System Security – Trusted Computing II

TDDE62 – Information Security: Privacy, System and Network Security

Ulf Kargén
Division for Database and Information Techniques (ADIT) at the Department of Computer and Information Science (IDA)

LINKÖPING UNIVERSITY

Recap of crypto primitives

Cryptographic hash – computes a fixed-length (e.g., 256 bits) *signature*, also called a *digest*, of a message

- Should not be possible to derive anything about M from **hash**(M).
- Given hash H, should be hard to find M such that **hash**(M) = H (pre-image resistance)
- Should be hard to find M_1 and M_2 such that **hash**(M_1) = **hash**(M_2) (collision resistance)

Hash-based Message Authentication Code (HMAC) – *keyed* hash for providing integrity/authenticity

- Conceptually: **hash**(key + M), where '+' denotes concatenation (in practice two rounds of hashing)
  - Send M and **HMAC** $_{\textbf{key}}$(M)
  - At receiver, recompute HMAC on M using shared secret key
  - Matching HMAC proves that sender knows key *and* that M has not been manipulated in transit

Recap of crypto primitives

Symmetric cryptography – encryption and decryption with *shared secret key*

- Well-known examples: AES, DES/3DES

Public key/asymmetric cryptography – uses a public/private key *pair*

- Messages encrypted with public key can only be decrypted by private key ***and vice versa***
- Can be used both for encryption and *signing*
- To **encrypt** message *M* with key pair pub/priv:
  - At sender: $\mathbf{encrypt}_{\mathbf{pub}}(M) \rightarrow C$ (anyone can encrypt with public key)
  - At receiver: $\mathbf{decrypt}_{\mathbf{priv}}(C) \rightarrow M$ (only holder of private key can decrypt)
- To **sign** message *M* with key pair pub/priv:
  - At sender: compute $\mathbf{encrypt}_{\mathbf{priv}}(\mathbf{hash}(M)) \rightarrow S$, send *M* and *S* (*S* is called a *signature* of *M*)
  - At receiver: compute $\mathbf{decrypt}_{\mathbf{pub}}(S)$ and compare with own hash of received *M*
    - If matching, proves that sender is the holder of priv
- Well-known examples: RSA and Elliptic Curve Cryptography (ECC)

Trusted computing technologies

Trusted Execution Environment – hardware features to allow strong isolation between trusted and non-trusted code running on a CPU

- ARM TrustZone, Intel SGX, AMD SEV, …

Secure Element – standalone tamper-resistant hardware (i.e., a chip) running its own software

- Allows e.g. protection of sensitive data, authenticity verification, etc. *without* a full-blown standalone TEE
- Security enabled by doing crypto operations in hardware and the ability to embed secrets (e.g. keys) that can *never leave the hardware chip*
- Examples: Smart cards, mobile-phone SIMs, vendor-specific solutions in smartphones

Trusted Platform Module (TPM)

- In a sense, a secure element built into modern PCs
- Developed by industry consortium Trusted Computing Group
- TPM 1.2 specification standardized in 2009
- TPM2.0 specification released in 2014
- Nowadays, all PCs have TPMs built-in to the CPU

TPM Features

- **Built-in crypto operations**
  - Cryptographically secure random number generator
  - Symmetric ciphers: AES, 3DES
  - Asymmetric ciphers: RSA and ECC
  - Cryptographic hash functions: SHA1 and SHA256
  - HMAC
  - Digital signature generation
- **Volatile RAM** (lost on reboots)
- **Nonvolatile storage (NVRAM)**

} Typically very limited in size…

- **Platform Configuration Registers (PCRs)** – used to record *measurements* of machine state – more on this later…

TPM APIs

- **Feature API (FAPI)**: High-level, but limited API for common tasks
- **Enhanced System API (ESAPI)**: Lower-level API for more control
- **System API (SAPI)**: 1-to-1 mapping to low-level hardware interface. Maximum control but very complex to use
  - We'll just give the names of TPM commands here, without going in to the details of the interface/API

