CHAPTER 14

Extended Authorization (EA) Policies

TPM 2.0 has unified the way that all entities controlled by the TPM may be authorized. Earlier chapters have discussed authorization data used for passwords and HMAC authorization. This chapter goes into detail about one of the most useful new forms of authorization in the TPM, starting with a description of why this feature was added to the TPM and then describing in broad brushstrokes the multifaceted approach that was taken.

This new approach for authorization has many capabilities. As a result, if a user wants to restrict an entity so it can be used only under specific circumstances, it's possible to do so. The sum total of restrictions on the use of an entity is called a *policy*. Extended authorization (EA) policies can become complex *very* quickly. Therefore this chapter's approach is incremental, first describing very simple policies and gradually adding complexity. This is done by examining how to build the following:

- Simple assertions
- Command-based assertions
- Multifactor authentication
- Multiuser/compound authorization
- Flexible policies that can be changed on the fly

Throughout this chapter, you see examples of practical policies like those used in most cases. It turns out that building policies is different than using them, so you learn how a user satisfies a policy; at that point it should become clear why policies are secure.

Finally, you consider some policies that can be used to solve certain special cases. This section may spur your creativity—you'll see that there are many more ways of using policies than you've thought of.

Let's begin by comparing EA policies to using passwords for authentication.

Policies and Passwords

All entities in the TPM can be authorized in two basic ways. The first is based on a password associated with the entity when it's created. The other is with a policy that is likewise associated with the entity when it's created. A policy is a means of authorizing a command that can consist of almost any approach to authorization that someone can think of. Some entities (hierarchies and dictionary attack reset handles) are created by the TPM and thus have default passwords and policies. The TPM-assigned *name* of these entities is fixed, not dependent on the policy that is used to authorize them. Such entities' policies can be changed.

All other entities—NVRAM indexes and keys—have their name calculated in part from the policy that is assigned when they're created. As a result, although their password can be changed, they have policies that are immutable. As you'll see, some policies can be made flexible so they can be easily managed in spite of this immutability.

Anything that can be done directly with a password can also be done with a policy, but the reverse isn't true. Some things (like duplicating a key) can only be authorized using a policy command. (However, making things more complicated, you can still use a password to authorize duplicating a key, by using a policy that describes a password authorization.)

A policy can be fine-tuned—everything is possible, from setting a policy to be the NULL policy that can never be satisfied, to having different authentication requirements for individual commands or for different users when applied to an entity. Thus EA is able to solve many issues that application developers need to deal with.

Why Extended Authorization?

EA in the TPM was created to solve the basic problem of manageability of TPM entity authorization. It makes it easier to learn how to use a TPM by having all TPM entities be authorized the same way, and it also allows a user to define authorization policies that can solve the following problems:

- Allow for multiple varieties of authentication (passwords, biometrics, and so on).
- Allow for multifactor authentication (requiring more than one type of authentication).
- Allow for creation of policies without the use of a TPM. Policies don't contain any secrets, so they can be created entirely in software. That doesn't mean secrets aren't needed to satisfy a policy.
- Allow attestation of the policy associated with an entity. It should be possible to prove what authorization is necessary in order to use an entity.
- Allow for multiple people or roles to satisfy a policy.
- Allow restriction of the capabilities of a particular role for an object to particular actions or users.

- Fix the PCR brittleness problem. In TPM 1.2, once an entity was locked to a set of PCRs that measured particular configurations, if the configurations ever had to be changed, the entity could no longer be used.
- Create a means to change how a policy behaves, providing flexibility.

Multiple Varieties of Authentication

Today, many different kinds of techniques and devices are used for authentication. Passwords are the oldest (and perhaps weakest) form of authentication. Biometrics such as fingerprints, iris scans, facial recognition, penned signatures, and even cardiac rhythm are used for authentication. Digital signatures and HMACs are forms of cryptographic authentication used in tokens or keys. Time clocks in banks use the time of day as a form of authentication and don't allow a vault to be opened except during business hours.

