CHAPTER 8

TPM Entities

A TPM 2.0 *entity* is an item in the TPM that can be directly referenced with a handle. The term encompasses more than objects because the specification uses the word *object* to identify a very specific subset of entities. This can be confusing, so this chapter briefly describes all of the entity types: permanent entities (hierarchies, the dictionary attack lockout mechanism, and PCRs); nonvolatile entities (NVRAM indexes), which are similar to permanent entities; objects (keys and data); and volatile entities (sessions of various types).

After this introduction, the following chapters discuss each entity and its uses in more detail. In particular, the next chapter delves into hierarchies, a collection of entities that are related and managed as a group.

Permanent Entities

A *permanent entity* is one whose handle is defined by the TPM specification and can't be created or deleted. In TPM 1.2, PCRs and the owner were the only permanent entities; the storage root key (SRK) did have a fixed handle but wasn't a permanent entity. In TPM 2.0, there are more: three persistent hierarchies, the ephemeral hierarchy, the dictionary attack lockout reset, PCRs, reserved handles, the plaintext password authorization session, and the platform hierarchy NV enable. The following sections discuss each in turn.

Persistent Hierarchies

TPM 2.0 has three persistent hierarchies (platform, storage, and endorsement), each referenced by a permanent handle: `TPM_RH_PLATFORM` (0x4000000C), `TPM_RH_OWNER` (0x40000001), and `TPM_RH_ENDORSEMENT` (x4000000B). Permission to use these hierarchies is granted through authorizations, so each has both an authorization value and a policy. Either can be changed at the will of the hierarchy's administrator (defined as anyone who can authorize such a change). The authorization value or policy value may change, but whenever we refer to, for example, the platform authorization, we mean the same entity. Persistent hierarchies can never be deleted, but they may be disabled by the administrator of the platform or the administrator of the hierarchy. These three hierarchies may have associated chains of keys and data, which can be wiped by clearing the hierarchy.

The next chapter describes the hierarchies in detail, including each hierarchy's management and use cases. At this point, it's sufficient to understand that the persistent hierarchies are permanent entities. They can't be created or deleted.

Other permanent entities similar to the hierarchies listed here are the ephemeral hierarchy and the dictionary attack lockout reset mechanism.

Ephemeral Hierarchy

TPM 2.0 has an ephemeral hierarchy called the `NULL` hierarchy, which is also referenced by a permanent handle: `TPM_RH_NULL` (0x40000007). This hierarchy is utilized when the TPM is being used as a cryptographic coprocessor, as described in Chapter 9. Its authorization value and policy are both always `NULL`.

Similar to the persistent hierarchies, the ephemeral hierarchy is permanent. It can't be deleted. However, unlike the persistent hierarchies, it's automatically cleared every time the TPM goes through a power cycle. See Chapter 9 for details.

Dictionary Attack Lockout Reset

Similar to the hierarchies is the dictionary attack lockout mechanism, which has the handle `TPM_RH_LOCKOUT` (0x4000000A). It also has both an authorization and a policy. Like the three persistent hierarchies, these authorizations can be changed at the will of the administrator of this hierarchy. It has no key or object hierarchy. Instead, this mechanism is used to reset the dictionary attack lockout mechanism if it has triggered, or to clear the `TPM_RH_OWNER` hierarchy. It generally represents the IT administrator of the TPM storage hierarchy.

EXAMPLE: FAILURE COUNT RESET

A TPM is configured to lock out a user for 24 hours after 5 password entry failures. *Lock out* means the user can't successfully authorize any entity that is subject to this dictionary attack protection. The user convinces an IT administrator this this wasn't an attack but rather was just a mistake. The administrator, using lockout authorization, resets the failure count so the user doesn't have to wait for the 24-hour lockout period to expire.

Platform Configuration Registers (PCRs)

The TPM has a number of PCRs, which are accessed using their index. Depending on the platform-specific specification, they can have one or more algorithms. They also have an authentication value and a policy, chosen by the specification (generally `NULL`), which may be used to change the value stored in the PCR via a PCR extend. Reading the value stored in a PCR doesn't require authentication. The PC Client platform specifies a minimum of 24 PCRs. Only one *bank* (a set of PCRs with the same hash algorithm) is mandatory, programmable to either SHA-1 or SHA-256 at boot time.

Because it's a permanent entity, there is no command to create or delete a PCR; you can only change its attributes or the PCR value. Chapter 12 discusses these permanent entities in detail.

Reserved Handles

Vendor-specific reserved handles may be present in a TPM if a platform-specific specification decides to use them. Such handles are meant to be used by a vendor in the case of a catastrophic security failure of the firmware in the TPM, allowing the TPM to testify to the hash of the software stored in the TPM. At the date of this writing, no reserved handles are specified by any platform specification.

Password Authorization Session

There is one session that is permanent as well, called a password authorization session at handle `TPM_RS_PW` (0x40000009). A caller uses this handle for plaintext password (as opposed to HMAC) authorization.

Platform NV Enable

The `TPM_RH_PLATFORM_NV` handle (0x4000000D) controls the platform hierarchy NV enable. When it's clear (disabled), access to any NV index in the platform hierarchy is denied.

