Linaro Connect
MADRID 2024 | MAY 14-17 2024

Simple, Yocto, Secure Boot

Using new systemd features to pick all three!

Maybe... Eventually...
At some point...

About me...

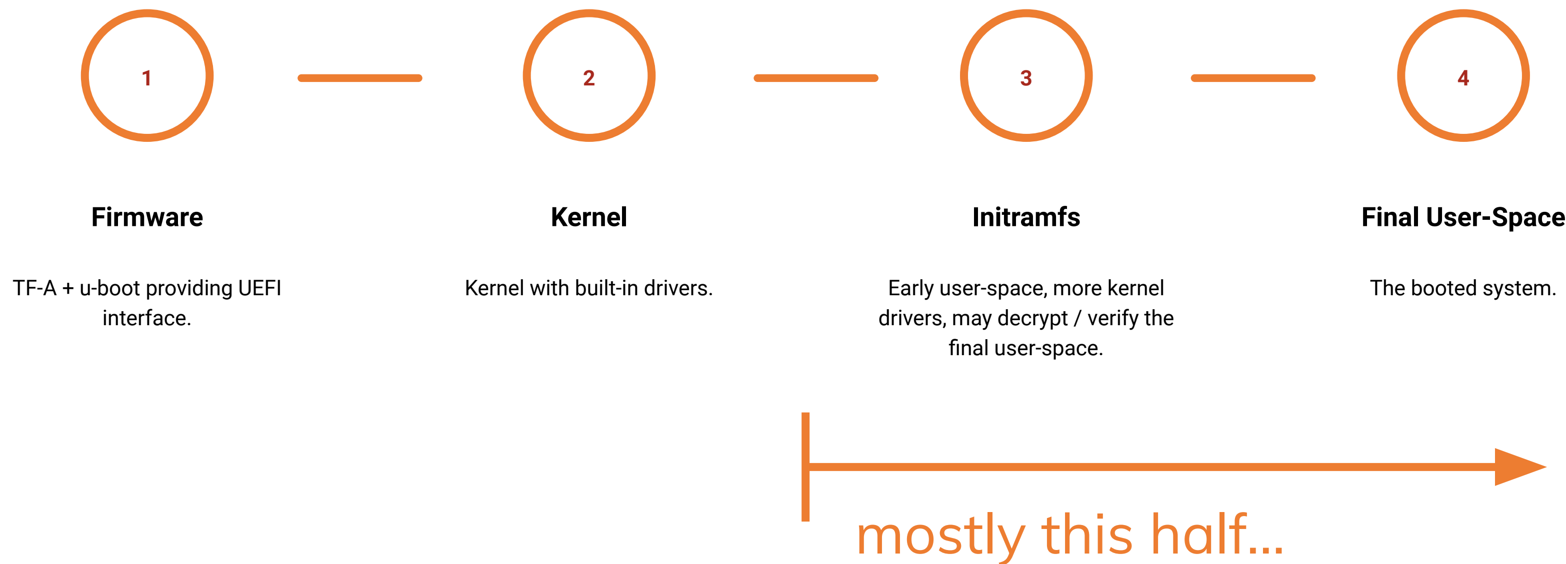
Name: Erik Schilling

with Linaro since beginning of 2023.