The TPM was designed so that objects can use almost any kind of authentication conceivable, although many forms require additional hardware. A policy can consist of a single type of authentication or multiple varieties.

Multifactor Authentication

Multifactor authentication is one of the most important forms of security and is popular today. It requires more than one means of authentication in order to provide authorization to execute a command. Those authentications may take many forms—smart cards, passwords, biometrics, and so on. The basic idea is that it's harder to defeat multiple authentication formats than it is to defeat a single one. Different forms of authentication have different strengths and weaknesses. For example, passwords can be easily supplied remotely—fingerprints less so, especially if the design is done correctly.

The TPM 2.0 design allows for many different forms of authentication and provides facilities to add even more using external hardware. Each mechanism that can be used for authentication is called an *assertion*. Assertions include the following:

- Passwords
- HMACs
- Smart cards providing digital signatures
- Physical presence
- State of the machine (Platform Configuration Register [PCR])
- State of the TPM (counters, time)
- State of external hardware (who has authenticated to a fingerprint reader, where a GPS is located, and so on)

A policy can require that any number of assertions be true in order to satisfy it. The innovation behind EA in the TPM is that it represents in a single hash value a complex policy consisting of many assertions.

How Extended Authorization Works

A policy is a hash that represents a set of authentications that together describe how to satisfy the policy. When an entity (for example, a key) is created, a policy may be associated with it. To use that entity, the user convinces the TPM that the policy has been satisfied.

This is done in three steps:

1. A policy session is created. When a policy session with the TPM is started, the TPM creates a session policy buffer for that session. (The size of the session policy buffer is the size of the hash algorithm chosen when the session was started, and it's initialized to all zeroes.)

2. The user provides one or more authentications to the TPM session, using `TPM2_PolicyXXX` commands. These change the value in that session policy buffer. They also may set flags in the session that represent checks that must be done when a command is executed.

3. When the entity is used in a command, the TPM compares the policy associated with the entity with the value in the session policy buffer. If they aren't the same, the command will not execute. (At this point, any session flags associated with policy authorizations are also checked. If they aren't also satisfied, this command isn't executed.)

Policies don't contain any secrets. As a result, all policies can be created purely in software outside a TPM. However, the TPM must be able to reproduce policies (in a session's policy digest) in order to use them. Because the TPM has this ability, it makes sense for the TPM to allow the user to use this facility to produce policies. This is done by using a *trial session*. A trial session can't be used to satisfy a policy, but it can be used to calculate one.

Policy sessions used to satisfy policies can be somewhat more complicated than the creation of a policy. Some policy commands are checked immediately and update a policy buffer stored in the session. Others set flags or variables in the session that must be checked when the session is used to authorize a command. Table 14-1 shows which policy commands require such checks.

Table 14-1. *Policy Commands that Set Flags*

Command	Sets Flag or Variable in Session Requiring the TPM to Check Something at Execution Time
`TPM_PolicyAuthorize`	No
`TPM_PolicyAuthValue`	Yes—sets a flag that requires an HMAC session to be used at command execution
`TPM_PolicyCommandCode`	Yes—checks that a particular command is being executed
`TPM_PolicyCounterTimer`	Yes—performs logical check against TPMS_TIME_INFO structured
`TPM_PolicyCpHash`	Yes—checks that the command and parameters have certain values
`TPM_PolicyLocality`	Yes—checks that the command is being executed from a particular locality
`TPM_PolicyNameHash`	Yes—identifies objects that will be checked to be sure they have specific values when the command is executed
`TPM_PolicyOR`	No
`TPM_PolicyTicket`	No
`TPM_PolicyPCR`	Yes—checks that PCRs have not changed when the command is executed
`TPM_PolicySigned`	No
`TPM_PolicySecret`	No
`TPM_PolicyNV`	No
`TPM_PolicyDuplicationSelect`	Yes—specifies where a key can be moved
`TPM_PolicyPassword`	Yes—sets a flag that requires a password at command execution

