<!--
#

Licensed to the Apache Software Foundation (ASF) under one

---

or more contributor license agreements. See the NOTICE file

---

distributed with this work for additional information

---

regarding copyright ownership. The ASF licenses this file

---

to you under the Apache License, Version 2.0 (the

"License"); you may not use this file except in compliance

---

with the License. You may obtain a copy of the License at

#

http://www.apache.org/licenses/LICENSE-2.0

#

Unless required by applicable law or agreed to in writing,

---

software distributed under the License is distributed on an

"AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY

KIND, either express or implied. See the License for the

---

specific language governing permissions and limitations

---

under the License.

#
-->

Encrypted images

Rationale

To provide confidentiality of image data while in transport to the device or while residing on an external flash, `MCUBoot` has support for encrypting/decrypting images on-the-fly while upgrading.

The image header needs to flag this image as `ENCRYPTED` (0x04) and a TLV with the key must be present in the image. When upgrading the image from the `secondary slot` to the `primary slot` it is automatically decrypted (after validation). If swap upgrades are enabled, the image located in the `primary slot`, also having the `ENCRYPTED` flag set and the TLV present, is re-encrypted while swapping to the `secondary slot`.

Threat model

The encrypted image support is supposed to allow for confidentiality if the image is not residing on the device or is written to external storage, eg a SPI flash being used for the secondary slot.

It does not protect against the possibility of attaching a JTAG and reading the internal flash memory, or using some attack vector that enables dumping the internal flash in any way.

Since decrypting requires a private key (or secret if using symmetric crypto) to reside inside the device, it is the responsibility of the device manufacturer to guarantee that this key is already in the device and not possible to extract.

Design

When encrypting an image, only the payload (FW) is encrypted. The header, TLVs are still sent as plain data.

Hashing and signing also remain functionally the same way as before, applied over the un-encrypted data. Validation on encrypted images, checks that the encrypted flag is set and TLV data is OK, then it decrypts each image block before sending the data to the hash routines.

The image is encrypted using AES-CTR-128, with a counter that starts from zero (over the payload blocks) and increments by 1 for each 16-byte block. AES-CTR-128 was chosen for speed/simplicity and allowing for any block to be encrypted/decrypted without requiring knowledge of any other block (allowing for simple resume operations on swap interruptions).

The key used is a randomized when creating a new image, by `imgtool` or `newt`. This key should never be reused and no checks are done for this, but randomizing a 16-byte block with a TRNG should make it highly improbable that duplicates ever happen.

To distribute this AES-CTR-128 key, new TLVs were defined. The key can be encrypted using either RSA-OAEP, AES-KW-128 or ECIES-P256.

For RSA-OAEP a new TLV with value `0x30` is added to the image, for AES-KW-128 a new TLV with value `0x31` is added to the image, and for ECIES-P256 a new TLV with value `0x32` is added. The contents of those TLVs are the results of applying the given operations over the AES-CTR-128 key.

ECIES-P256 encryption

ECIES follows a well defined protocol to generate an encryption key. There are multiple standards which differ only on which building blocks are used; for MCUBoot we settled on some primitives that are easily found on our crypto libraries. The whole key encryption can be summarized as:

- Generate a new secp256r1 private key and derive the public key; this will be our ephemeral key.
- Generate a new secret (DH) using the ephemeral private key and the public key that corresponds to the private key embedded in the HW.
- Derive the new keys from the secret using HKDF (built on HMAC-SHA256). We are not using a `salt` and using an `info` of `MCUBoot_ECIES_v1`, generating 48 bytes of key material.
- A new random encryption key of 16 bytes is generated (for AES-128). This is the AES key used to encrypt the images.
- The key is encrypted with AES-128-CTR and a `nonce` of 0 using the first 16 bytes of key material generated previously by the HKDF.
- The encrypted key now goes through a HMAC-SHA256 using the remaining 32 bytes of key material from the HKDF.

The final TLV is built from the 65 bytes of the ephemeral public key, followed by the 32 bytes of MAC tag and the 16 bytes of the encrypted key, resulting in a TLV of 113 bytes.

Since other EC primitives could be used, we name this particular implementation ECIES-P256 or ENC_EC256 in the source code and artifacts.

Upgrade process

When starting a new upgrade process, `MCUBoot` checks that the image in the `secondary slot` has the `ENCRYPTED` flag set and has the required TLV with the encrypted key. It then uses its internal private/secret key to decrypt the TLV containing the key. Given that no errors are found, it will then start the validation process, decrypting the blocks before check. A good image being determined, the upgrade consists in reading the blocks from the `secondary slot`, decrypting and writing to the `primary slot`.

If swap is used for the upgrade process, the encryption happens when copying the sectors of the `secondary slot` to the scratch area.

The `scratch` area is not encrypted, so it must reside in the internal flash of the MCU to avoid attacks that could interrupt the upgrade and dump the data.

Also when swap is used, the image in the `primary slot` is checked for presence of the `ENCRYPTED` flag and the key TLV. If those are present the sectors are re-encrypted when copying from the `primary slot` to the `secondary slot`.

PS: Each encrypted image must have its own key TLV that should be unique and used only for this particular image.

Also when swap method is employed, the sizes of both images are saved to the status area just before starting the upgrade process, because it would be very hard to determine this information when an interruption occurs and the information is spread across multiple areas.

Creating your keys with imgtool

`imgtool` can generate keys by using `imgtool genkey -k <output.pem> -t <type>`, where type can be one of `rsa-2048`, `rsa-3072`, `ecdsa-p256`, `ecdsa-p224` or `ed25519`. This will generate a keypair or private key.

To extract the public key in source file form, use `imgtool getpub -k <input.pem> -l <lang>`, where lang can be one of `c` or `rust` (defaults to `c`).

If using AES-KW-128, follow the steps in the next section to generate the required keys.

Creating your keys with Unix tooling

- If using RSA-OAEP, generate a keypair following steps similar to those described in signed_images to create RSA keys.
- If using ECIES-P256, generate a keypair following steps similar to those described in signed_images to create ECDSA256 keys.
- If using AES-KW-128 (`newt` only), the `kek` can be generated with a command like `dd if=/dev/urandom bs=1 count=16 | base64 > my_kek.b64`