

# SAN19-413: TEE based Trusted Keys in Linux

Sumit Garg



**Linaro  
connect**  
San Diego 2019

# Overview

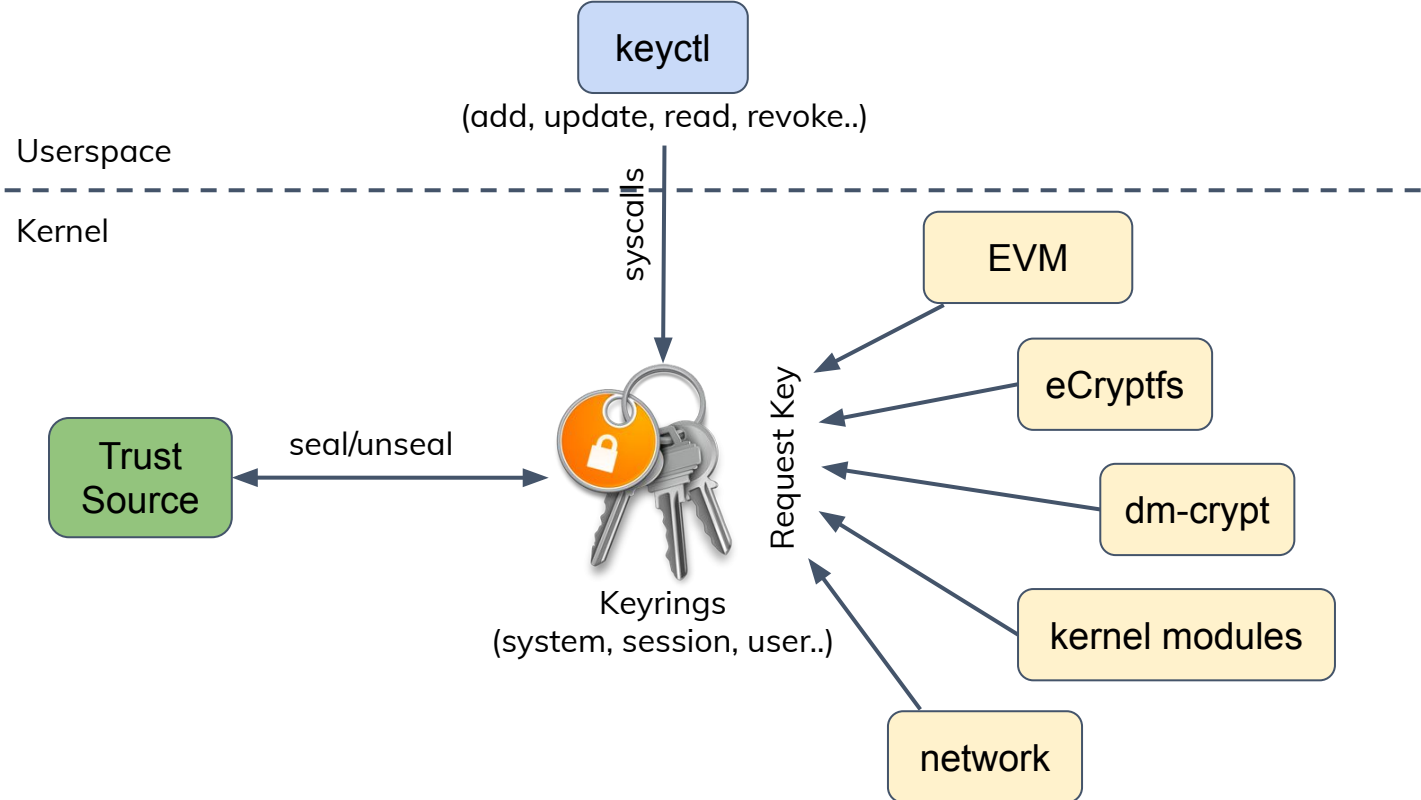
- Introduction to Kernel Keys
- Discuss Kernel Keys use-cases.
- How can we protect key confidentiality?
- Trusted and Encrypted Keys
  - Trusted Platform Module (TPM) as a trust source
- A system without a TPM device?
  - Trusted Execution Environment (TEE) as a trust source?
- New trust source: TEE device
  - Mechanism
  - Usage

# Kernel Keys

- **Kernel 'Key'**: A unit of cryptographic data, an authentication token, or some similar element represented in the kernel by “`struct key`”.
- **Kernel Key Retention Service**: This service allows keys to be cached in the kernel for the use of filesystems and other kernel services.
  - **Keyrings**: These are special keys that contain a list of other keys.