Creating Policies

Incredibly complicated policies are possible but are unlikely to be used in real life. In order to explain the creation of policies, this chapter introduces an artificial distinction between different kinds of policies, which are described in detail:

- *Simple assertion policy*: Uses a single authentication to create a policy. Examples include passwords, smart cards, biometrics, time of day, and so on.
- *Multi-assertion policy*: Combines several assertions, such as requiring both a biometric and a password; or a smart card and a PIN; or a password, a smart card, a biometric, and a GPS location. Such as policy is equivalent to using a logical AND between different assertions.
- *Compound policy*: Introduces a logical OR, such as "Bill can authorize with a smart card OR Sally can authorize with her password." Compound policies can be made from any other policies.
- *Flexible policy*: Uses a wild card or placeholder to be defined later. A policy can be created in which a specific term can be substituted with any other approved policy. It looks like a simple assertion, but any approved (simple or complicated) policy can be substituted for it.

As mentioned, a policy is a digest that represents the means of satisfying the policy. A policy starts out as a buffer that is the size of the hash algorithm associated with an entity, but set to all zeroes. As parts of the policy are satisfied, this buffer is extended with values representing what has happened. Extending a buffer is done by concatenating the current value with new data and hashing the resulting array with the designated hash algorithm. Let's demonstrate this with the simplest of all polices: those than require only one type of authorization to be satisfied.

Simple Assertion Policies

A simple Extended Authorization (EA) policy: the simple assertion policy, which consists of a single authentication, can be one of the following types:

- Password or HMAC (policies that require proof of knowledge of an object's password)
- Digital signatures (smart cards)
- Attestation of an external machine (a particular biometric reader attests that a particular user has matched, or a particular GPS attests that the machine is in a particular location)

- Physical presence (an indication such as a switch that proves a user is physically present at the TPM. While this is in the specification, it is not likely to be implemented, so we will ignore it in the following.)
- PCRs (state of the machine on which the TPM exists)
- Locality (the software layer that originated the TPM command)
- Internal state of the TPM (counter values, timer values, and so on)

Creating simple assertion policies can be done using the TPM itself, in three steps:

1. Create the trial session. This is as simple as executing the following command:

   `TPM2_StartAuthSession`

   It's passed a parameter `TPM_SE_TRIAL` to tell the TPM to start a trial session, and a hash algorithm to use for calculating the policy. This returns (among other things) the handle of a trial session. It's referred to as `myTrialSessionHandle`.

2. Execute TPM2 policy commands that describe the policy (described shortly).

3. Ask the TPM for the value of the policy created with the command by executing

   `TPM2_PolicyGetDigest`

   and passing it the handle of the trial session: `myTrialSessionHandle`).

4. End the session (or reset it if you want to use it again) by executing

   `TPM2_FlushContext`

   again passing it the name of the trial session: `myTrialSessionHandle`.

Because steps 1, 3, and 4 are common to all simple assertions, they aren't repeated in the following; we merely describe the second step for each command.

Passwords (Plaintext and HMAC) of the Object

Passwords are the most basic form of authentication used today, but they're far from the most secure. Nonetheless, because they're in use in so many devices, it was important that the TPM support them. (The TPM 1.2 did not support passwords in the clear—only proof of knowledge of the password using an HMAC. The TPM 2.0 supports both.) It's assumed that when a password is used, the device provides for a trusted path between the password entry and the TPM. If this doesn't exist, facilities are present in the TPM 2.0

architecture to allow for using a salted HMAC session to prove knowledge of a password without sending it in the clear, as seen in Chapter 13. When an object is loaded into a TPM, the TPM knows its associated password. Therefore, the policy doesn't need to include the password. Thus the same policy can be used with different entities that have different passwords.

