CHAPTER 3

Quick Tutorial on TPM 2.0

This chapter describes the major uses of TPM capabilities. The use cases for which TPM 1.2 was designed still pertain to TPM 2.0, so we begin by exploring those use cases and the designed functionality that enables them. Then we move to new aspects of the TPM 2.0 design and the use cases enabled by those capabilities.

As noted in Chapter 1, the rise of the Internet and the corresponding increase in security problems, particularly in the area of e-business, were the main driving forces for designing TPMs. A hardware-based standardized security solution became imperative. At the same time, due to the lack of a legacy solution, security researchers were presented with a golden opportunity to design a new security system from the ground up. It has long been a dream of security architects to not merely patch problems that existed in earlier designs, but also provide a security anchor on which new architectures can be built.

The TPM 1.2 specification was the Trusted Computing Group's (TPG's) first attempt to solve this problem and was aimed at addressing the following major issues in the industry:

- *Identification of devices*: Prior to the release of the TPM specification, devices were mostly identified by MAC addresses or IP addresses—not security identifiers.
- *Secure generation of keys*: Having a hardware random-number generator is a big advantage when creating keys. A number of security solutions have been broken due to poor key generation.
- *Secure storage of keys*: Keeping good keys secure, particularly from software attacks, is a big advantage that the TPM design brings to a device.
- *NVRAM storage*: When an IT organization acquires a new device, it often wipes the hard disk and rewrites the disk with the organization's standard load. Having NVRAM allows a TPM to maintain a certificate store.
- *Device health attestation*: Prior to systems having TPMs, IT organizations used software to attest to system health. But if a system was compromised, it might report it was healthy, even when it wasn't.

The TPM 2.0 implementations enable the same features as 1.2, plus several more:

- *Algorithm agility*: Algorithms can be changed without revisiting the specification, should they prove to be cryptographically weaker than expected.
- *Enhanced authorization*: This new capability unifies the way all entities in a TPM are authorized, while extending the TPM's ability to enable authorization policies that allow for multifactor and multiuser authentication. Additional management functions are also included.
- *Quick key loading*: Loading keys into a TPM used to take a relatively long time. They now can be loaded quickly, using symmetric rather than asymmetric encryption.
- *Non-brittle PCRs*: In the past, locking keys to device states caused management problems. Often, when a device state had to go through an authorized state change, keys had to be changed as well. This is no longer the case.
- *Flexible management*: Different kinds of authorization can be separated, allowing for much more flexible management of TPM resources.
- *Identifying resources by name*: Indirect references in the TPM 1.2 design led to security challenges. Those have been fixed by using cryptographically secure names for all TPM resources.

TPM 1.2 was a success, as indicated by the fact that more than 1 billion TPMs using the 1.2 specification have been deployed in computer systems. TPM 2.0 expands on TPM 1.2's legacy. Currently, many vendors are developing implementations for TPM 2.0, and some are shipping them. Microsoft has a TPM 2.0 simulator that can also act as a software implementation of TPM 2.0. Some vendors are in the process of sampling hardware TPMs, and other companies are working on firmware TPMs.

Scenarios for Using TPM 1.2

In general, the TPM 2.0 design can do anything a TPM 1.2 chip can do. Thus, in considering applications that can use a TPM 2.0 chip, it's wise to first examine the applications that were enabled by the TPM 1.2 design.

Identification

The use envisioned for the first embedded security chip was device identification (DeviceID). Smart cards use their keys for this purpose. The private key embedded in the chip identifies the card on which it resides, an authentication password or PIN is used to authenticate a person to the card, and together they form "the thing you have" and "the thing you know" for authentication. Nothing keeps several people from using the same smart card, as long as they all know the PIN. There is also nothing that ties the smart card to a particular machine, which is an advantage when the smart card is used as a proxy for identifying an individual instead of a machine.

By embedding a private key mechanism in a personal computing device, that device can be identified. This is a big advantage for an IT organization, which owns the device and is in control of its software load and security protections. But as computers became more portable with the production of smaller and lighter laptops, the PC itself began to be useful as "the thing you have" in place of a smart card. It turned out that many times, when a smart card was used to authenticate a person to a computer network, the user left the smart card with the device. If one was stolen, both were stolen. As a result, there was no advantage to keeping the two separate.