NV indexes can belong to either the platform or the storage hierarchy. The storage hierarchy enable controls NV indexes in the storage hierarchy. However, the platform enable doesn't control platform hierarchy NV indexes. That uses is a separate control: platform NV enable. Having two controls permits independent control of the platform hierarchy (for example, keys) and these platform NV indexes.

USE CASE: STORING BOOT PARAMETERS

Platform firmware can use the TPM as a convenient NV space for boot parameters. This space must remain readable even if the TPM platform hierarchy is disabled.

Next let's examine some entities that are similar to permanent entities: nonvolatile indexes, which are nonvolatile but not architecturally defined.

Nonvolatile Indexes

An NVRAM index in a TPM is a nonvolatile entity. There is a certain amount of nonvolatile space in a TPM that a user can configure for storage. When configured, it's given an index and a set of attributes, chosen by the user.

NVRAM indexes aren't considered objects by the TPM specification, because they have more attributes than a standard object. Reading and writing them can be individually controlled. They can be configured as entities that look like PCRs, counters, or bit fields. They can be made into "write once" entities as well. Chapter 11 explains their properties and use cases.

NVRAM indexes have both an associated authorization value and an authorization policy. The authorization value can be changed at the will of the owner of the index, but the policy can't be changed once it's set at the creation of the NVRAM index. NVRAM indexes are associated with a hierarchy when they're created. Hence, when the hierarchy is cleared, the NVRAM indexes associated with that hierarchy are deleted.

Objects are similar to NVRAM in that they belong to a hierarchy and have data and authorization mechanisms, but they have fewer attributes.

Objects

A TPM object is either a key or data. It has a public part and perhaps a private part such as an asymmetric private key, a symmetric key, or encrypted data. Objects belong to a hierarchy. All objects have both associated authorization data and an authorization policy. As with NV indexes, an object's policy can't be changed after it's created.

When an object is used in a command, some commands are considered *administrative* and others are considered *user* commands. At object creation, the user decides which of these commands can be performed with the authorization data and which can exclusively be done with a policy. This comes with a caveat: certain commands can only be done with a policy no matter how the attributes are set at key creation.

Like NVRAM indexes, all objects belong to one of the four hierarchies: platform, storage, endorsement, or `NULL`. When a hierarchy is cleared, all objects belonging to that hierarchy are also cleared.

Typically, most objects are keys. They're described in detail in the Chapter 10. Using keys or other objects requires the use of a TPM non-persistent entity: the session.

Nonpersistent Entities

A nonpersistent entity never persists through power cycles.[1] Although a nonpersistent entity can be saved (see `TPM2_ContextSave`), a TPM cryptographic mechanism prevents the saved context from being loaded after a power cycle, thus enforcing volatility. This type of entity has several classes.

Authorization sessions, including HMAC and policy sessions, are perhaps the most widely used, permitting entity authorization, command and response parameter encryption, and command audit. Chapter 13 is devoted to their use.

[1]To be precise, it doesn't persist through what the specification refers to as a *TPM Reset* (a reboot). It does persist through a *TPM Restart* (resume from hibernate) or *TPM Resume* (resume from sleep).

Hash and HMAC event sequence entities hold the intermediate results of the typical crypto library "start, update, complete" design pattern. They permit the hashing or HMAC of data blocks that are larger than the TPM command buffer. Chapter 9 describes their application.

In contrast to a nonpersistent entity, a persistent entity persists through power cycles.

Persistent Entities

A *persistent entity* is an object that the owner of a hierarchy has asked to remain resident in the TPM through power cycles. It differs from a permanent entity (which can never be deleted) in that the owner of the hierarchy to which a permanent entity belongs can evict it. A TPM has a limited amount of persistent memory, so you should be sparing in your use of persistent entities. There are, however, some valuable use cases.

USE CASE: VPN KEY ACCESS

A signing key is needed for VPN access early in a boot cycle. At that time, the disk isn't available. The application transfers the key to TPM persistent storage, where it's immediately available for use in early boot cryptographic operations.

USE CASE: PRIMARY KEY OPTIMIZATION

A primary storage key (the equivalent to a TPM 1.2 SRK) is routinely used as the root of a key hierarchy. Key generation is often the most time-consuming cryptographic calculation. After creation, the key is moved to persistent storage to avoid the performance penalty of recalculating the key on every boot cycle.

USE CASE: IDENTITY KEY PROVISIONING

An enterprise provisions a motherboard with a restricted signing key that is fixed to the TPM. The enterprise uses this key to identify the platform. If the motherboard fails and the TPM is thus replaced, this existing key can no longer be loaded. The IT department wishes to provision spare motherboards with new signing keys. Because a motherboard has no disk, the IT department generates the key and moves it to TPM persistent storage. The signing key now travels with the motherboard when it replaces a failed one in a platform.

Usually, primary storage keys (such as an SRK), primary restricted signing keys (such as an attestation identity key [AIK]), and possibly endorsement keys (EK) are the only entities that remain persistent in a TPM. These are discussed in more detail in Chapter 10.

