org/spdxdocs/bitbake-addba517-4804-5ae3-87c2-0c3a1a5812ba/bitbake/tool/oe-spdx-creator_1_0"],
      "specVersion": "3.0.1"
    },
    ...
    {
      "type": "Relationship",
      "spdxId": "http://spdx.org/spdxdocs/kiss-image-289390b0-d487-53ac-ab83-c6af87b87d1f/9f6c<...>1961/relationship/4a02<...>82ee",
      "creationInfo": "_:CreationInfo1",
      "extension": [
        {
          "type": "https://rdf.openembedded.org/spdx/3.0/id-alias",
          "https://rdf.openembedded.org/spdx/3.0/alias":
            "http://spdxdocs.org/openembedded-alias/by-doc-hash/cdfb<...>c0ee/kiss-image/UNIHASH/relationship/4a02dce12485470d12bf7e20117282ee"
        }
      ],
      "from": "http://spdx.org/spdxdocs/kiss-image-289390b0-d487-53ac-ab83-c6af87b87d1f/9f6c<...>1961/rootfs/kiss-image",
      "relationshipType": "contains",
      "to": [
        "http://spdx.org/spdxdocs/kiss-image-289390b0-d487-53ac-ab83-c6af87b87d1f/9f6c<...>1961/rootfs-file/etc_default_dropbear",
        "http://spdx.org/spdxdocs/kiss-image-289390b0-d487-53ac-ab83-c6af87b87d1f/9f6c<...>1961/rootfs-file/etc_group"
        ...
      ]
}
```

VEX

- Vulnerability Exploitability eXchange
- Not actually a standard: CISA only defines a set of minimum requirements
- There are a few implementation options:
  - CycloneDX integrates VEX within the *vulnerabilities* property
  - The Common Security Advisory Framework (CSAF) defines a VEX profile
  - OpenVEX is a community-driven standard of VEX that meets the CSAF minimum requirements
- The SBoM is an inventory of the software which can be used to lookup vulnerabilities
- VEX expresses e.g. the applicability of those vulnerabilities

SPDX3: Example