However, if the password of a key stored in a security chip inside a personal computer was going to be used as a proxy for an individual, it was clear that the key could not reside in a single computer. The key has to be able to exist in multiple machines, because individuals tend to use more than one device. Further, machines are upgraded on average every 3 to 5 years, and keys must move from an old system to a new system in order to make system management possible.

These realizations led to two of the objectives of the original embedded security chips. They needed keys that identified the device—keys that couldn't be moved to different machines. And they needed keys that identified individuals—keys that could be duplicated across a number of machines. In either case, the keys had to be able to be deleted when an old system was disposed of.

What is the identification used for? There are a large number of uses, including these:

- *VPN identifying a machine before granting access to a network*: An IT organization can be certain that only enterprise-owned machines are allowed on the enterprise's network.
- *VPN identifying a user before granting access to a network*: An IT organization can be certain that only authorized personnel are granted access to an enterprise's network.
- *User signing e-mail*: The recipient of an e-mail can know with some certainty who sent the e-mail.
- *User decrypting e-mail sent to them*: This allows for confidentiality of correspondence.
- *User identifying themselves to their bank*: A user can prevent others from logging in to their account.
- *User authorizing a payment*: A user can prevent others from making payments in their name.
- *User logging in remotely to a system*: Only authorized personnel can log in to a remote system.

Encryption

The second use case for a security chip embedded on systems was to provide a means of encrypting keys that were used in turn to encrypt files on the hard drive or to decrypt files that arrived from other systems. Export regulations made putting a bulk encryption/decryption engine in the security chip a nonstarter; but using the chip to store encryption keys was allowed, so that functionality was included. The chip already had to do public/private encryption in order to perform cryptographic signing, so it was inexpensive to add the ability to decrypt a small amount of data containing a key that was encrypted with a public key, if the chip knew the private portion of the key.

Once this basic capability was available, it enabled a number of scenarios such as the following:

- File and folder encryption on a device
- Full disk encryption
- Encryption of passwords for a password manager
- Encryption of files stored remotely

Key Storage

One basic question the designers of the TPM had for possible users was, “How many keys do you think people will want to use with this chip?” If the answer had been “One or two,” there would have been sufficient room in the chip to store those keys. However, the answer received was “More than three.” Thus cost reasons made it infeasible to store all the keys on the chip, as was done in a smart card. However, the chip is used in PCs, which have hard disks and hence almost unlimited storage for keys—and TPG decided to make use of that fact.

The TPM has access to a self-generated private key, so it can encrypt keys with a public key and then store the resulting blob on the hard disk. This way, the TPM can keep a virtually unlimited number of keys available for use but not waste valuable internal storage. Keys stored on the hard disk can be erased, but they can also be backed up, which seemed to the designers like an acceptable trade-off. Cheap keys associated with a TPM enable a number of scenarios like these:

- *Privacy-sensitive solutions that use different keys to provide only a minimum of information to a requestor*: You don’t need a single identity key that includes a user’s age, weight, marital status, health conditions, political affiliation, and so on.
- *Different keys for different security levels*: Personal, financial, and business data as well as data that is contractually restricted all require different levels of confidentiality.
- *Different keys for multiple users of the same PC*: Sometimes several people share a computer. If that is the case, they typically don’t want to give each other complete access to their files.
- *“Hoteling” of PCs in an office*: Keys are stored on a server and downloaded and used on a PC as required.

Random Number Generator

In order to generate keys, a random number generator (RNG) is necessary, and early PCs generally didn't contain good RNGs. There have been several cases where poor key generation was used to break security protocols. This is true. So the standards body required that a RNG be one of the components of the first TPM.