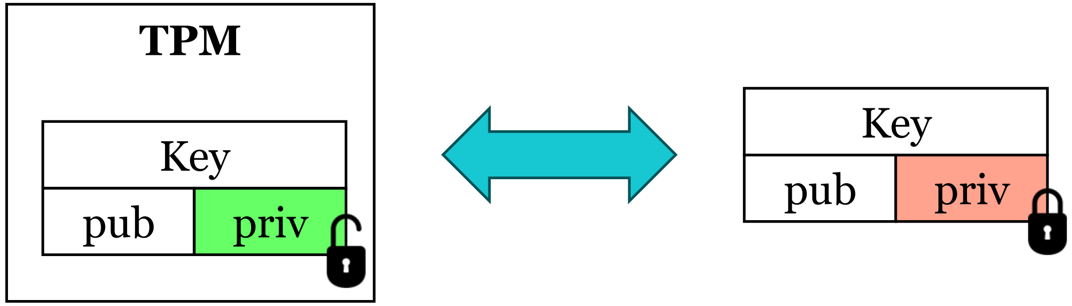
- **tpm2-tools suite**: Command-line tools that implement most of the SAPI functionality but hides *a lot* of its complexity
  - Allows interacting with a TPM directly from command line with no programming
  - Ships with the open-source TPM software stack used in e.g. Linux.

What we'll be using in the lab

TPM Keys

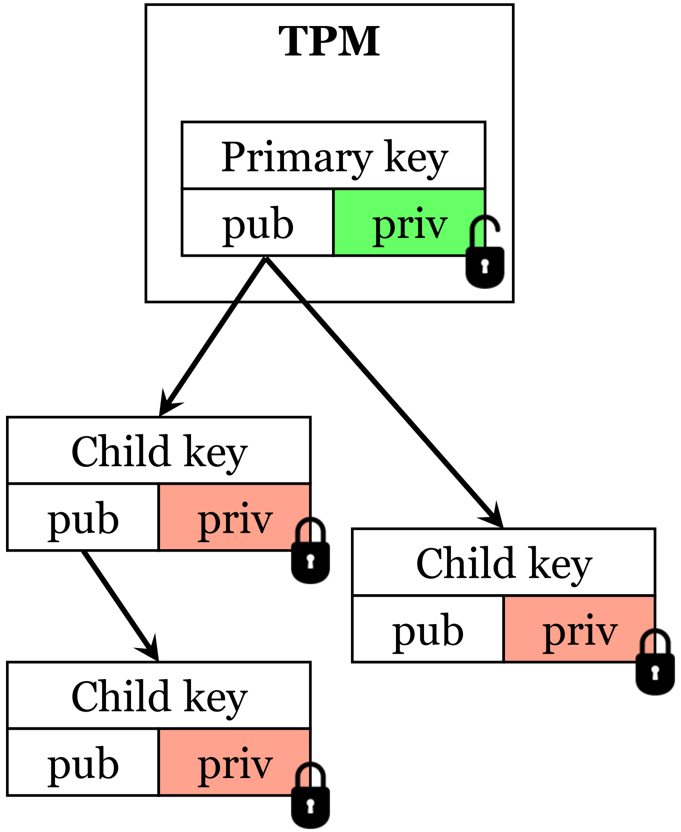
Safe storage of keys is a core function of the TPM

- No interface for reading out private or symmetric keys stored inside TPM
- Possible to store (some) keys *outside of TPM*
  - Often the favored approach – remember that storage in TPM is limited
- However, **private part of key never resides outside of TPM in unencrypted form**
- Externally stored keys can be loaded into TPM with `TPM2_Load`
- Somewhat confusingly, symmetric keys also have separate public and private entries in TPM data structures
  - Private part is the actual key
  - Public part contains metadata about key



Key wrapping

- **Primary keys** always reside inside the TPM
- A primary key can *wrap* (or "*bind*") **child** keys
  - Private key of child is encrypted using parent key
- Child keys can have child keys of their own
- Keys used to wrap other keys are called **storage keys**
- Wrapping works by encrypting private part of key with its storage key
  - Allows storing child keys *outside of TPM*
- Primary key serve as Root of Trust for all keys under it
- Loading a key requires specifying its parent key (and also loading the parent, unless it's a primary key)



Key hierarchies

The TPM has 4 *key hierarchies*

- Each hierarchy has its own **seed**, which is used to **derive primary keys** for that hierarchy
  - The seeds are the **actual embedded secrets** on which the security of all TPM functionality rests
  - All primary keys are integrity protected (HMAC:ed) based on key derived from seed
    - **Resetting a seed invalidates all keys** (internally or externally stored) **on the corresponding hierarchy**