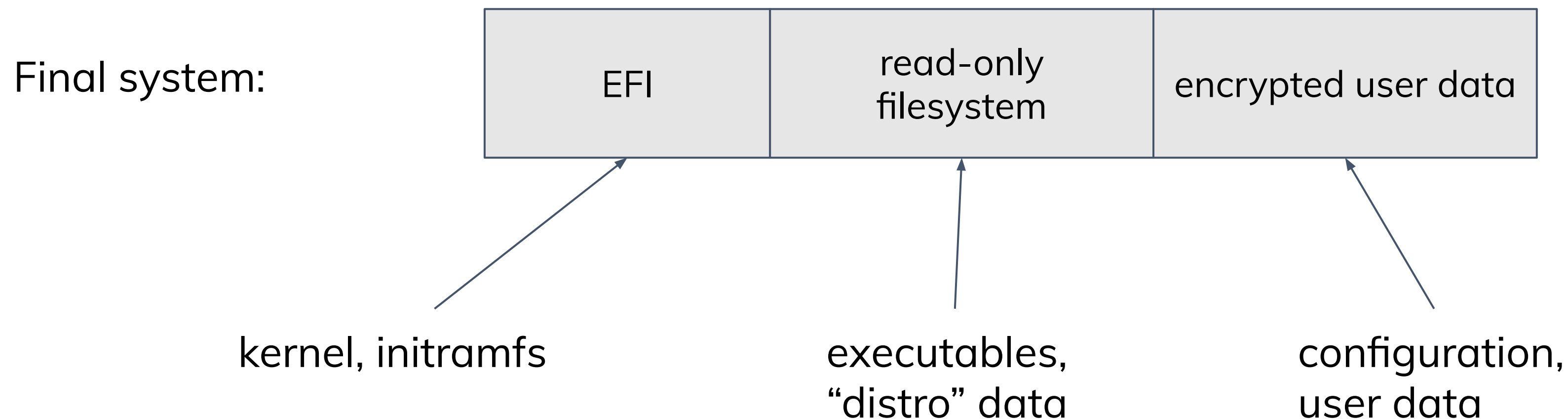
before that I worked on Embedded Linux projects in agriculture.

Mail: erik.schilling@linaro.org

What I mean with secure boot today...



The (assumed) Goal



Your exact use-case may differ!

(but you may find the same building blocks useful!)

Problem 1: How to create a secure user partition?

A read-only filesystem is great for reliability. However it is mostly useful for the binaries and data that is common across a fleet of devices.

But eventually, we will want to store data or configuration somewhere!

We will want to create a filesystem that allows writing, but still encrypts and ensures authenticity of the data on disk.

Ideally, the keys for this should be generated on the device and should never leave it!

Privately encrypted data in practise (today)

1. Either pre-allocate or dynamically create the partition.
2. If pre-populated data is required, we need to move that away temporary (dangerous!)
3. Create the LUKS partition.
4. Enroll key in TPM.
5. Copy pre-populated data back (if needed).

That's a lot of steps (that can go wrong)!

Example from TRS (enrollment only)

SOURCE

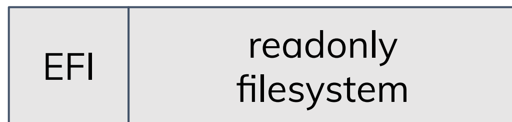
```
tpm2_createprimary -Q --hierarchy=o --key-context=prim.ctx
tpm2_loadexternal --key-algorithm=rsa --hierarchy=o \
      --public=signing_key_public.pem --key-context=signing_key.ctx \
      --name=signing_key.name > /dev/null
tpm2_startauthsession --session=session.ctx
tpm2_policyauthorize --session=session.ctx --policy=authorized.policy \
      --name=signing_key.name > /dev/null
tpm2_flushcontext session.ctx
cat ${passfilename} | tpm2_create --hash-algorithm=sha256 \
       --public=auth_pcr_seal_key.pub --private=auth_pcr_seal_key.priv \
      --sealing-input=- --parent-context=prim.ctx --policy=authorized.policy > /dev/null
tpm2_load -Q --parent-context=prim.ctx \
      --public=auth_pcr_seal_key.pub --private=auth_pcr_seal_key.priv \
      --name=seal.name --key-context=seal.ctx > /dev/null
tpm2_evictcontrol -Q -C o -c ${TPM_NVINDEX_ROOTFS} 2> /dev/null || \
      echo "initramfs: TPM NVRAM ${TPM_NVINDEX_ROOTFS} index deleted."
tpm2_evictcontrol --hierarchy=o --object-context=seal.ctx ${TPM_NVINDEX_ROOTFS} > /dev/null
```