> *Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.*[1]
>
> —Von Neumann

There are many uses for a good RNG:

- Seeding the OS random number generator
- Generating nonces (random numbers) used in security protocols
- Generating ephemeral (one-time use) keys for file encryption
- Generating long-term use keys (such as keys used for storage)
- Seeding Monte Carlo software routines

NVRAM Storage

A small amount of NVRAM storage that has restricted access-control properties can be very useful in a PC. It can store keys that shouldn't be available when the machine is off, give faster access to data than decryption using public/private key pairs can, and provide a mechanism to pass information back and forth between different parts of a system. NVRAM in TPMs can be configured to control read and write capabilities separately, which means some data can be provided to a user without worrying that it will be erased by accident or malicious intent. Additionally, you can use NVRAM to store keys that are used when the PC doesn't have access to its main storage. This can happen early during the boot cycle or before a self-encrypting drive has been given its password, allowing it to be read.

Having NVRAM provides the following:

- *Storage for root keys for certificate chains*: These are public keys to which everyone should have access—but it's very important that they not be changed.
- *Storage for an endorsement key (EK)*: An EK is stored by the manufacturer and used to decrypt certificates and pass passwords into the TPM during provisioning. In spite of misleading statements made on the Internet, the EK was designed to be privacy sensitive.
- *Storage for a representation of what the state of the machine ought to be*: This is used by some Intel implementations using TPMs and Intel Trusted Execution Technology (TXT), where

[1]*Monte Carlo Method* (1951), John von Neumann.

it's called a *launch control policy*. Like the public root key used in Unified Extensible Firmware Interface (UEFI) secure-boot implementations, this is used by the system owner to specify the state they want the machine to be in when it goes through a controlled launch, usually of a hypervisor. The advantage over the UEFI secure-boot method is that with the TPM, the end user has full control over the contents of the NVRAM storage.

- *Storage for decryption keys used before the hard disk is available*: For example, a key used for a self-encrypting drive.

Platform Configuration Registers

One unique thing about a TPM that can't be guaranteed with smart cards is that it's on the motherboard and available before the machine boots. As a result, it can be counted on as a place to store measurements taken during the boot process. Platform Configuration Registers (PCRs) are used for this purpose. They store hashes of measurements taken by external software, and the TPM can later report those measurements by signing them with a specified key. Later in the book, we describe how the registers work; for now, know that they have a one-way characteristic that prevents them from being spoofed. That is, if the registers provide a representation of trusted software that behaves as expected, then all the register values can be trusted.

A clever thing that's done with these registers is to use them as a kind of authentication signal. Just as, for example, a time lock won't allow a bank vault to unlock unless the time is during business hours, you can create a key or other object in a TPM that can't be used unless a PCR (or PCRs) is in a given state. Many interesting scenarios are enabled by this, including these:

- A VPN may not allow a PC access to a network unless it can prove it's running approved IT software.
- A file system may not obtain its encryption key unless its MBR has not been disturbed and the hard disk is on the same system.

Privacy Enablement

The architects of the first TPM were very concerned about privacy. Privacy is of major importance to enterprises, because losing systems or data that contain personally identifiable information (PII) can cause an enormous loss of money. Laws in many states require enterprises to inform people whose private data has been lost; so, for example, if a laptop containing a database of Human Resources data is stolen, the enterprise is required to notify everyone whose data might have been compromised. This can cost millions of dollars. Before the advent of embedded security systems, encryption of private files was nearly impossible on a standard PC because there was no place to put the key. As a result, most encryption solutions either "hid" the key in a place that was easily found by the technically adept, or derived a key from a password. Passwords have a basic problem: if a person can remember it, a computer can figure it out. The best way to prevent this is to have hardware track when too many wrong attempts are made to guess a password and then cause a delay before another attempt is allowed. The TPM specification requires this approach to be implemented, providing an enormous privacy advantage to those who use it.

The second privacy-related problem the architects tried to solve was much harder: providing a means to prove that a key was created and was protected by a TPM without the recipient of that proof knowing which TPM was the creator and protector of the key. Like many problems in computer science, this one was solved with a level of indirection. By making the EK a decryption-only key, as opposed to a signing key, it can't be (directly) used to identify a particular TPM. Instead, a protocol is provided for making *attestation identity keys* (AIKs), which are pseudo-identity keys for the platform. Providing a protocol for using a privacy CA means the EKs can be used to prove that an AIK originated with a TPM without proving which TPM the AIK originated from. Because there can be an unlimited number of AIKs, you can destroy AIKs after creating and using them, or have multiple AIKs for different purposes. For instance, a person can have three different AIKs that prove they're a senior citizen, rich, and live alone, rather than combining all three into one key and exposing extra information when proving one of their properties.