```
{
  "name": "busybox",
  "layer": "meta",
  "version": "1.36.1",
  "products": [
    {
      "product": "busybox",
      "cvesInRecord": "No"
    }
  ],
  "cpes": [ "cpe:2.3:*:*:busybox:1.36.1:*:*:*:*:*:*:*" ],
  "issue": [
    {
      "id": "CVE-2021-42380",
      "status": "Patched",
      "link": "https://nvd.nist.gov/vuln/detail/CVE-2021-42380",
      "detail": "backported-patch"
    },
    {
      "id": "CVE-2022-28391",
      "status": "Patched",
      "link": "https://nvd.nist.gov/vuln/detail/CVE-2022-28391",
      "detail": "backported-patch"
    },
  ]
}
```

Generating your SBoM

- Buildroot:
  - generates a JSON blurb listing packages with `make show-info` and `make pkg-stats`
  - can convert that output to CycloneDX using utils/generate-cyclonedx
- Yocto
  - has a create-spdx class, which supports both SPDX 2.3 and SPDX 3.0
  - process is documented here
  - also has a vex class, which was used to generate the example VEX
  - Yocto can annotate a CVE's status, as e.g. in their exclusion lists

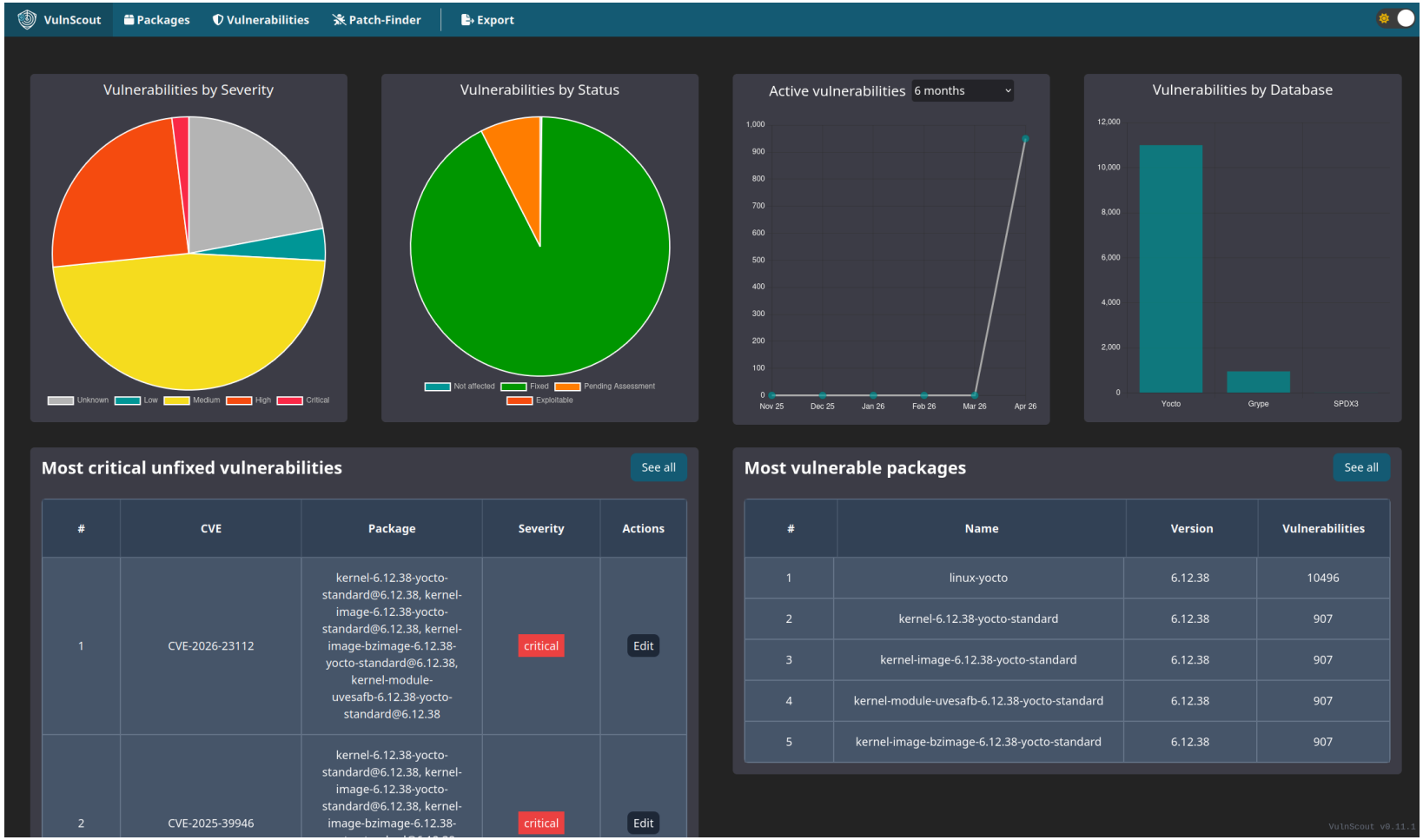
Using the SBoM

- SBoMs are a catalogue of the software on your device, including its version
- They can be used to look up vulnerabilities in databases (NVD, CVElistV5, EUVD...)
- They can also include *annotations* in the form of VEX information
  - For instance, yocto includes annotations that are part of the layer in their SBoMs
- They can be used periodically to scan *new* vulnerabilities, without rebuilding

VulnScout

- Open Source tool from Savoir-faire Linux
- Graphical interface
- Uses a docker container for the HTTP server
- Supports SPDX2.2, SPDX 3 and Cyclone DX

VulnScout


VulnScout
Packages
Vulnerabilities
Patch-Finder
Export
Vulnerabilities by Severity
Unknown
Low
Medium
High
Critical
Vulnerabilities by Status
Not affected
Fixed
Pending Assessment
Exploitable
Active vulnerabilities
6 months
1,000
900
800
700
600
500
400
300
200
100
0
Nov 25
Dec 25
Jan 26
Feb 26
Mar 26
Apr 26
Vulnerabilities by Database
12,000
10,000
8,000
6,000
4,000
2,000
0
Yocto
Grype
SPDX3
Most critical unfixed vulnerabilities
See all
CVE Package Severity Actions
1 CVE-2026-23112 kernel-6.12.38-yocto-standard@6.12.38, kernel-image-6.12.38-yocto-standard@6.12.38, kernel-image-bzimage-6.12.38-yocto-standard@6.12.38, kernel-module-uvesafb-6.12.38-yocto-standard@6.12.38 critical Edit
2 CVE-2025-39946 kernel-6.12.38-yocto-standard@6.12.38, kernel-image-6.12.38-yocto-standard@6.12.38, kernel-image-bzimage-6.12.38- critical Edit
Most vulnerable packages
See all
Name Version Vulnerabilities
1 linux-yocto 6.12.38 10496
2 kernel-6.12.38-yocto-standard 6.12.38 907
3 kernel-image-6.12.38-yocto-standard 6.12.38 907
4 kernel-module-uvesafb-6.12.38-yocto-standard 6.12.38 907
5 kernel-image-bzimage-6.12.38-yocto-standard 6.12.38 907
VulnScout v0.11.1

sbom-cve-check

- Open Source tool from Bootlin
- code - documentation
- Written in python, with as few dependencies as possible
- SBoM enrichment based on CVE databases: NVD, CVEListV5
- Supports SPDX 2.2 (Yocto's format) and SPDX 3
- Can take CVE annotations in format:
  - simple-annotations (YAML)
  - Yocto VEX manifest
  - OpenVEX

Upgrading strategy

Upgrades

- Updates can be necessary for multiple reasons:
  - Patching a security vulnerability
  - Rotating cryptographic material (e.g. later stage secure boot keys)
  - Legal obligation (e.g. CRA)
  - Adding functionality
- Some upgrades are simple, e.g. deploying a patch
- Version upgrades can be hard, especially when skipping over versions

Picking a version