Creating a simple assertion policy can be reduced to four steps:

1. Set the policy buffer to all zeroes, with the length equal to the size of the hash algorithm.

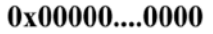

Figure 14-1. *Initializing the Policy*

2. Concatenate `TPM_CC_PolicyAuthValue` to this buffer.

Figure 14-2. *Concatenation of the buffer with the policy data per the Specification*

3. Substitute the value of `TPM_CC_PolicyAuthValue` from its value in part 2 of the specification.

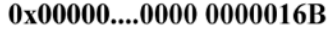

Figure 14-3. *Substituting the value of TPM_CC_PolicyAuthValue*

4. Calculate the hash of this concatenation, and put the result in the buffer. This end result *is* the policy for a simple assertion.

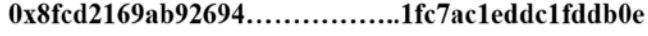

Figure 14-4. *Hashing the result provides a new value for the buffer*

When this policy command is executed, the policy buffer of the session is set to this final value. In addition, a flag is set in the TPM's session specifying that when a command with an object in it that requires authorization is used, a password session must be provided with that command and the password provided must match that of the object.

Similarly, when a policy is created to use the HMAC assertion (`TPM2_PolicyAuthValue`), two things happen

1. The policy is extended with the value `TPM_CC_PolicyAuthValue`.
2. A flag is set in the TPM's session indicating that when objects requiring authorization are used, a separate HMAC session is required. The TPM checks the password HMAC against the object's authorization data and allows access if they match (see Chapter 13.)

If you're using a trial session to create the policy, you accomplish this by executing the command `TPM2_PolicyAuthValue and` passing it the handle of the trial session.

This inherently means that when you're using passwords, either in plaintext or as an HMAC, either the plaintext password or the HMAC must be included to authorize a command with a policy session. The fact that `TPM_CC_PolicyAuthValue` appears twice in the previous explanation isn't a typo: the repetition means the choice of password or HMAC isn't decided when the policy is created, but rather when the non-policy command is executed. It's up to the user of the entity, not the creator of the entity, to decide how they will prove their knowledge of the password to the TPM.

Passwords aren't the most secure means of authentication. A much more secure approach is to use a digital signature, often implemented with a smart card such as a United States Department of Defense (DoD) Common Access Card (CAC card) or United States Federal Personal Identity Verification (PIV) card.

Passwords of a Different Object

A new (and very useful) assertion policy in TPM 2.0 is an assertion that the user knows the password of an entity *different from the one being used*. Although this might seem odd at first, it's particularly useful because of the difference in the behavior of NVRAM entities versus key objects. When the password of a key object is changed with `TPM2_ChangeAuth`, what is really happening is that a new copy of the key is being created that has a new password. There is no guarantee that the old copy is discarded. This is because key objects normally reside in files outside the TPM, and the TPM therefore can't guarantee that the old copy of the key file has been erased. However, NV entities reside entirely in the TPM: if their password is changed, it really is *changed*. The old copy can no longer be used.

This means if a key is created and a policy is created for it that requires the user to prove knowledge of an NV entity's password, it's possible to change the password necessary to use the key without worrying that the old password can still be used to authorize the key. In this case, changing the password of the NV entity effectively changes the password of the key. TPM 2.0 allows you to make authorization of a key dependent on knowing an NVRAM entity's password.

This further provides opportunities to manage the passwords of a large number of entities. Suppose you create a policy that points to a particular NV index's password, and then you associate that policy with a large number of keys. You can effectively change the password of all those keys by changing the password of the one NV index.

The `TPM2_PolicySecret` command requires you to pass in the name of the object whose password is required to satisfy the policy. It's perhaps not obvious, but when creating the policy for an object, you can't pass in the name of the object being created. This is because the name of the object depends on the policy, and if the policy depends on the name of the object, a vicious circle is created. This explains why the `TPM2_PolicyAuthValue` command is also needed. It provides a way of pointing to the authorization of the object being authorized.