Entity Names

The *Name* of an entity is a TPM 2.0 concept, invented to solve a problem noticed with the TPM 1.2 specification. A paranoid security analyst (and all security analysts are paranoid) noticed that it might be possible for an attacker to intercept data as it was being sent to the TPM. The TPM design had protections against such an attack changing most data that was sent to the TPM. However, the TPM has very few resources, so it allowed a key manager to load and unload keys into the TPM as necessary. After keys were loaded, they were referred to by a handle, a shorthand for the location in memory where the key was loaded. Because the software might not realize that a key manager had been relocating keys in the TPM to free up space, the handle itself wasn't protected against manipulation, and middleware would patch the data that was sent to the TPM to point to the correct handle location.

Normally this wouldn't be a problem. But if someone decided to give the same password to more than one key, then it would be possible for one of those keys to be substituted for another by an attacker, and the attacker could then authorize the wrong key to be used in a command. You might think such an attack would be unlikely, but the people who wrote the TPM specification also tend to be paranoid and decided this was unacceptable behavior. Instead of just warning everyone not to use the same password for multiple keys, they decided to give every entity a unique Name, and that Name is used in the HMAC authorization calculation sent when executing a command that uses that entity. The handle may change, but the name doesn't.

The command parameter stream that is hashed and then HMACed implicitly includes the Name of each entity referred to by handle, even though the command parameters may not include the Name. An attacker can change the handle but can't change the corresponding Name value after it's authorized through the HMAC calculation.

The Name is the entity's unique identifier. Permanent entities (PCRs and hierarchy handles) have handles that never change, so their Name is simply their handle. Other entities (NV indexes and loaded objects) have a calculated name that is essentially a hash of the entity's public data. Both the TPM and caller independently calculate the Name value for use during authorization.

For security, it's extremely important that the Name is calculated and stored when the entity is created. A naïve implementation might offer to help by providing a "handle to Name" function that reads the TPM handle and uses the resulting public area to generate the Name. This would defeat the entire purpose of using the Name in the HMAC calculation, because the result is the Name of the entity currently at the handle, not the Name the authorizer expected.

Following are some examples of how the Name is used.

EXAMPLE: ATTACKER CLEARING A BIT-FIELD NV INDEX

A key owner uses an NV bit-field index in the key's policy, with a set bit 3 revoking the key for a key user. The revoked user / attacker deletes the NV index and re-creates it with the same policy. When the key owner wants to set bit 5, they use the handle-to-Name function to calculate the Name. The key owner uses the result for authorization, and sets bit 5. However, bit 3 is now clear because the TPM initializes bit fields to all bits clear.

If the key owner had properly stored the Name and used it for authorization, the authorization would fail. This would happen because, when the attacker re-created the index, the "written" bit in the public area attributes would go from set to clear, changing the Name on the TPM.

The Name of an NV index is a digest of its public area. An attacker can delete and redefine an index, but unless the public area (the index value, its attributes, and its policy) is the same, the Name will change and the authorization will not verify.

EXAMPLE: ATTACKER READING A SECRET

The user defines an ordinary index intended to hold a secret. The index policy is such that only the user can read the secret. Before the secret is written, an attacker deletes the index and redefines it with a different policy, such that the attacker can also read the secret. The attack fails because the policy change causes the Name to change. When the user tries to write the secret, the authorization fails because the original name was used to calculate the command parameter hash.

The Name of a transient or persistent entity is also a digest of its public area. The public area varies with the type of entity.

USE CASE: PERMITTING A RESOURCE MANAGER TO SECURELY MANAGE TPM KEYS

The user loads a key and receives back a handle for the loaded key. The user authorizes the key with an HMAC of the command parameters, which implicitly includes the Name. Unknown to the user, a resource manager had unloaded the key, and now loads it. Keep in mind, however, that the handle changes. The resource manager replaces the user's handle with the new handle value. The authorization still verifies, because the calculation didn't include the handle (which changed), only the Name (which didn't).

USE CASE: ATTACKER REPLACING A KEY AT THE SAME HANDLE

The user loads a key and receives a handle. The user authorizes that key with an HMAC. However, an attacker replaces the key on the TPM with their own key, and the attacker's key has the same handle. The attack fails because the attacker's key has a different Name, so the HMAC authorization fails.

Summary

The TPM has several types of entities: items that can be referred to by a handle. Permanent entities have a handle that is fixed by the TPM specification. The handle value can't change; nor can such an entity be created or deleted. Its data can be either persistent or volatile. Nonvolatile indexes can be created or deleted at a user-specified handle, and they persist through TPM power cycles. Objects — entities that are attached to a hierarchy — may have a private area and can be volatile or made persistent. When the object is made persistent, it's called a persistent entity.

An entity's Name is its unique identifier. It's used in authorization calculations rather than the entity's handle because the handle may change over time as a resource manager loads and flushes entities.

This chapter has summarized the entity types and provides a road map to the chapters that follow. The details come next, with chapters on hierarchies (permanent entities), keys (objects), NV indexes (persistent entities), PCRs (permanent entities), and sessions (nonpersistent entities).