Secure and flexible boot with U-Boot bootloader

Marek Vašut <marex@denx.de>

October 15, 2014
Software engineer at DENX S.E. since 2011
- Embedded and Real-Time Systems Services, Linux kernel and driver development, U-Boot development, consulting, training.
- Custodian at U-Boot bootloader
- Versatile Linux kernel hacker
Tips to build a system, which...

- ...is resistant against storage data corruption
- ...is resistant against offline tampering
- ...is resistant against data extraction
The boot process

That’s easy ... not:

- Power on or Reset
- CPU starts executing from predefined address
- Bootloader is started
- Kernel is started
- Root filesystem is used

Lots of things happen inbetween, that’s where the problems are.
Power on or Reset

Hardware magic happens before CPU starts executing code:

- All relevant components are put into reset
- Reset brings components into defined state
- CPU start executing code after released from reset

...but...

- There are multiple types of reset
- Well defined post-reset state allows for proper analysis
- Not well defined post-reset state is source of problems

Make sure your hardware is reliable in the first place!
Recurring problem!
Reset is not connected properly to all components
Often seen with MTD devices (SPI NOR) or SD/MMC cards
Example: CPU boots from SPI NOR
  - Software does a PP operation and feeds SPI NOR with data
    → Reset happens
    ⇒ Board does not boot – WHY?
    ⇒ Data corruption might happen – WHY?
Naive solution: Send RESET opcode in software (FAILS!)
Solution: CPU has reset output
  - Connect it to the boot media reset input
Tip: Other boot media

- SD/eSD/MMC/eMMC:
  - Verify EOL behavior
    - Must indicate bad blocks, not emit bad data
  - Baked firmware problems

- NAND:
  - First EB often guaranteed to be OK by vendor
    - This might not extend to reprogramming of the first EB.
    - Read the datasheet carefully!
  - First page is 1/2/4 KiB big ⇒ U-Boot SPL
  - MLC NAND has even worse problems than SLC NAND
First code running on the CPU

Might be executing from within the CPU (BootROM)

Might be executing from external memory (NOR, FPGA, ...)

BootROM:

Facilitates loading from non-trivial media
(SPI NOR, SD/MMC, RAW NAND, USB, Network, ...)

Might provide facilities for verified and encrypted boot

Often closed source

Usually cannot be updated with fixes (ROM)
U-Boot SPL:

- First user-supplied code running
- Smaller size than U-Boot
- Function varies on per-device basis
- Does basic hardware initialization
- Loads payload from media, verifies it and executes it
  → Payload can be either U-Boot, Linux, ...

RAW NAND specifics:

- UBI doesn’t fit into first 4KiB of NAND
- U-Boot SPL does ECC, but doesn’t update NAND
- Multiple copies of U-Boot in NAND and update them
- Better: Store U-Boot in NOR, kernel and FS in NAND
U-Boot

- The size limits of SPL are almost non-existent
- Full support for filesystems (ext234, reiserfs, vfat...)
- UBI and UBIFS support for NAND
- Supports verification and encryption
- fitImage support
Make sure your HW starts from a defined state
Always verify the next payload
Boot from reliable boot media (not RAW NAND)
Never place anything important into RAW NAND
Common kernel image types

- **zImage**
  - Prone to silent data corruption, which can go unnoticed
  - Contains only kernel image
  - In widespread use

- **uImage (legacy)**
  - Weak CRC32 checksum
  - Contains only kernel image
  - In widespread use

- **fitImage**
  - Configurable checksum algorithm
  - Can be signed
  - Contains arbitrary payloads (kernel, DTB, firmware…)
  - There is more!
  - Not used much :-(
The fitImage in detail

- Successor to ulmage
- Descriptor of image contents based on DTS
- Can contain multiple files (kernels, DTBs, firmwares...)
- Can contain multiple configurations (combo logic)
- New image features can be added as needed
- Supports stronger csums (SHA1, SHA256...)
- Protection against silent corruption
- U-Boot can verify fitImage signature against public key
- Protection against tampering
- Linux build system can not generate fitImage :-(
- Yocto can not generate fitImage yet :-(

Marek Vašut <marex@denx.de>  Secure and flexible boot with U-Boot bootloader
ulimage vs. fitImage: Creation