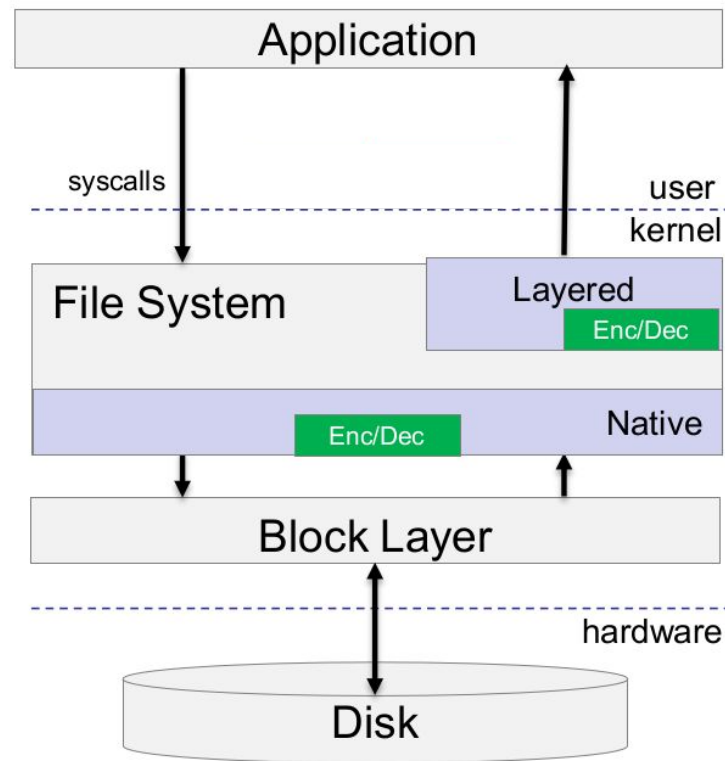
Our focus will be on cryptographic kernel keys used for services like disk encryption, file encryption and protecting the integrity of file metadata.

# Kernel Key Retention Service



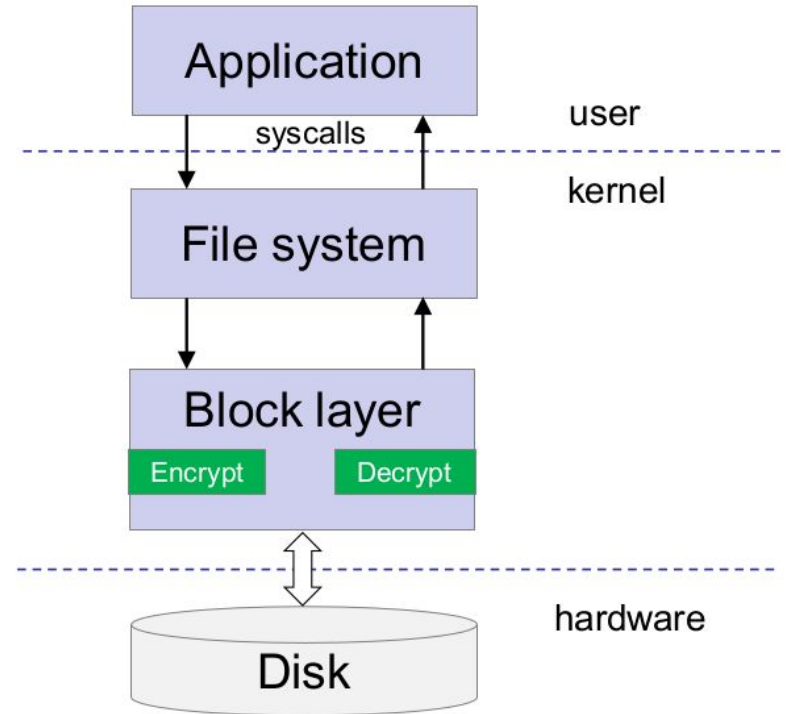
# Kernel Keys use-case: file system encryption