- Embedded systems rely on a plethora of Open Source projects
- A lot of work is done **upstream**
- When a vulnerability is found or reported, or an important bug is fixed, only **supported versions** will get the fix
- The further one's version strays from a supported version, the harder **tracking** and **porting** fixes becomes
- In terms of security, being on the newest stable version is usually best
- The issue is compatibility

Long-Term Support

- Long-Term Support (LTS) versions stay supported for longer
  - Linux kernel LTS versions are supported for 2 years minimum
  - Yocto LTS versions receive 4 years of support
  - Buildroot LTSs are supported for 3 **years** instead of 3 **months**
- This is advertised plainly:
  - the Linux kernel has a list of LTS versions
  - so does the Yocto project
  - the same for buildroot
- The Civil Infrastructure Platform aims to extend LTS windows to over 5 years
  - consortium of large industry members
  - under the Linux Foundation
  - some features are introduced, not just fixes

Practical lab - Exploring compliance topics

Time to get familiar with SBoMs and CVEs!

- Generating an SBoM for a Yocto distribution
- Analyzing the generated SBoM using sbom-cve-check
- Patching a CVE present in the SBoM
- Differential analysis

A/B updates

A/B updates

- Mechanism for Over-The-Air (OTA) *image* updates
- Key idea: avoid one update bricking the device
  - Even if tested, one update might prevent the system from booting
  - Enable the device to recover from a bad update automatically
  - Avoid RMAs or in-the-field intervention
- The key idea is to maintain 2 copies of the system, in two **slots**
  - minimum is 2 rootFS
  - usually also includes the kernel
  - can involve the bootloader, depending on the bootROM
- Non-A/B OTAs are deprecated in Android since Android 6 (pre August 2016)

A/B updates: slots

- A/B systems have 2 slots: slot A and slot B
- They each have dedicated boot locations (usually boot medium partitions)
- On the running system, the slot booted is the **current** slot
- An OTA should never touch boot locations from the **current** slot
- Each slot can be marked **bootable**
- Updates will usually be built independently from the slot they will occupy

A/B updates: strategy

- At boot, the system detect which slot (A or B) is the **current** slot
- The updater will locate the boot locations for the **alternate** slot
- The updater then applies the update to the **alternate** boot locations
- The updater marks the **alternate** slot as **next** to be booted
- Eventually, the system reboots. Possibly triggered at the end of the update
- At boot, the bootloader detects the **next** slot and boots it
  - If the system fails to boot, the watchdog reboots the system. If the slot fails to boot several times, the system boots the other slot.
  - If the boot succeed, the system runs checks and if they pass, marks the **current** slot as **primary**

A/B updates: bootloader

- The bootloader is responsible for:
  - Booting the **primary** slot by default
  - Keeping track of boot failures per slot
  - Trying out the **alternate** slot if it is marked as **next**
  - Optionally, telling the kernel which slot has been booted (A or B)
- Communication between bootloader and the system is short

Updater: SWUpdate

- Open source project
- website – code – doc
- Used e.g. by the Civil Infrastructure Project
- Integrated in both Buildroot and Yocto
- Written in C
- Uses `libconfig` syntax for configuration

SWUpdate: configuration