/dts-v1/;
/
{
    description = "Linux kernel";
    #address-cells = <1>;
    images {
        kernel@1 {
            description = "Linux kernel";
            data = /incbin/"./arch/arm/boot/zImage";
            arch = "arm";
            os = "linux";
            type = "kernel";
            compression = "none";
            load = <0x8000>;
            entry = <0x8000>;
            hash@1 {
                algo = "sha1";
            };
        };
    };
    configurations {
        default = "conf@1";
        conf@1 {
            description = "Boot Linux kernel";
            kernel = "kernel@1";
            hash@1 {
                algo = "sha256";
            };
        };
    };
}$ mkimage -f fit-image.its fitImage
$ mkimage -A arm -O linux -T kernel -C none -a 0x8000 -e 0x8000 -n "Linux kernel"
    -d arch/arm/boot/zImage uImage

Marek Vašut <marex@denx.de>  Secure and flexible boot with U-BooT bootloader
uImage => load mmc 0:1 ${loadaddr} uImage
uImage => bootm ${loadaddr}

fitImage => load mmc 0:1 ${loadaddr} fitImage
fitImage => bootm ${loadaddr}

- uImage is easier to construct
- uImage does not need fit-image.its file
- uImage boot command is the same as fitImage one

uImage wins thus far...
ulimage vs. fitImage: Device Tree Blob

... / {
    images {
        ...
        + fdt01 {
            description = "Flattened Device Tree blob";
            data = /incbin/(/arch/arm/boot/dts/imx28-m28evk.dtb);
            type = "flat_dt";
            arch = "arm";
            compression = "none";
            hash01 {
                algo = "sha256";
            };
        };
        ...
    };
    configurations {
        conf01 {
            ...
            + fdt = "fdt01";
            ...
        };
    };
};

Marek Vašut <marex@denx.de> Secure and flexible boot with U-Boot bootloader
ulimage vs. fitImage: Boot with DT

uImage   => load mmc 0:1 ${loadaddr} uImage
uImage   => load mmc 0:1 ${fdtaddr} imx28-m28evk.dtb
uImage   => bootm ${loadaddr} - ${fdtaddr}

fitImage => load mmc 0:1 ${loadaddr} fitImage
fitImage => bootm ${loadaddr}

- fitImage allows an update of all boot components at the same time
- fitImage protects the DTB with a strong checksum (hash node)
- fitImage does not require change of the boot command here
fitImage: Multiple configurations

... /

images {
    kernel@1 {};
    fdt@1 {};
    fdt@2 {};
    ...
};
configurations {
    conf@1 {
        kernel = "kernel@1";
        fdt = "fdt@1";
        ...
    };
    conf@2 {
        kernel = "kernel@1";
        fdt = "fdt@2";
        ...
    };
};

=> bootm ${loadaddr}#conf@2
=> bootm ${loadaddr}:kernel@2

▶ fitImage can carry multiple predefined configurations
▶ fitImage allows for execution of config using the # (HASH)
▶ fitImage allows for direct execution of image using the : (COLON)
... 
/ {
    images {
        ...
        firmware@1 {
            description = "My FPGA firmware";
            data = /incbin/("./firmware.rbf");
            type = "firmware";
            arch = "arm";
            compression = "none";
            hash@1 {
                algo = "sha256";
            }
        }
    }
    ...
};

=> imxtract ${loadaddr} firmware@1 ${fwaddr}
=> fpga load 0 ${fwaddr}

▶ fitImage can contain multiple arbitrary firmware blobs
▶ fitImage protects them with strong checksums
=> iminfo ${loadaddr}

## Checking Image at 10000000 ...

FIT image found
FIT description: Linux kernel and FDT blob for mcvevk
Created: 2014-09-22 15:37:52 UTC

Image 0 (kernel@1)
Description: Linux kernel
Created: 2014-09-22 15:37:52 UTC
Type: Kernel Image
Compression: uncompressed
Data Start: 0x100000d8
Data Size: 3363584 Bytes = 3.2 MiB
Architecture: ARM
OS: Linux
Load Address: 0x00008000
Entry Point: 0x00008000
Hash algo: crc32
Hash value: 5c7efdb5

Image 1 (fdt@1)
Description: Flattened Device Tree blob
Created: 2014-09-22 15:37:52 UTC
Type: Flat Device Tree

Default Configuration: 'conf@1'

Configuration 0 (conf@1)
Description: Boot Linux kernel with FDT blob
Kernel: kernel@1
FDT: fdt@1