Solution: systemd-repart

```
# 10-esp.conf
[Partition]
Type=esp

# 50-usr.conf
[Partition]
Type=usr

# 60-root.conf
[Partition]
Type=root
Format=ext4
Encrypt=tpm2
FactoryReset=yes
```

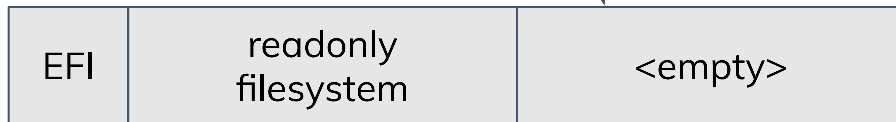
processed in alphabetical order

FACTORY IMAGE



minimal factory image flashed to device storage (hopefully) leaves empty space

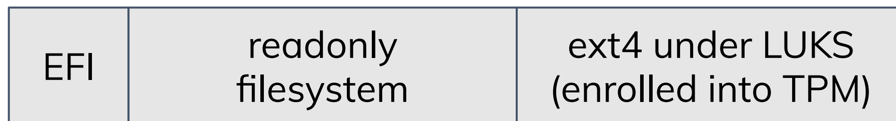
BEFORE BOOT



matched against 10-esp.conf, 50-usr.conf. Therefore left unchanged.

60-root.conf is not found, therefore it will get created.

AFTER BOOT



Aside: / or `/usr` as read-only

Yocto comes with existing tooling (read-only-rootfs) to build a readonly filesystem.

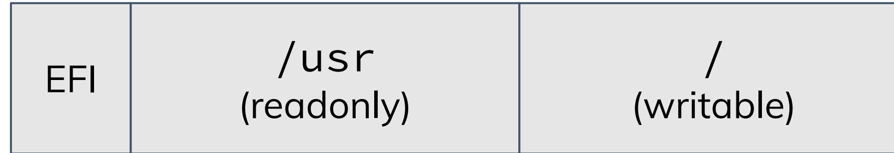
Here, a filesystem for / is created, then a bunch of symlinks that need to be mutable to a volatile tmpfs.

Yocto's read-only-rootfs solves some common scenarios, but some fundamental problems remains:

- Many `/etc` locations are often assumed to be writable by software. We will need a lot of exceptions.
- /var is (by default) volatile and discarded after reboot.
- Tweaking any of this quickly becomes non-trivial.

Aside: / or `/usr` as read-only

Solution: Only boot with /usr populated!



Initramfs will mount `/usr` into `/` before switching to `/`.

Systemd is perfectly fine booting with only /usr. It will automatically create all necessary files on / automatically (through systemd-tmpfiles.d).

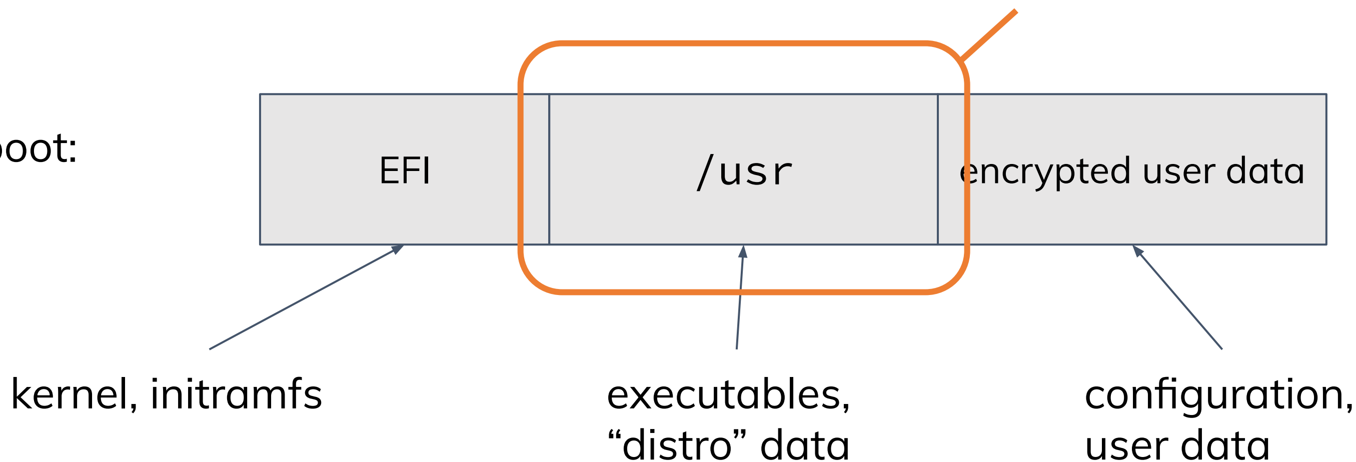
(Yocto currently does not 100% like this and a few tweaks are required)