```
software =
{
        version = "0.1.0";
        description = "Firmware update for XXXXX Project";

        hardware-compatibility: [ "1.0", "1.2", "1.3"];

        partitions: (
                { name = "rootfs"; device = "mtd4"; size = 104896512; },
                { name = "data"; device = "mtd5"; size = 50448384; }
        );

        images: (
                { filename = "rootfs.ubifs"; volume = "rootfs"; },
                { filename = "swupdate.ext3.gz.u-boot"; volume = "fs_recovery"; },
                { filename = "sdcard.ext3.gz"; device = "/dev/mmcblk0p1"; compressed = "zlib";},
                { filename = "bootlogo.bmp"; volume = "splash"; },
                { filename = "uImage.bin"; volume = "kernel"; },
                { filename = "fpga.txt"; type = "fpga"; },
                { filename = "bootloader-env"; type = "bootloader"; }
        );

        files: ({ filename = "README"; path = "/README"; device = "/dev/mmcblk0p1"; filesystem = "vfat"; });

        scripts: ( { filename = "erase_at_end"; type = "lua"; }, { filename = "display_info"; type = "lua"; });

        bootenv: (
                { name = "vram"; value = "4M"; },
                { name = "addfb"; value = "setenv bootargs ${bootargs} omapfb.vram=1:2M,2:2M,3:2M omapdss.def_disp=lcd"; }
        );
}
```

Updater: RAUC

- Robust Auto-Update Controller
- Also open source
- website – code – doc
- Integrated in Buildroot and Yocto
- Written in python
- Uses a configuration and `manifest` in INI format
- Updates are handled in the form of `bundles`

RAUC: configuration

```
[system]
compatible=rauc-demo-x86
bootloader=grub
mountprefix=/mnt/rauc
bundle-formats=-plain

[keyring]
path=demo.cert.pem

[slot.rootfs.0]
device=/dev/sda2
type=ext4
bootname=A

[slot.rootfs.1]
device=/dev/sda3
type=ext4
bootname=B
```

RAUC: Bundle manifest

- Must be called `manifest.raucm`

```
[update]
compatible=Test Platform
version=2023.11.0

[bundle]
format=verity

[image.rootfs]
filename=system-image.ext4

[image.bootloader]
filename=barebox.img
```

Updates backend: hawkBit

- Open Source project from the Eclipse foundation
- Back-end for update rollout
- Can break down shipping updates into groups
- Includes a management interface and visualisation
- Supported by both RAUC and SWUpdate

Examples

A/B updates using RAUC + U-Boot

- U-Boot is natively supported by RAUC
- RAUC will use U-Boot's environment to affect boot order
  - `BOOT_ORDER`: Names of all slots, in order of priority
  - `BOOT_<slot_name>_LEFT`: remaining boot attempts for the slot
  - When using secure boot, u-boot environment should be protected using `CONFIG_ENV_WRITEABLE_LIST`
- The environments variables will be read and set by a boot script
  - RAUC includes an example (see next slide)
- Alternatively, one can configure U-Boot to use the RAUC bootmeth

A/B updates using RAUC + U-Boot

```
test -n "${BOOT_ORDER}" || setenv BOOT_ORDER "A B"
test -n "${BOOT_A_LEFT}" || setenv BOOT_A_LEFT 3
test -n "${BOOT_B_LEFT}" || setenv BOOT_B_LEFT 3

setenv bootargs for BOOT_SLOT in "${BOOT_ORDER}"; do
  if test "x${bootargs}" != "x"; then
    # skip remaining slots
  elif test "x${BOOT_SLOT}" = "xA"; then
    if test 0x${BOOT_A_LEFT} -gt 0; then
      echo "Found valid slot A, ${BOOT_A_LEFT} attempts remaining"
      setexpr BOOT_A_LEFT ${BOOT_A_LEFT} - 1
      setenv load_kernel "nand read ${kernel_loadaddr} ${kernel_a_nandoffset} ${kernel_size}"
      setenv bootargs "${default_bootargs} root=/dev/mmcblk0p1 rauc.slot=A"
    fi
  elif test "x${BOOT_SLOT}" = "xB"; then
    if test 0x${BOOT_B_LEFT} -gt 0; then
      setexpr BOOT_B_LEFT ${BOOT_B_LEFT} - 1
      setenv load_kernel "nand read ${kernel_loadaddr} ${kernel_b_nandoffset} ${kernel_size}"
      setenv bootargs "${default_bootargs} root=/dev/mmcblk0p2 rauc.slot=B"
    fi
  fi done

if test -n "${bootargs}"; then
  saveenv else
  echo "No valid slot found, resetting tries to 3"
  setenv BOOT_A_LEFT 3
  setenv BOOT_B_LEFT 3
  saveenv
  reset fi

run load_kernel bootm ${loadaddr_kernel}
```

A/B updates on RPi 5

- Raspberry Pis have an interesting boot sequence
- the boot actually stats on the VideoCore, which is the GPU
- this lets users configure the CPU's bootloader via `.txt` files
  - `config.txt`
  - `autoboot.txt`
- the RPi bootloader already supports A/B updates, via its `tryboot` feature
- It also supports dynamically changing configuration based on the `boot_partition`
  - This helps making the OTA update slot-agnostic

Practical lab - Exploring A/B updates

Time to setup RAUC!

- RAUC configuration on the target
- Building, shipping and installing a RAUC bundle
- Bundle signature verification using PKCS#11

Last slides

Thank you!

And may the Source be with you

Rights to copy

Document sources: https://github.com/bootlin/training-materials/