Additionally, some clever cryptographers at Intel, IBM, and HP came up with a protocol called direct anonymous attestation (DAA), which is based on group signatures and provides a very complicated method for proving that a key was created by a TPM without providing information as to which TPM created it. The advantage of this protocol is that it lets the AIK creator choose a variable amount of knowledge they want the privacy CA to have, ranging from perfect anonymity (when a certificate is created, the privacy CA is given proof that an AIK belongs to a TPM, but not which one) to perfect knowledge (the privacy CA knows which EK is associated with an AIK when it's providing a pseudonymous certificate for the AIK). The difference between the two is apparent when a TPM is broken and a particular EK's private key is leaked to the Internet. At this point, a privacy CA can revoke certificates if it knows a certificate it created is associated with that particular EK, but can't do so if it doesn't know.

PCR sensitivity to small changes in design, implementation, and use of PCs makes PCRs for the most part irreversible. That is, knowing a PC's PCR values provides almost no information about how the PC is set up. This is unfortunate for an IT organization that notices a change in PCR values and is trying to figure out why. It does provide privacy to end users, though.

Scenarios for Using Additional TPM 2.0 Capabilities

Lessons learned in the use of TPM 1.2 led to a number of changes in the architecture of TPM 2.0. In particular, the SHA-1 algorithm, on which most 1.2 structures were based, was subjected to cryptographic attacks. As a result, the new design needed to not be catastrophically broken if any one algorithm used in the design become insecure.

Algorithm Agility (New in 2.0)

Beginning in TPM 2.0, the specification allows a lot of flexibility in what algorithms a TPM can use. Instead of having to use SHA-1, a TPM can now use virtually any hash algorithm. SHA 256 will likely be used in most early TPM 2.0 designs. Symmetric algorithms like Advanced Encryption Standard (AES) are also available, and new asymmetric algorithms such as elliptic curve cryptography (ECC) are available in addition to RSA.

The addition of symmetric algorithms (enabled by the weakening of export-control laws) allows keys to be stored off the chip and encrypted with symmetric encryption instead of asymmetric encryption. With this major change to the method of key storage, TPM 2.0 allows any kind of encryption algorithm. This in turn means if another algorithm is weakened by cryptanalysis in the future, the specification won't need to change.

Ideally, the key algorithms should be matched in strength. Table 3-1 lists the key strengths of approved algorithms according to the National Institute of Standards and Technology NIST).[2]

Table 3-1. *Approved algorithms*

Type	Algorithm	Key strength (bits)
Asymmetric	RSA 1024	80
Asymmetric	RSA 2048	112
Asymmetric	RSA 3072	128
Asymmetric	RSA 16384	256
Asymmetric	ECC 224	112
Asymmetric	ECC 256	128
Asymmetric	ECC 384	192
Asymmetric	ECC 521	260
Symmetric	DES	56
Symmetric	3DES (2 keys)	127
Symmetric	3DES (3 key)	128
Symmetric	AES 128	128
Symmetric	AES 256	256
Hash	SHA-1	65
Hash	SHA 224	112
Hash	SHA 256	128
Hash	SHA 384	192
Hash	SHA 512	256
Hash	SHA-3	Variable

[2]NIST, "Recommendation for Key Management – Part 1: General (Revision 3)," Special Publication 800-57, `http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57_part1_rev3_general.pdf`.

AES is typically used for the symmetric algorithm today. At 128 bits, the two most frequently used asymmetric algorithms are RSA 2048 or ECC 256. RSA 2048 isn't quite as strong as ECC 256 and is much slower. It also takes up a lot more space. However, the patents on RSA have expired, and it's compatible with most software today, so many people still use it. Many people are using RSA 2048 together with SHA-1 and AES-128, even though they're far from a matched set, because they're free and compatible. Most of the examples in this book use both RSA and ECC for encryption and decryption, but SHA 256 is used exclusively for hashing.