This solution does not require careful symlinks and is easy to factory reset (just wipe the partition).

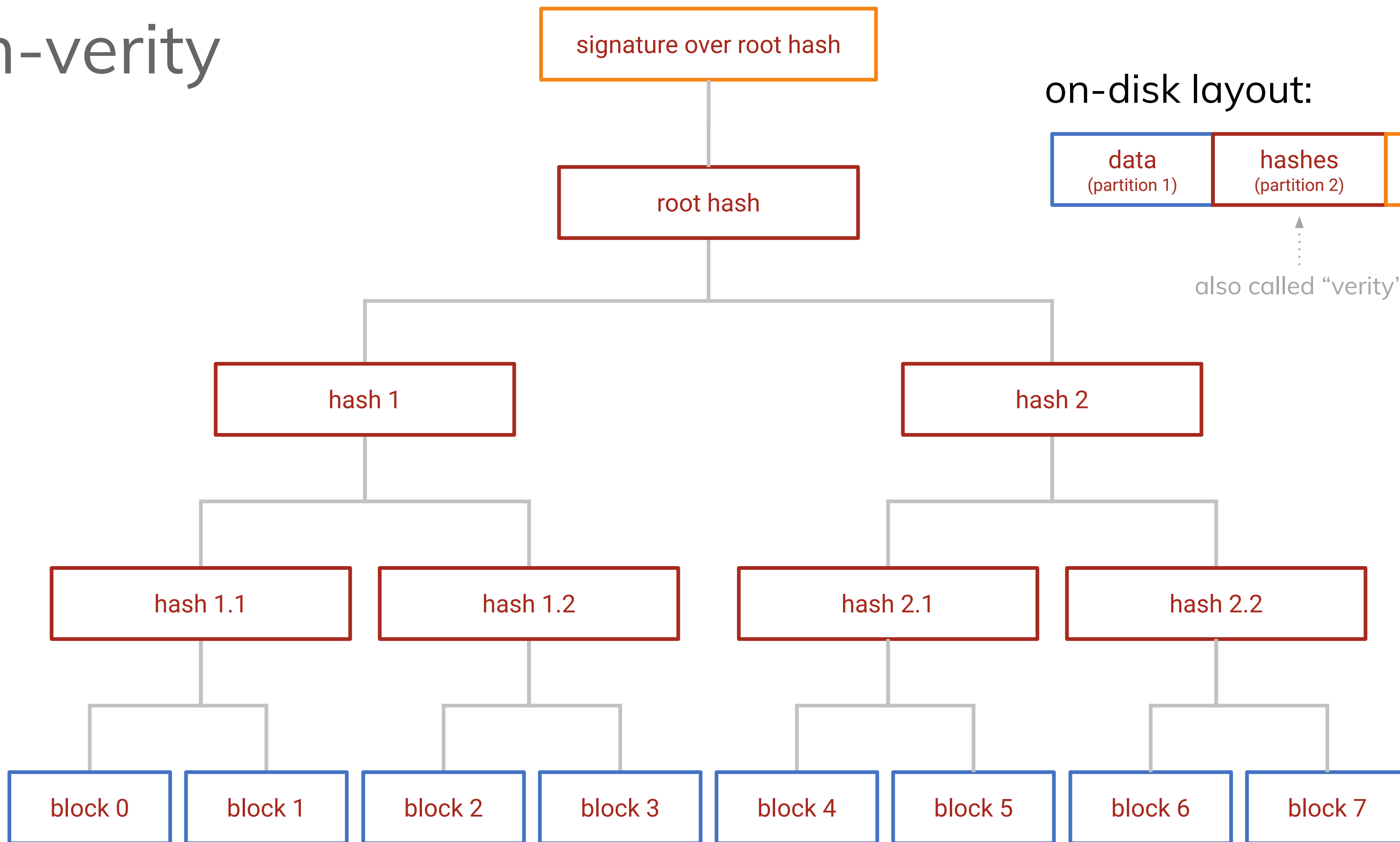
Problem 2: How to secure /usr?