To calculate the policy in a trial session, you execute the command `TPM2_PolicySecret` and pass it the handle of the trial session, as well as the handle of the object whose authorization will be used. Doing so extends the session policy buffer with `TPM_CC_PolicySecret || authObject→Name || policyRef`. The variable of note that is passed to the command is, of course, a handle for the object whose authorization will be used. As explained regarding names, although the handle of that object is passed to the TPM when executing `TPM_CC_PolicySecret`, the TPM internally uses the `Name` of the object in extending the session policy buffer. This prevents a change in the handle from causing a security exposure.

Technically, you need to include an authorization session for the handle of the object being authorized when executing this command. Although the specification indicates that it doesn't need to be satisfied in a trial session, most implementations require it. Therefore you must also include a correct password or HMAC session when executing this command. If you instead calculate the policy without using the TPM, this requirement isn't necessary.

Digital Signatures (such as Smart Cards)

It wasn't generally possible to authenticate use of a TPM 1.2 entity using a private key. In TPM 2.0, this has changed. It's now possible to require a digital signature as a form of access control. When a policy is formed using this assertion, the policy value is extended with three values: `TPM_CC_PolicySigned, SHA256`[1]`(publicKey)` and `a policyRef`. (A `policyRef` is used to identify precisely how the signed assertion will be used. Often it will be left as an Empty Buffer, but if a person is asked to authorize an action remotely, that person may want to precisely identify what action is being authorized. If the `policyRef` is part of the policy, the authorizing party will have to sign that value when authorizing the action.)

This can be done using a trial session by using the `TPM2_PolicySigned` command; but before this can be done, the TPM must know the public key used to verify the signature. This is done by loading that public key into the TPM first. The easy way to do this is to use a `TPM2_LoadExternal` command and load the public key into the `TPM_RH_NULL` hierarchy. You do so with the command `TPM2_LoadExternal`, passing in the public key structure.

[1]SHA256 is used throughout this book as the hash algorithm for everything except PCRs. However, technically you can use any hash algorithm that matches that chosen when the policy is created.

This returns a handle to the loaded public key, for now called `aPublicHandle`. Then you execute the command `TPM2_PolicySigned`, passing in the handle of the trial session and the handle of the loaded public key.

Satisfying this policy is trickier. Proving to the TPM that the user has the smart card with the private key that corresponds to this public key is a bit more involved. This is done by using the private key to sign a nonce produced by the TPM. You see this in detail at the end of this chapter.

Another assertion can be required: that the TPM resides in a machine that is healthy. This is done with PCRs.

PCRs: State of the Machine

Platform Configuration Registers (PCRs) in a TPM are typically extended by preboot or postboot software to reflect the basic software running on a system. In the 1.2 design, only a few things could use this authorization. Further, because using PCRs to restrict use of TPM 1.2 keys is a brittle operation, the restriction made this feature difficult to use.

In the 2.0 design it's possible to require that PCRs contain particular values for authorizing any command or entity. The policy merely has to specify which PCRs are being referenced and the hash of their values. Additionally, TPM 2.0 includes multiple ways of handling the brittleness. Again, all policies begin as a variable of size equal to the hash algorithm and initialized to zero. To use the PCR assertion, the policy is extended with `TPM_CC_PolicyPCR || PCRs selected || digest of the values to be in the PCRs selected`.

If a trial session is being used to calculate this policy, the user first selects the PCRs they wish to have defined values and puts them into a `TPML_PCR_SELECTION`. The user then calculates the hash of the concatenation of the defined values, calling the result `pcrDigest`. Then the user executes the command `TPM2_PolicyPCR`, passing in again the handle of the trial session and the PCRs selected and the `pcrDigest` just calculated.

When a user wishes to use an entity locked to PCRs, they execute the `TPM2_PolicyPCR` command, passing it the list of PCRs selected and the expected value of `pcrDigest`. Internally the TPM calculates the digest of the then-current values of those PCRs, checks it against the passed in value, and, if they match, extends the session's digest with `TPM_CC_PolicyPCR || PCRs selected || digest of the values currently in the PCRs selected`.