SHA-1 has been deprecated by NIST, and it won't be accepted after 2014 for any use for signatures (even though most uses of SHA-1 in TPM 1.2 don't fall prey to the types of attacks that are made possible by current cryptanalysis). The bit strength of SHA-1 is significantly weaker than that of the other algorithms, so there doesn't appear to be any good reason to use it other than backward compatibility.

TCG has announced the families of algorithms that can be supported by publishing a separate list of algorithm IDs that identify algorithms to be used with a TPM. This includes the hash algorithms to be used by the PCRs. This list may change with time.

Algorithm agility enables a number of nice features, including the following

- Using sets of algorithms compatible with legacy applications
- Using sets of algorithms compatible with the US Government's Suite B for Secret
- Using sets of algorithms compatible with the US Government's Suite B for Top Secret
- Using sets of algorithms compatible with other governments' requirements
- Upgrading from SHA-1 to SHA 256 (or other more secure algorithms)
- Changing the algorithms in a TPM without revisiting the specification

Enhanced Authorization (New in 2.0)

The TPM 1.2 specification accrued a number of new facilities over the years. This resulted in a very complicated specification with respect to means and management of authentication. The TPM was managed using either physical presence or owner authorization. Use of the EK was gated by owner authorization. Keys had two authorizations: one for use of the key and one to make duplicates of the key (called *migration* in the TPM 1.2 specification). Additionally, keys could be locked to localities and values stored in PCRs.

Similarly, the NVRAM in TPM 1.2 could be locked to PCRs and particular localities, and to two different authorizations—one for reading and one for writing. But the only way the two authorizations could differ was if one of them were the owner authorization.

Certified migratable keys had the same authorizations as other keys; but to complete the migration, a migration authority had to sign an authorization, and that authorization had to be checked by the TPM. This process also required owner authorization.

Making things even more complicated, the use of certain owner-authorized commands and keys could be delegated to a secondary password. However, the owner of the primary authorization knew those passwords, and delegation used precious NVRAM in the TPM. Even worse, the technique was difficult to understand and, as a result, was never employed to our knowledge.

The 2.0 specification has a completely different take, called *enhanced authorization (EA)*. It uses the following kinds of authorizations:

- *Password (in the clear)*: This was missing in TPM 1.2. In some environments, such as when BIOS has control of a TPM before the OS has launched, the added security obtained by using a hash message authentication code (HMAC) doesn't warrant the extra software cost and complexity of using an HMAC authorization to use the TPM's services.
- *HMAC key (as in 1.2)*: In some cases, particularly when the OS that is being used as an interface to talk with the TPM isn't trusted but the software talking to the TPM *is* trusted, the added cost and complexity of using an HMAC for authorization is warranted. An example is when a TPM is used on a remote system.
- *Signature (for example, via a smart card)*: When an IT employee needs to perform maintenance on a TPM, a smart card is a good way to prevent abuse of an IT organization's privileges. The smart card can be retrieved when an employee leaves a position, and it can't be exposed as easily as a password.
- *Signature with additional data*: The extra data could be, for example, a fingerprint identified via a particular fingerprint reader. This is a particularly useful new feature in EA. For example, a biometric reader can report that a particular person has matched their biometric, or a GPS can report that a machine is in a particular region. This eliminates the TPM having to match fingerprints or understand what GPS coordinates mean.
- *PCR values as a proxy for the state of the system, at least as it booted*: One use of this is to prevent the release of a full-disk encryption key if the system-management module software has been compromised.[3]

[3]Yuriy Bulygin, Andrew Furtak, and Oleksandr Bazhaniuk, "A Tale of One Software Bypass of Windows 8 Secure Boot" (presentation, Black Hat 2013), https://www.blackhat.com/us-13/briefings.html#Bulygin.

- *Locality as a proxy for where a particular command came from*: So far this has only been used to indicate whether a command originated from the CPU in response to a special request, as implemented by Intel TXT and AMD in AMD-v. Flicker,[4] a free software application from Carnegie Mellon University, used this approach to provide a small, secure OS that can be triggered when secure operations need to be performed.
- *Time*: Policies can limit the use of a key to certain times. This is like a bank's time lock, which allows the vault to be opened only during business hours.
- *Internal counter values*: An object can be used only when an internal counter is between certain values. This approach is useful to set up a key that can only be used a certain number of times.
- *Value in an NV index*: Use of a key is restricted to when certain bits are set to 1 or 0. This is useful for revoking access to a key.
- *NV index*: Authorization is based on whether the NV index has been written.
- *Physical presence*: This approach requires proof that the user is physically in possession of the platform.