- File system encryption in kernel can be implemented at following levels:
  - Layered FS on top of native FS like eCryptfs.
  - In native FS, better performance
- Per-file encryption and key management.
- Master key used to wrap file encryption key.
- Trusted master key:
  - Protect against user-space compromises.



# Kernel Keys use-case: block layer encryption

- Encrypt everything on the disk – one master key for whole disk (volume) like dm-crypt.
- Trusted master key:
  - Protect against user-space compromises.



# Kernel Keys use-case: EVM

- EVM: Extended Verification Module
- EVM detects offline tampering of the security extended attributes, which are the basis for:
  - Linux Security Module (LSM) permission decisions.
  - Integrity Measurement Architecture (IMA) appraisal decisions.
- Standard “security” extended attributes:
  - security.ima (IMA's stored “good” hash for the file)
  - security.selinux (the selinux label/context on the file)
  - security.SMACK64 (Smack's label on the file)
  - security.capability (Capability's label on executables)
- At boot time, EVM needs a high quality symmetric key for HMAC protection of file metadata.
- Trusted EVM key:
  - Protect against user-space compromises.

# Trusted and Encrypted Keys

- Introduced in Linux kernel since v2.6.38.
- Variable length symmetric keys.
- Accessible in plain form to kernel only.
- User-space only sees, stores and loads them as encrypted blobs.
- Added two new key types in the kernel:
  - Trusted Key type
  - Encrypted Key type



# Trusted and Encrypted Keys

## Trusted Key

- Relies on a hardware based trust source like TPM.
- A random key generated using trust source's RNG.
- Trust source contains a secret key used to seal/unseal this key to/from encrypted blobs.
- Usage: mostly used as a master key for encrypted keys.

## Encrypted Key

- Doesn't depend on trust source like TPM, and are faster.
- A random key generated using kernel random numbers pool.
- Master key is used for wrapping:
  - Trusted Key
  - User Key
- Usage: most of Trusted-encrypted keys users request this key for actual encryption/decryption.



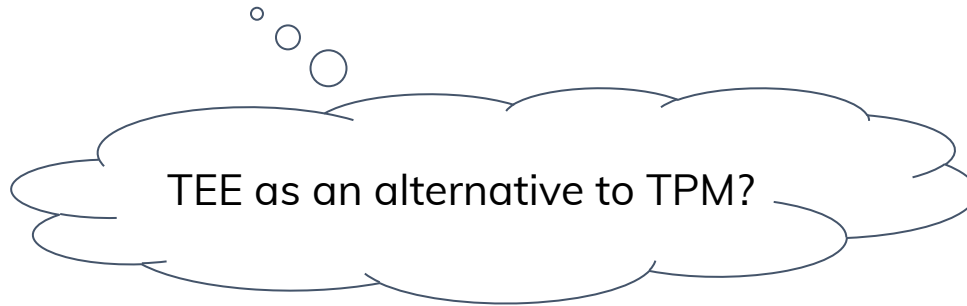
# A system without a TPM device?

Most of the available embedded systems doesn't possess a TPM device, likely reasons:

- Additional hardware, increases BoM cost.
- Constrained hardware resources.