This might leave a security hole—what if the PCR values change after the assertion is made? The TPM protects against this by recording its PCR-generation counter in the TPM session state as `TPM_PolicyPCR` is executed. Each time any PCR is extended, the TPM generation counter is incremented. When the policy session is used to authorize a command, the current state of the generation counter is matched against the recorded value. If they don't match, it indicates that one or more PCRs have changed, and the session is unable to authorize anything.

As added flexibility, the platform-specific specification can indicate that certain PCRs will not increment the TPM generation counter. Changes to those PCRs will not invalidate the session.

Locality of Command

The 1.2 design had a characteristic called *locality* that was used to designate which software stack originated a command when it was sent to the TPM. The main usage in 1.2 was to provide proof that a command originated when the CPU was in a peculiar mode caused by entering either the Intel TXT or AMD-V command (in Intel or AMD processors, respectively). These commands are used for the Dynamic Root of Trust Measurement (DRTM) when the machine is put into a vanilla trusted state while in the midst of operations, so that the state of the machine's software can be reported in a trusted manner.

In 2.0, just as PCR assertions are extended for use whenever authorization can be used, locality is extended to a general-purpose assertion. When locality is used as an assertion in a policy, the session policy digest is extended with `TPM_CC_PolicyLocality || locality(ies)`.

When using the trial session to calculate the policy, you execute the command `TPM2_PolicyLocality`, passing in the handle of the trial session and the locality structure, `TPMA_LOCALITY`, found in part 2 of the specification.

When satisfying a locality for a session, the user uses `TPM2_PolicyLocality` to pass the localities to which the session is to be bound. Then two things happen:

1. The session digest is extended with `TPM_CC_PolicyLocality || locality(ies)`.
2. A session variable is set, recording the locality passed in.

When a command is then executed with that session, the locality from which the command is coming is compared to the locality variable set in the session. If they don't match, the command will not execute.

In the 1.2 specification, there were five localities—0, 1, 2, 3, and 4—which were represented by a bitmap in a single byte. This allowed you to select several localities at a time: for example, 0b00011101 represented the selection of localities 0, 2, 3, and 4. In the 2.0 specification, this result can be easily achieved using the `PolicyOr` command; but to reduce the cognitive load on people moving from 1.2 to 2.0, the localities 0–4 are represented the same way as before.

The problem with this solution is that it limits the number of localities available. It was possible to add three more localities, represented by bits 5, 6, and 7. However, the mobile and virtualization workgroups in TCG wanted more. This resulted in a bit of a hack in the specification. To extend the number of localities, the byte values above the fifth bit are used to represent single localities. This results in localities of the form 0, 1, 2, 3, 4, 32, 33, 34, ...255. That is, there is no way to represent localities 5–31. This is shown in Table 14-2. Note the change that happens when the value is 32.

Table 14-2. *Locality Representations and the Locality(ies) They Represent*

Value	Binary Representation	Locality(ies) Represented
0	0b00000000	None
1	0b00000001	Locality 0
2	0b00000010	Locality 1
3	0b00000011	Localities 0, 1
4	0b00000100	Locality 2
5	0b00000101	Localities 0, 2
6	0b00000110	Localities 1, 2
7	0b00000111	Localities 0, 1, 2
8	0b00001000	Locality 3
9–30	…	…
31	0b00011111	Localities 0, 1, 2, 3, 4
32	0b00100000	Locality 32
33	0b00100001	Locality 33
34	0b00100010	Locality 34
35–254	…	…
255	0b11111111	Locality 255

Localities can be used in a number of places. They can represent the origin of a command used to create an entity. They can also be used to lock functions so they can be used only if the command originates from a certain location. In 1.2, the locality was used to allow the CPU to control resetting and extending certain PCRs (for example, 17 and 18) to record putting the PC in a known state before doing a DRTM. Trusted Boot (tboot) is a program available on SourceForge[2] that shows how this is used; Flicker,[3] a program from CMU, used tboot to do run security-sensitive operations in a memory space separate from the OS.

Localities therefore tell the TPM where a command originated. The TPM inherently knows the values of its internal data, and localities can also be used for authorization restrictions.