This list isn't complete, but it gives examples of how the new policy authorization scheme can be used. Additionally, you can create more complicated policies by combining these forms of authorization with logical AND or OR operations such as these:

- Mary identifies herself with an HMAC key and a smart card associated with a public key.
- Joe identifies himself with a fingerprint authentication via a particular reader identified by the public key.
- This key can be used by Mary OR Joe.

Policies can be created that are either simple or complex, and all objects or entities of the TPM (including the TPM's hierarchies) can have policies associated with them. EA has enormously extended the possible uses of the TPM, particularly in managing authorizations; yet the net result has been to reduce the amount of code necessary to create a TPM, eliminate the NVRAM that was used for delegation, and eliminate all the previously existing special cases (thus lowering the learning curve for using a TPM).

Clever policy designs can allow virtually any restriction on key use that you can envision, although some (such as restricting use of a document to only one kind of document processor) would be exceptionally difficult, if possible at all.[5] The new EA allows a number of new scenarios, including the following:

[4]`http://sourceforge.net/p/flickertcb/wiki/Home/`.

[5]E.W. Felten, "Understanding Trusted Computing: Will Its Benefits Outweigh Its Drawbacks?" *IEEE Security & Privacy* 1, no. 3 (2003): 60–62.

- Multifactor authentication of resources
- Multiuser authentication of resources
- Resources used only *n* times
- Resources used only for certain periods of time
- Revocation of use of resources
- Restricting ways resources can be used by different people

Quick Key Loading (new in 2.0)

In the TPM 1.2 specification, when a key was initially loaded, it had to go through a time-consuming private-key decryption using the key's parent's private key. To avoid having to do this multiple times during a session, it was possible to cache loaded keys by encrypting them with a symmetric key that only the TPM knew. During that power cycle, the TPM could reload the key using a symmetric-key operation, which was faster even if the parent no longer resided in the TPM. Once the TPM was turned off, the symmetric key was erased: the next time the key was loaded, it again required a private key operation.

In 2.0, except for the case of a key being imported into a TPM's key structure from outside, keys stored by the TPM using external memory are encrypted using a symmetric-key operation. As a result, the keys are loaded quickly. There is little reason to cache keys out to disk (unless a parent key becomes unavailable), because loading them is usually as fast as recovering them from a cached file.

This quicker loading enables multiple users to use a TPM without noticing a long delay. This in turn makes it easier to design a system on which multiple applications appear to have unfettered access to a TPM.

Non-Brittle PCRs (New in 2.0)

Fragility of PCR values was one of the most annoying problems with the 1.0 family of TPMs. PCR values typically represent the state of the machine, with lower-numbered PCRs representing the process of booting of the system and higher-numbered ones representing events after the kernel has booted. Both keys and data can be locked to certain PCRs having particular values, an action called *sealing*. But if keys or data are locked to a PCR that represents the BIOS of a system, it's tricky to upgrade the BIOS. This is *PCR fragility*. Typically, before a BIOS upgrade was performed on TPM 1.2 systems, all secrets locked to PCR 0, for example (which represents the BIOS), had to be unsealed and then resealed after the upgrade was done. This is both a manageability nightmare and a nuisance to users.

In the TPM 2.0 specification, you can seal things to a PCR value approved by a particular signer instead of to a particular PCR value (although this can still be done if you wish). That is, you can have the TPM release a secret only if PCRs are in a state approved (via a digital signature) by a particular authority.

In typical usage, an IT organization may approve BIOS versions for PCs and then provide signatures of the PCRs that would result from approved BIOS versions being installed on PC clients. Values that formerly could be recovered in only one of those states become recoverable in any of them.

This is done via the `TPM2_PolicyAuthorize command`, which you can also use many other ways. It's a general-purpose way of making any policy flexible.

