Encrypted updates

Contents

The encryption of the update file makes accessing its contents more difficult for bystanders, but doesn't necessarily protect from more resourceful attackers that can extract the decryption key from the user-owned device.

The bundle encryption is done using the loop device with standard/proven kernel facilities for de/encryption (e.g. dm-crypt/LUKS). This allows the mechanism to be system agnostic (not tied to OSTree bundles), and can be used to ship updates to multiple components at once by including multiple files in the bundle.
dm-crypt is the Linux kernel module which provides transparent encryption of block devices using the kernel crypto API, see dm-crypt[1].
LUKS is the standard for Linux hard disk encryption. It provides secure management of multiple user passwords, see LUKS wiki[2].

The authenticity of the update is checked by verifying the OSTree signature as dm-crypt utilises symmetric cryptography which can't be used to ensure trust as the on-device key can be used to encrypt malicious files, not just decrypt them.

[1] https://gitlab.com/cryptsetup/cryptsetup/-/wikis/DMCrypt
[2] https://gitlab.com/cryptsetup/cryptsetup/-/wikis/home

Threat model

Objectives

1. end-users can download updates from the product website and apply them offline to the device via a USB key or SD card
2. only official updates should be accepted by the device
3. the contents of the updates should not be easily extracted, increasing the effort required for an attacker and providing some protection for the business' intellectual property

Properties

1. **integrity**: the device should only accept updates which have not been altered
2. **authenticity**: the device should only accept updates coming from the producer
3. **confidentiality**: the contents of the updates should not be disclosed

Threats

1. Alice owns a device and wants to make it run her own software
2. Emily owns a device and Alice wants to have her own software on Emily's device
3. Vincent develops a competing product and wants to gain insights into the inner workings of the device

Mitigations

1. **integrity**: the update is checksummed, causing alteration to be detectable
2. **authenticity**: the update is signed with a private key by the vendor and the device only accepts updates with a signature matching one of the public keys in its trusted set
3. **confidentiality**: the update is encrypted with a symmetric key (due to technology limitations public key decryption is not available)

Risks and impacts

the private key for signing is leaked

Impact

- the private key allows Alice to generate updates that can be accepted by all devices

Mitigations

- the private key is only needed on the vendor infrastructure producing the updates
- the chance of leaks is minimized by securing the infrastructure and ensuring that access to the key is restricted as much as possible
- public keys for the leaked private keys should be revoked
- multiple public keys should be trusted on the device, so if one is revoked updates can be rolled out using a different key
- keys should not be re-used across products to compartimentalize them against leaks

the private key for signing is lost

Impact

- updates can't be generated if no private key matching the on-device public ones is available

Mitigations

- if multiple public keys are trusted on the device, the private key used can be rotated if another private key is still available
- backup private keys should be stored securely in different locations

the symmetric key for encryption/decryption is leaked

Impact

- Alice has access to all symmetric keys stored in bundles encrypted with the leaked key
- the symmetric key allows Alice to generate updates that can be decrypted by devices

Mitigations

- due to its symmetric nature, the secret key has to be available on both the vendor infrastructure and on each device
- secure enclave technologies can help use the symmetric key for decryption without exposing the key in any way
- if secure enclave is not available the key has to be stored on the device and can be extracted via physical access
- if the key can't be provisioned in the factory the key has to be provisioned via unencrypted updates, from which an attacker can extract the keys without physical access to the device
- multiple decryption keys must be provisioned, to be able to rotate them in case of leaks

the symmetric key for encryption/decryption is lost

Impact

- encrypted updates can't be generated for devices only using this symmetric key

Mitigations

- given that the key has to be available on each device, the chance of losing the encryption/decryption key is small
- if multiple decryption keys are provisioned on the device, the encryption key can be rotated
- if all keys are lost or corrupted on the device, it will not be possible to decrypt bundles on USB/SDCard and so to update the device using this method.

Key infrastructure

LUKS is able to manage up to 8 key slots, any of the 8 different keys can be used to decrypt the update bundle. This can allow a bundle to be read using a main key or fallback key(s), and/or by different devices with a different subsets of the used keys.

On the device itself, Apertis Update Manager is in charge of decrypting the bundle and it will try as many keys as needed to unlock the bundle, there's no limitation on the number of keys which can be stored.

Random keys for bundle encryption can be generated using:

```
head -c128 /dev/random | base64 --wrap=0
```

How keys can be stored on devices

- Keys can be stored in separated files, located in read-only part of the filesystem: `/usr/share/apertis-update-manager/`
- In future versions, keys may be stored using the secure-boot-verified key storage system

How keys can be deployed to devices

- Keys stored in the filesystem can be deployed by the normal update mechanism

When new keys should be generated

New keys should be generated:

- for new products

- when a key has been compromised

How the build pipeline can fetch the keys

- As for the signing key, the key(s) used to encrypt the static delta bundle should be passed to the encryption script GitLab CI/CD variable(s)

How multiple keys can be used for key rotations

- When the keys are stored on the filesystem, key rotation will not provide any benefit as the leak of one key implies the leak of the others
- When the keys will be stored using the secure-boot-verified key storage system, the encrypted updates will be generated wih non-leaked keys and will remove the leaked keys while adding the new keys to the secure-boot-verified key storage system, so the number of available keys remain the same

How to handle the leak of a key to the public and how that impacts future updates

- If the keys are stored on the filesystem, the leak of one key implies the leak of the others
- If the keys are stored using the secure-boot-verified key storage system, the next update should be signed with a key that hasn't been leaked and the update should revoke the leaked key

Encryption Parameters

In a classical usage, the encryption is setup through a benchmark on the computer/board which will use it, allowing a good balance between password strength and unlocking time. This could end-up by encrypted file not usable due to out of memory error or slow unlocking time.

LUKS key strength is managed through 3 `cryptsetup` parameters: `--pbkdf-memory`, `--pbkdf-force-iterations` and `--pbkdf-parallel`.

`--pbkdf-parallel` configures the maximun number of threads used to unlock the encrypted file. This is automatically decreased on hardware devices that have only one of just a few cores.

As encrypted update file is created during image build on computer with more CPU power and memory, and that it is important to find a balance between password strength and usability, the `--pbkdf-memory` and `--pbkdf-force-iterations` should be forced to appropriate values for the target board.