An alternative could be a software based TPM but with following shortcomings:

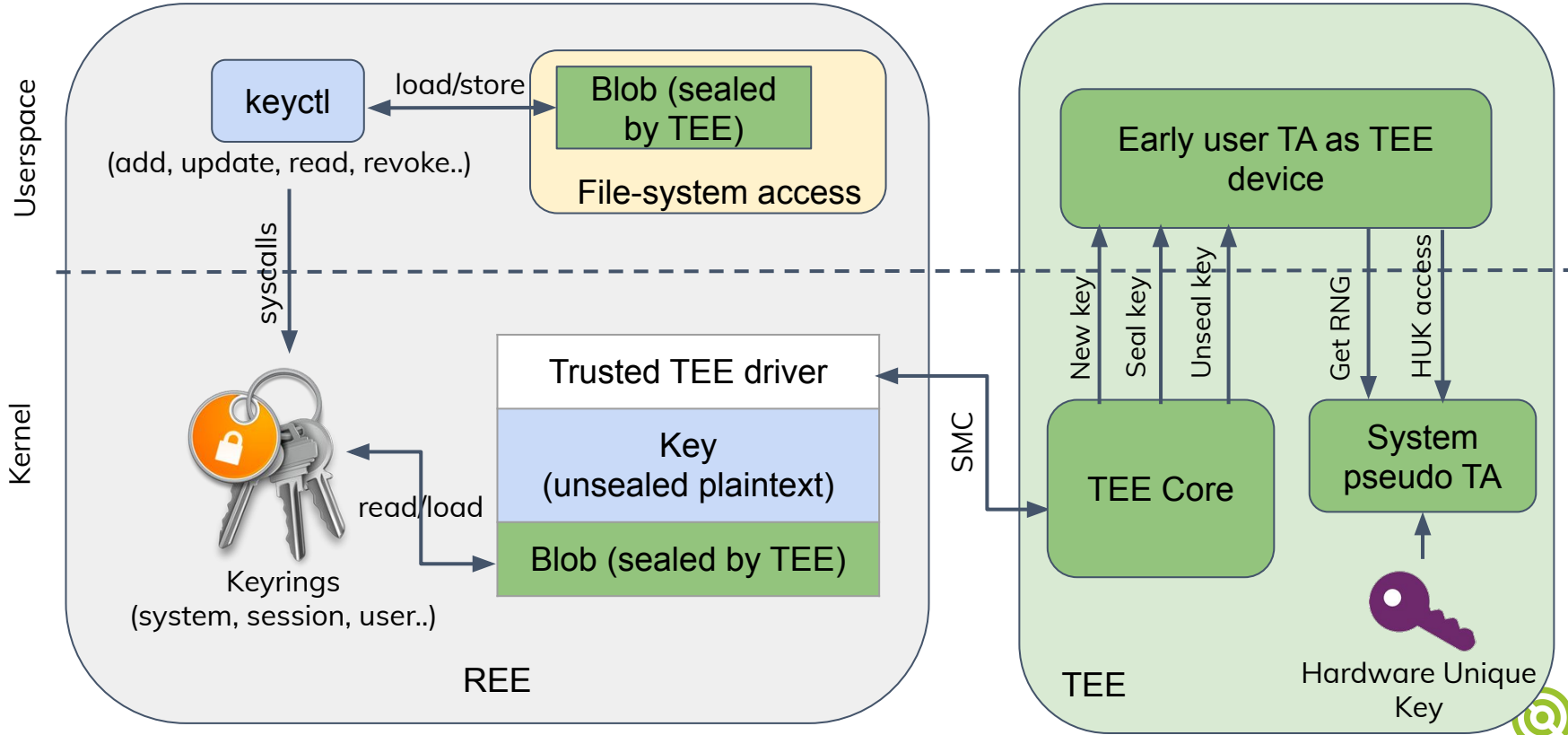
- Big and complicated software stack.
- Constrained devices with limited flash space, may be difficult to fit along with boot firmware.