## Checking hash(es) for FIT Image at 10000000 ...

Hash(es) for Image 0 (kernel@1): crc32+
Hash(es) for Image 1 (fdt@1): crc32+
fitImage can protect all artifacts needed during boot
fitImage can batch all files into one
⇒ Essential boot files can be updated at once
fitImage supersedes uImage with flexibility and extensibility
fitImage is much less prone to silent corruption of its payloads
Tampering protection for boot artifacts
Attach signature to fitImage image or config node
  - SHA-1 + RSA-2048
  - SHA-256 + RSA-2048
  - SHA-256 + RSA-4096
U-Boot verifies the signature against a public key
Public key must be stored in read-only location
This is five step process:

- Enable control FDT support in U-Boot and make use of it
- Generate cryptographic material (using OpenSSL)
- Generate the control FDT with public key in it
- Assemble U-Boot that can verify the fitImage signature
- Update U-Boot and test the setup...
- **CONFIG_RSA** – support for RSA signatures
- **CONFIG_FIT_SIGNATURE** – support for signed fitImage
- **CONFIG_OF_CONTROL** – support for control DT in U-Boot
Our cryptomaterial goes into key_dir="/work/keys/"

The shared name of the key is key_name="my_key"

Generate a **private** signing key (RSA2048):

```bash
$ openssl genrsa -F4 -out \
"${key_dir}"/"${key_name}".key 2048
```

Generate a **public** key:

```bash
$ openssl req -batch -new -x509 \ 
-key "${key_dir}"/"${key_name}".key \ 
-out "${key_dir}"/"${key_name}".crt
```
Example of control FDT (u-boot.dts):

```
/dts-v1/;
/

model = "Keys";
compatible = "denx,m28evk";
signature {
    sig@0 {
        required = "conf"; /* or "image" */
algo = "sha256,rsa2048";
        key-name-hint = "my_key";
    };
    sig@1 {...};
    ...
};
```

- The `my_key` in `key-name-hint` node must be `$\{key_name\}$`
- There can be multiple keys in the control DT
- The `u-boot.dtb` must be read-only on the device
Example of signature node in fitImage ITS (fit-image.its):

```plaintext
... 
/ { 
    ... 
    configurations { 
        conf@1 { 
            ... 
            hash@1 {...}; 
            + signature@1 { 
                algo = "sha256,rsa2048"; 
                key-name-hint = "my_key"; 
                sign-images = "kernel,fdt"; 
                } 
        } 
    } 
};
}

▶ The my_key in key-name-hint node must be ${key_name}
Assemble control FDT for U-Boot with space for public key:
$ dtc -p 0x1000 u-boot.dts -O dtb -o u-boot.dtb

Generate fitImage with space for signature:
$ mkimage -D "-I dts -O dtb -p 2000" -f fit-image.its fitImage

Sign fitImage and add public key into u-boot.dtb:
$ mkimage -D "-I dts -O dtb -p 2000" -F -k "${key_dir}" -K u-boot.dtb -r fitImage

Signing subsequent fitImage:
$ mkimage -D "-I dts -O dtb -p 2000" -k "${key_dir}" -f fit-image.its -r fitImage

Now rebuild U-Boot, update both U-Boot and u-boot.dtb on the board and verify that U-Boot correctly starts.
Load the signed fitImage and use bootm start (or iminfo):

- **Verification passed (+ sign):**
  
  Verifying Hash Integrity ...
  sha256,rsa2048:my_key+ OK

- **Verification failed (- sign):**
  
  Verifying Hash Integrity ...
  sha256,rsa2048:my_key- Failed to verify required signature 'key-my_key'
Signed fitImage looks a bit difficult to assemble
Difficult part is done only once
The u-boot.dtb must be in read-only storage
Loading the kernel image

- Use the `load` command for all but NAND
- Use the `ubi*`/`ubifs*` commands for NAND
- The `fitImage` will assure that the image was not tampered with
In Linux

- Use Linux Integrity framework (IMA/EVM)
- Use UBI/UBIFS for RAW flash-based media
UBI/UBIFS

- UBI is not full solution against silent corruption
- UBI does not actively refresh the content on flash
  ⇒ Irrepairable corruption can still happen!
  ⇒ Implement a "scrubber" job:
    
    ```
    $ find / -exec cat {} > /dev/null 2>&1
    ```

! UBI does not support MLC NAND
Encryption support

- Encryption of U-Boot (using BootROM)
- Encryption of U-Boot environment
  - U-Boot has CONFIG_ENV_AES
  - Implement env_aes_cbc_get_key
- Encryption of kernel image
  - U-Boot has CONFIG_CMD_AES
  - Use aes dec
- Encryption of filesystem (use dm_crypt)
Thank you for your attention!

Contact: Marek Vasut <marex@denx.de>