Factory image:

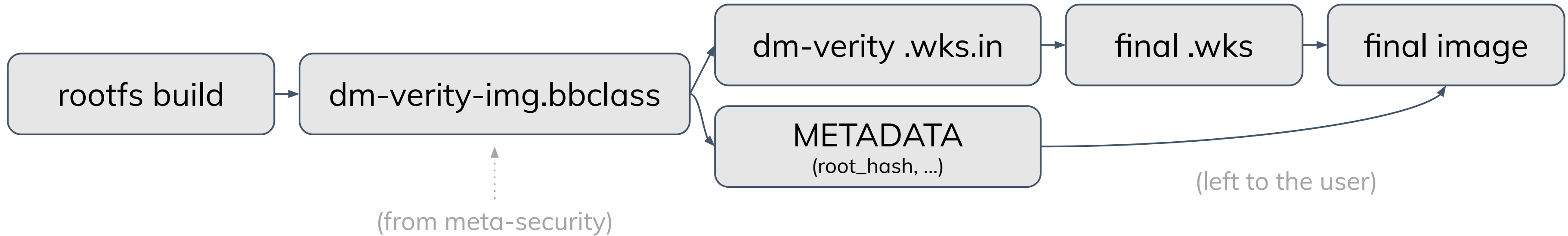
After first boot:



dm-verity



dm-verity in practise (today)



The intermediate .wks.in only offers limited flexibility (always tagged as / in .wks).

root_hash is hard to handle: It has to be read from a intermediate file. This makes dependency ordering annoying for signing or embedding into kernel command lines.

⇒ The solution requires quite some setup and customization for deployment (splitting into separate images that are built in stages may help).

dm-verity (maybe tomorrow): systemd-repart

It could be simpler!

systemd-repart can also *create* disk images from scratch!