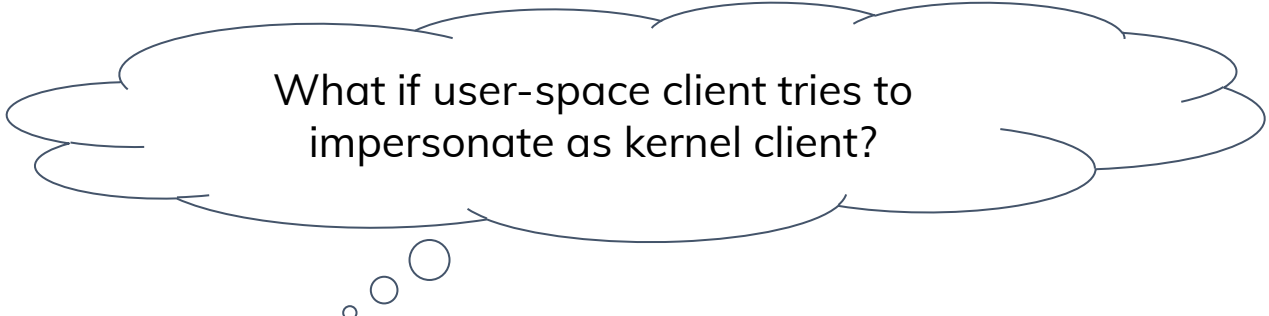
# TEE as a trust source?

- A Trusted Execution Environment based on ARM TrustZone provides hardware based isolation to perform trusted operations.
- Especially OP-TEE which offers a standardized API to exploit hardware unique key (HUK).
- HUK can be utilized to perform seal/unseal operations for Trusted keys.
- Sealed Trusted key blobs can be exported to user-space.
- Trusted key blobs can be unsealed on a particular hardware only.

# Trusted Keys: New TEE device



# A new TEE kernel client login method



What if user-space client tries to impersonate as kernel client?

- The TEE framework means that Trusted Applications are normally accessible to both kernel and userspace.
- Need to restrict user-space client access to Trusted Keys early TA service. So introduce a private REE kernel client login method:
  - TEE\_IOCTL\_LOGIN\_REE\_KERNEL

# TEE bus interface

Linux kernel provides a TEE bus interface to interact with TEE based services.

```
/**
 * struct tee_client_device_id - tee based device identifier
 * @uuid: For TEE based client devices we use the device uuid as
 *        the identifier.
 */
struct tee_client_device_id {
    uuid_t uuid;
};

struct bus_type tee_bus_type = {
    .name          = "tee",
    .match         = tee_client_device_match,
    .uevent        = tee_client_device_uevent,
};

EXPORT_SYMBOL_GPL(tee_bus_type);
```

# Trusted keys usage

Create a new trusted key:

```
$ keyctl add trusted kmk "new 32" @u  
$ keyctl show  
$ keyctl print <key serial no>
```

Save and load trusted key:

```
$ keyctl pipe <key serial no> > kmk.blob  
$ keyctl add trusted kmk "load `cat kmk.blob`" @u
```



# Encrypted keys usage

Create an encrypted key using trusted key as master key.

```
$ keyctl add encrypted evm "new trusted:kmk 32" @u
$ keyctl show
$ keyctl print <key serial no>
```

Save and load encrypted key:

```
$ keyctl pipe <key serial no> > evm.blob
$ keyctl add encrypted evm "load `cat evm.blob`" @u
```

# Next Steps

- Linux patch-set is already pushed in upstream, v2 is [here](#).
  - Followup.
- OP-TEE reference pseudo TA for testing is [here](#).
  - Need to convert this to an early TA using required support from system pTA.

# Thank you

Join Linaro to accelerate deployment of your  
Arm-based solutions through collaboration

[contactus@linaro.org](mailto:contactus@linaro.org)



9Boards is a range of specifications with boards and peripherals offering different performance levels and features in a standard footprint.



**Linaro**  
**connect**  
San Diego 2019

# TEE based devices

Device enumeration takes place during specific TEE driver probe (OP-TEE driver in this case). A particular TEE device is represented via following struct:

```
/**
 * struct tee_client_device - tee based device
 * @id:                device identifier
 * @dev:                device structure
 */
struct tee_client_device {
    struct tee_client_device_id id;
    struct device dev;
};
```

# TEE device drivers

Drivers register on TEE bus with a table of devices they support. A particular TEE device driver is represented via following struct:

```
/**
 * struct tee_client_driver - tee client driver
 * @id_table:           device id table supported by this driver
 * @driver:            driver structure
 */
struct tee_client_driver {
    const struct tee_client_device_id *id_table;
    struct device_driver driver;
};
```