[2] http://sourceforge.net/projects/tboot/.
[3] http://flickertcb.sourceforge.net.

Internal State of the TPM (Boot Counter and Timers)

TPM 1.2 had both an internal timer that measured the amount of time elapsed since the TPM was last powered on (and that could be correlated with an external time) and internal monotonic counters. Neither could be used as authentication elements. TPM 2.0 has a timer, a clock, and boot counters, which can be used in complicated formulas to provide for new assertions. A *boot counter* counts the number of times the machine has been booted. The timer is the amount of time since the TPM started up this time. The clock is similar to a timer, except that it (mostly) can only go forward in time, can be set equal to an external time, and stops whenever the TPM loses power.

These can be used to restrict usage of a TPM entity to only work when a boot counter remained unchanged, or when the clock is read between certain times. The entity's use can also be restricted to daylight hours. The latter is the most likely use case—restricting a computer to accessing files only during business hours helps protect data if a hacker gets access to the network at night.

The TPM can always check the values stored in its internal clock and boot counter, so they're referred to as *internal states*. Internal state assertions require that a policy session be created before the command is executed and that the assertion be satisfied before the command is executed. They need not be true when the command is actually executed.

This is done by extending a policy with `TPM_CC_PolicyCounterTimer || HASH(Time or Counter value || offset to either the internal clock or the boot counter || operation)`. The `operation` parameter indicates the comparison being performed. The table of operations is in part 2 of the specification: a set of two-byte values representing equality, non-equality, greater than, less than, and so on.

Using the trial session to create such a policy involves sending `TPM2_PolicyCounterTimer` with four parameters: the handle of the trial session; an indication as to whether the comparison is being done to the timer, the clock, or the boot counter; something to compare that value to; and the comparison being done.

Although these values are considered the TPM's internal state values, it's also true that the TPM can read values that are in any of its NV index locations. Those can also be used for policy commands.

Internal Value of an NV RAM Location

A new command for the TPM 2.0 specification allows the use of an entity based on the value stored in a particular NVRAM location. For example, if an NV index is associated with 32 bits of memory, you can gate access to a TPM entity based on whether one of those bits is a zero or a one. If each bit is assigned to a different user, a user's access to a particular entity can be revoked or enabled by simply changing a single bit in an NVRAM location. Of course, this means the person with authority to write to that NVRAM location has the ultimate authority for using the key.

This command is more powerful than that, because logical operations on the NVRAM location are allowed. So you could say that the entity could be used only if

```
6 <= NVRAM location <8 OR 9 < NVRAM location < 23
```

was a true statement.

NVRAM locations in a 2.0 TPM can be set to be counters. This means you can use them in clever manipulations in a policy that can make a counter useable only *n* time. An example of this is shown later in the chapter.

This works by extending the policy buffer with `TPM_CC_PolicyNV || calculated Value || name of NV location`. The calculated value is `HASH(value to compare to || offset into the NVRAM location || number that represents the operation)`, where the operation is one of the following:

- Equals.
- Not equal.
- Signed greater than.
- Unsigned greater than.
- Signed less than.
- Unsigned less than.
- Unsigned greater than or equal.
- Signed greater than or equal.
- Unsigned less than or equal.
- Signed greater than or equal.
- All bits match the challenge.
- If a bit is clear in the challenge, it's also clear in memory.

Using these functions, you can allow all values greater than 1 or less than 1,000. When you get to multifactor authentication, you can combine these to have a value that is between 1 and 1000, including or not including the endpoints.

You can use a trial session to create this policy by executing `TPM2_PolicyNV` with the same parameters used in the `TPM2_PolicyCounterTimer` command: the handle of the trial session, the index being compared (and the offset from the beginning of the index), the thing to compare against, and how it is to be compared.

If you consider an entity like a lock, the value of the NVRAM is like the tumblers. If their state is correct, the entity can be used. Locks open if their internal state is correct.

However, TPM 2.0 allows something more interesting: an entity can be used according to the state of a device *external* to the TPM.