This new capability enables a number of different use cases, such as these:

- Locking resources to be used on machines that have any BIOS signed by the OEM
- Locking resources to be used on machines that have any kernels signed by an OEM
- Locking resources to be used on machines that have any set of values for PCRs that are approved by the IT organization

Flexible Management (New in 2.0)

In the 1.0 family of TPM specifications, only two authentications existed in a TPM at a time: the owner authorization and the storage root key (SRK) authorization. Because the SRK authorization was usually the well-known secret (20 bytes of 0s), the owner authorization was used for many purposes:

- To reset the dictionary-attack counter
- To reset the TPM back to factory settings
- To prevent the SRK from having its password changed by someone who knew the well-known secret
- To provide privacy to the end user by preventing creation of AIKs except by the owner of the TPM
- To avoid NVRAM wearout in the TPM by preventing creation and deletion of NVRAM indexes except by those who knew the owner authorization

The problem with giving the same authorization to so many different roles is that it becomes very difficult to manage those roles independently. You might want to delegate some of those roles to different people. For example, privacy controls like those used to restrict creation of AIKs are very different from controls used to reset the dictionary-attack counter or manage SRK authorization.

In the TPM 1.2 family, you could delegate the owner-authorization role to different entities using the `Delegate` commands in the TPM, but those commands were fairly complicated and used up valuable NVRAM space. We know of no applications that actually used them.

An additional problem with TPM 1.2–enabled systems was that the TPM couldn't be guaranteed to be enabled and active (meaning the TPM couldn't be used). So, many OEMs were unwilling to create software that relied on the TPM to do cryptographic things such as setting up VPNs during the boot process or verifying BIOS software before installation. This inhibited use of the TPM. In TPM 2.0, the OEM can rely on the platform hierarchy always being enabled.

In the TPM 2.0 family, the roles represented by the various uses of the TPM 1.2 owner authorization are separated in the specification itself. This is done by giving them different authorizations and policies, and also by having different hierarchies in the TPM. One is the dictionary-attack logic, which has its own password for resetting the dictionary-attack counter. The others are covered by several hierarchies in TPM 2.0:

- *Standard storage hierarchy*: Replicates the TPM 1.0 family SRK for the most part
- *Platform hierarchy*: Used by the BIOS and System Management Mode (SMM), *not* by the end user
- *Endorsement hierarchy or privacy hierarchy*: Prevents someone from using the TPM for attestation without the approval of the device's owner
- *Null hierarchy*: Uses the TPM as a cryptographic coprocessor

Each hierarchy (except the null hierarchy) has its own authorization password and authorization policy. The dictionary-attack logic also has an associated policy. All Entities on the TPM with an authorization value also have an associated authorization policy.

Identifying Resources by Name (New in 2.0)

In the TPM 1.2 specification, resources were identified by handle instead of by a cryptographically bound name. As a result, if two resources had the same authorization, and the low-level software could be tricked into changing the handle identifying the resource, it was possible to fool a user into authorizing a different action than they thought they were authorizing.[6]

In TPM 2.0, resources are identified by their name, which is cryptographically bound to them, thus eliminating this attack. Additionally, you can use a TPM key to sign the name, thus providing evidence that the name is correct. Because the name includes the key's policy, this signature can be used as evidence to prove what means are possible for authorizing use of a key. The chapter on enhanced authorization describes this in detail. If the key can be duplicated, this signature can also be used to provide a "birth certificate" for the key, proving which TPM was used to create the key.

[6]Sigrid Gürgens of Fraunhofer SIT found this attack.

Summary

This chapter has described at a high level the use cases enabled by TPM 1.2 and 2.0. The capabilities of TPM 1.2 are the basis for trusted computing—an anchor for secure generation, use, and storage of keys and for storage and attestation of a PC's health status. TPM 2.0 enhanced this functionality by adding sophisticated management and authorization capabilities, as well as algorithm agility that prevents new cryptographic attacks from breaking the specification.

The next chapter examines applications and SDKs that take advantage of those capabilities to solving existing problems. These include solutions for securing data at rest, like BitLocker and TrueCrypt; for PC health attestation and device identification, like Wave Systems, strongSwan and JW Secure; and a number of SDKs you can use to create applications with that functionality.