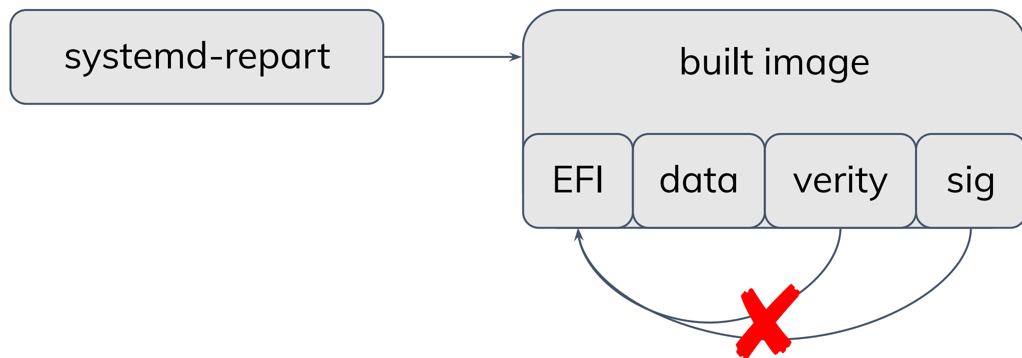
```
> systemd-repart --root="<path-to-rootfs>" \
    --definitions="<path-to-repart-conf-dir>" \
    --empty=create \
    --size=auto \
    --dry-run=no \
    --private-key=db.key --certificate=db.crt \
    --offline=yes \
    output.img
```

```
# 02-usr.conf
[Partition]
Type=usr
CopyFiles=/usr/:/
Verity=data
VerityMatchKey=usr
Minimize=guess

# 02-usr-verity.conf
[Partition]
Type=usr-verity
Verity=hash
VerityMatchKey=usr
Minimize=guess

# 04-usr-verity-sig.conf
[Partition]
Type=usr-verity-sig
Verity=signature
VerityMatchKey=usr
```

Problem 3: How do we get this mounted?

We have a data, verity and signature partition. How do we get it mounted?

We *could* bake the roothash + signature into the kernel command line. However, this info would likely live somewhere on the factory image that we just built:



but we do not need this!

Image is built in one step. Customization of the kernel command line would likely need to be a post-processing step on the EFI partition.

Introducing: uapi-group

“The userspace API (“uapi”) group is a community for people with an interest in innovating how we build, deploy, run, and securely update modern Linux operating systems. It serves as a central gathering place for specs, documentation, and ideas.”

https://uapi-group.org/

The group has a decent overlap with the systemd development community, but is set up as standalone interest group.

We are going to look at:

- Discoverable Partition Specification
- Unified Kernel Images
- Bootloader Interface

Discoverable Partitions Specification

systemd-repart will assign these automatically!

Partition Type	UUID
EFI System Partition	SD_GPT_ESP c12a7328-f81f-11d2-ba4b-00a0c93ec93b
Root Partition (AArch64)	SD_GPT_ROOT_ARM64 b921b045-1df0-41c3-af44-4c6f280d3fae
/usr/ Partition (AArch64)	SD_GPT_USR_ARM64 b0e01050-ee5f-4390-949a-9101b17104e9
Root Verity Partition (AArch64)	SD_GPT_ROOT_ARM64_VERITY df3300ce-d69f-4c92-978c-9bfb0f38d820
/usr/ Verity Partition (AArch64)	SD_GPT_USR_ARM64_VERITY 6e11a4e7-fbca-4ded-b9e9-e1a512bb664e
Root Verity Signature Partition (AArch64)	SD_GPT_ROOT_ARM64_VERITY_SIG 6db69de6-29f4-4758-a7a5-962190f00ce3
/usr/ Verity Signature Partition (AArch64)	SD_GPT_USR_ARM64_VERITY_SIG c23ce4ff-44bd-4b00-b2d4-b41b3419e02a

Warning!

Blindly using this auto-discovery won't give you a secure system!

You will want to set an image-policy to avoid mounting unauthenticated partitions!

Discoverable Partitions Specification

Systemd will scan the disk that was selected as boot medium and auto-mounts discovered partitions.

In our case of a dm-verity backend /usr partition it will:

1. Detect the data, verity and verity signature partitions.
2. Read the root_hash from the verity signature metadata.
3. Verify the signature on the root_hash.
4. Mount the partition.

Problem 4: Which disk is booted?

The auto-discovery mechanism needs to know which disk was booted.

We won't detail this here… but…

systemd simply defines a set of expected EFI vars for this:
https://systemd.io/BOOT_LOADER_INTERFACE/

Problem 5: This logic has to happen somewhere!

Since we auto-discover the partition where all our software lives one, we need an initramfs to get us started!

Building an initramfs with systemd is fairly straight forward. But now we have another component that we need to verify.