- **Endorsement hierarchy** – seed set during manufacture
  - User cannot reset seed or change keys on this hierarchy
- **Platform hierarchy** – seed set by OEM during initial setup
  - Also locked for user
- **Owner hierarchy** – seed can be reset by owner/user of machine with `TPM2_Clear`
  - Most TPM-based solutions work on this hierarchy
- **Null hierarchy** – seed reset on every reboot

Key creation

A number of parameters can be specified during key creation in a `TPMT_PUBLIC` template structure:

- **“Unique”** field (optional) – essentially a secondary user-provided seed
- Type/algorithm (e.g., AES, RSA, HMAC, etc.)
- **Decryption** or **signing**?
- **Restricted** or **unrestricted**?
- and a bunch more …

More on this soon …

Keys are derived using a **Key Derivation Function** (KDF) from seed of the given hierarchy and template

- KDF is **deterministic** – same “unique” field gives same key – unless seed is reset
  - Can be used to re-generate keys instead of storing them permanently in TPM

Key creation

`TPM2_CreatePrimary` creates and **automatically loads** a primary key into volatile RAM

Child keys are created with `TPM2_Create` – requires specifying the parent key

- Returns an encrypted (wrapped) "key blob" for external storage. Must be loaded with `TPM2_Load`.
- `TPM2_Import` works similarly to `TPM2_Create`, but allows importing existing key instead of generating new one with KDF
- `TPM2_Create` can also be used to protect arbitrary data by wrapping it with a storage key

Note that `TPM2_CreatePrimary` only loads key into *volatile* RAM – will be lost on reboot!

- Use `TPM2_EvictControl` to move keys (and other data) to/from NVRAM.

Many TPM commands take **handles** (typically expressed as a hexadecimal number) as arguments – used to refer to a location in volatile RAM or NVRAM

- Reference to loaded keys/data or NVRAM locations
- Different handle ranges for different hierarchies and for NVRAM

Key parameters

Keys can be **restricted** or **unrestricted**

- Unrestricted *decryption* keys can be used to encrypt/decrypt and return arbitrary data
- Restricted decryption keys only allow decrypting data blobs **created by TPM** that will **reside inside the TPM** after decryption
  - Storage keys are always restricted keys – otherwise possible to break key wrapping!
- Unrestricted *signing* keys can be used for signing arbitrary data – restricted signing keys can only sign internally generated data from the TPM (e.g., the *quote* mechanism that we will see later)

Keys can also be created for signing data: `TPM2_Sign` and `TPM2_VerifySignature`

- Restricted keys cannot be used for both signing and encryption at once
  - Recall that RSA encryption and decryption is the same operation (exponentiation)
  - Calling `TPM2_Sign` on encrypted data would actually decrypt it!

Typical TPM setup

The **Endorsement Key** (EK) is the primary key on the **endorsement hierarchy**

- Authenticity of EK is certified by a certificate stored in reserved slot in NVRAM
- Can be used to verify that we're talking to a genuine TPM

Two important keys on the **owner hierarchy** (setup by owner/admin of machine):

- **Storage Root Key** (SRK) – primary storage key on owner hierarchy
- **Attestation Identity Key** (AIK) – restricted signing key created under SRK

Common crypto operations

Encrypt/decrypt:

- `TPM2_EncryptDecrypt` (symmetric ciphers)
- `TPM2_RSA_Encrypt`/`TPM2_RSA_Decrypt`
  - and similar for ECC
- Signing:
  - `TPM2_Sign`
  - `TPM2_VerifySignature`
- Hashing:
  - `TPM2_Hash`
  - `TPM2_HMAC`

Platform Configuration Registers (PCRs)

Special-purpose, fixed-size registers built into TPM

- Initialized to zero on every boot
- Only support two operations: **read** and **extend**
  - Extend operation is used to record **measurements** (hash digests of something) into a hash chain: $\mathbf{hash_A}(\mathrm{PCR} + \mathbf{hash_B}(\mathit{input})) \rightarrow \mathrm{PCR}$
  - Note that $\mathbf{hash_A}$ and $\mathbf{hash_B}$ doesn't necessarily have to be same hash function, but digest sizes need to be the same
- Different **banks** for different PCR hash functions (i.e., the $\mathbf{hash_A}$ above)
  - All TPMs support at least SHA1 and SHA256 PCRs
  - 24 PCRs per bank