In fact, we have not covered any of the early boot components:

- Kernel
- Kernel command line
- Splash screen
- Initramfs ⇐ *this is our new requirement!*
- Device-Tree

The Problem with Problem 5:

Only binaries can easily be verified against signatures before execution.

UEFI also specifies `EFI_PKCS7_VERIFY_PROTOCOL`, but the protocol is not mandated by Arm SystemReady standards.

⇒ There is no readily available way to verify non-executable binaries.

Signing and validating all of the components separately may also easily become cumbersome.

Unified Kernel Image

https://uapi-group.org/specifications/specs/unified_kernel_image/

Let's just stuff it all into one big EFI binary!

The solution:

Combine the kernel + initramfs + splash + device-tree into a combined binary.

A stub then extracts the info and invokes the actual kernel.

The bootloader only needs to verify combined binary.

nth	paddr	size	vsize	perm	type	name
0	0x00000400	0x19400	0x1a000	-r-x	----	.text
1	0x00019800	0x6000	0x6000	-r--	----	.rodata
2	0x0001f800	0x200	0x1000	-rw-	----	.data
3	0x0001fa00	0x200	0x1000	-r--	----	.sbat
4	0x0001fc00	0x200	0x1000	-r--	----	.sdmagic
5	0x0001fe00	0x200	0x1000	-r--	----	.reloc
6	0x00020000	0x400	0x1000	-r--	----	.osrel
7	0x00020400	0x200	0x1000	-r--	----	.cmdline
8	0x00020600	0x200	0x1000	-r--	----	.uname
9	0x00020800	0x200	0x1000	-r--	----	.pcrpkey
10	**0x00020a00**	**0x9f59600**	**0x9f5a000**	**-r--**	**----**	**.initrd**
11	0x09f7a000	0x1200	0x2000	-r--	----	.pcrsig
12	**0x09f7b200**	**0xdfe400**	**0xdff000**	**-r--**	**----**	**.linux**

Unified Kernel Image

Building a UKI can be done with objcopy. But systemd also provides a neat little tool:

```
ukify.py build \
      --linux linux.efi \
      --initrd core-image-base-v8a-arm64.cpio.xz \
      --secureboot-certificate db.crt \
      --secureboot-private-key db.key \
      --cmdline 'console=hvc0 rootwait' \
      --os-release=@os-release \
      --efi-arch aa64 \
      --stub path/to/linuxaa64.efi.stub
```

Where is the catch?

Most of the things shown are usable today!

But: The integration work in Yocto is still lacking. Luckily WIP patches are starting to hit the maillinglists:

- [PATCH] uki: Add support for building Unified Kernel Images
  - by: `Michelle Lin <michelle.linto91@gmail.com>`
- [PATCH v5] systemd-boot: Add recipe to compile native
  - by: `Viswanath Kraleti <quic_vkraleti@quicinc.com>`
- [PATCH RFC] systemd-repart.bbclass: provide build-time partitioning helper
  - by: `Erik Schilling <erik.schilling@linaro.org>`

Further interesting patches bringing integration of new systemd features:

- [PATCH v5 0/3] pkg-database and systemd-sysext image
  - by: `Johannes Schneider <johannes.schneider@leica-geosystems.com>`

The catch... continued...

Support for auto-discovery of /usr partitions under verity is not currently supported yet.

Early boot TPM discovery has some problems.

One still needs to pass the usrhash through the kernel command line.

Which means that UKI's end up depending on the filesystem build again 😥.

No tooling for resigning images (UKI + verity-signature partition) against production keys.

Thanks to

Mikko Rapeli (Yocto integration)
Ilias Apalodimas (UEFI and Bootloader)
Daniel Thompson (Slide review)
Luca Boccassi (Systemd discussions)
Fabian Vogt (Systemd discussions)
Lennart Poettering (Systemd discussions)
Daan De Meyer (Systemd discussions)

Thank you