Hash chain is **deterministic** – sequence of PCR extends can be used to verify system integrity

PCRs – Measured Boot

PCRs can be used for **sealing** data/keys to a certain system state, or for **remote attestation** of system integrity (i.e., proving that system isn't compromised)

- Attestation relies on a mechanism called **measured boot**: each element in the boot process measures its successor's code and data regions before handing off execution to that element
  - Firmware/BIOS: **extend**(PCR_i, **hash**(*Boot loader*))
  - Boot loader: **extend**(PCR_i, **hash**(*OS kernel*))
  - …
- First component in chain must be trusted a priori: the **Core Root of Trust for Measurement** (CRTM)
- Different PCRs are reserved for measuring different parts of the boot process
- Also record measurement details (including hash digests) into **boot log**
  - Allows tracing back offending component in case PCR doesn't match the expected at end of boot
  - Can be verified by replaying (outside of the TPM) hash chain

PCRs – Measured Boot

Remote attestation makes use of **quoting** mechanism to securely send PCR values to remote verifier

`TPM2_Quote` signs, with private half K_{priv} of **restricted signing key**, e.g., the AIK:

- A set of PCRs from a given bank
- A nonce (= a random number) sent by verifier (to prevent replay attacks)

After receiving signed set of PCRs, verifier checks signature using K_{pub}

Since entire operation happens inside TPM, quote can be trusted **even if machine is, e.g., infected with rootkit**

- Worst thing attacker can do is to DoS remote attestation attempts, but then host would remain untrusted anyway

Authorization

TPM has two types of authorization to prevent unintended access to e.g. keys

- **Hierarchy Authorization** – protects access to an entire hierarchy using a password
- **Key Authorization** – protects access to individual keys, set during key creation
  - In addition to simple passwords, also:
    - **HMAC authorization**: HMAC computed from shared secret + nonce. Safer than cleartext passwords.
    - **Policy authorization**: Can involve complex combinations of different kinds of authentication mechanisms
      - Input from biometrics, smartcard readers, etc.
      - Requiring specific PCR values

TPMs also have a **brute force/dictionary attack protection**

- When authentication fails more than a configurable number of times (e.g. 3), the TPM enters a locked-out mode – does not accept commands during a configurable **lockout time**
  - Possible to change parameters and recover from lockout mode with a **lockout password**

Sealing

"Lock" a key (or other data object) to a **specific system state**

- Unsealing only possible if a given **set of PCRs have the correct values**
- For example, make sure that a key is only usable if system is in known trusted state

Makes use of the **policy authorization** functionality

- First, create policy digest by
  - Using `TPM2_StartAuthSession` to start a **trial session** to record **expected** system state
  - Specify expected set of PCRs to consider and their values with `TPM2_PolicyPCR`
  - Finally, get a so-called **policy digest** with `TPM2_PolicyGetDigest`
- Then, provide policy digest as argument to `TPM2_CreatePrimary`, `TPM2_Create`, `TPM2_Import`

To unseal:

- Use `TPM2_StartAuthSession` to start a **policy session**
- Call `TPM2_PolicyPCR` to retrieve **actual** PCR values
- Can now use primary key or load child key – succeeds only if PCRs have correct values

The tpm2-tools

You'll be using the tpm2-tools suite during the lab

- Set of commands that mostly mirror the corresponding TPM commands
- For example, `tpm2_createprimary` tool wraps call to the low-level command `TPM2_CreatePrimary`

Documentation at https://github.com/tpm2-software/tpm2-tools/tree/master/man

Uses output *context* files for "stringing together" commands – no "real" programming needed

Example: Create a primary RSA key, then an AES key under it, and finally load AES key:

```
tpm2_createprimary -C o -c primary.ctx
tpm2_create -C primary.ctx –G aes -u aeskey.pub -r aeskey.priv
tpm2_load -C primary.ctx -u aeskey.pub -r aeskey.priv -c loaded.ctx
```