State of the External Device (GPS, Fingerprint Reader, and So On)

Perhaps one of the most interesting new assertions in the TPM design is the ability to use an assertion that is dependent on the state of an external device. The device is represented by a public/private key pair. The state of the device may be anything the device can use its private key to sign (together with a nonce from the TPM). If the device is a biometric, it may be as simple as "Bob just authenticated himself to me." If it's a GPS unit, it may be "My current position is Baltimore." If it's a time service, it may be

"The current time is business hours." The assertion identifies both the public key that represents the external device and the value expected. The TPM does nothing more than compare the signature and the identified information with what it's expecting. It doesn't perform calculations on the resulting information, so the device making the representation needs to decide if its input matches the thing it's signing.

This provides flexibility for a biometric: if Bob has registered several fingerprints with the matcher, the TPM doesn't need to know which one was signed with—just that the match corresponds to "Bob." A GPS coordinate need not be exact—just in a specified area. The assertion need not specify an exact time, but rather an identifier for the range of times that are acceptable. However, the flexibility isn't entirely general. This doesn't say "Some fingerprint reader attests that Bob has authenticated to the device"; it says "This particular fingerprint reader (as demonstrated by a signature) attests that Bob has authenticated to the device." This allows the creator of the policy to determine which biometric (or other devices) it trusts to not be easily spoofed.

Once this policy is satisfied, there are no further checks, so it's possible for an assertion to no longer be satisfied when the TPM actually executes the command.

Creating the policy is done by starting with a variable of size equal to the hash algorithm and initialized to zero. This is then extended with `TPM_CC_PolicySigned || SHA256(publicKey) || stateOfRemoteDevice`, where `stateOfRemoteDevice` consists of two parts: the size of the description followed by the description.

If you're using a trial session to create this policy, you execute the command `TPM2_PolicySigned`. Again, you must pass the handle of the trial session, the handle of the public key that corresponds to the private key of the device, and the state of the remote device that it will sign when the policy is satisfied. For example, if the remote device is a fingerprint reader, the device may sign "Sally correctly authenticated."

Sometimes the object's creator doesn't really know under what circumstances they want a key to be used. Perhaps the key will be used in case of an emergency, and the creator doesn't know who will use the key or how. This is a use case for a wild card policy.

Flexible (Wild Card) Policy

One major problem with the TPM 1.2 design was the brittleness of PCRs. When an entity was locked to a PCR, it was not possible to change the required values of the PCR after it was so locked. PCR0 represents the BIOS firmware, which is security critical. If PCR0 changed, it could indicate a security breach. As a result, applications like Microsoft BitLocker use it for security. However, BIOS firmware may need to be upgraded. When it's upgraded, the value of PCR0 will change, which makes anything locked to that PCR no longer useable.

Programs got around this limitation by decrypting keys, upgrading the BIOS, and then re-encrypting the keys to the new value of PCR0. However, this process is messy and leaves keys exposed for a short period of time while the upgrade is taking place. As a result, it was important that EA be able to allow for changing of the values to which a PCR was locked without decrypting the locked data. But it needed to also be obvious to anyone who wished to check the policy under what circumstances the policy could be changed. A number of possibilities were considered, including having yet another authorization whose only use was to change the policy.

The solution chosen was clever and is given using the command `TPM2_PolicyAuthorize`, which I call a *wild card policy*. A wild card policy is owned by a private key whose public key is associated with the wild card. In poker, a wild card can substitute for any card the holder of the wild card wishes. A wild card policy can substitute for any policy the owner of wild card wishes. Any policy approved by the owner of a wild card can be used to satisfy the wild card policy. Policies also can be restricted with a `wildCardName` that can be given to the wild card when it's created. This allows the owner of the wild card to specify that only wild cards with a particular name can substitute for a particular policy. A wild card associated with an OEM's BIOS signing key could theoretically be used to approve any BIOS signed by the OEM.

The wild card policy is created in a way similar to the command used for the state of an external device, by extending a policy session with `TPM_CC_PolicyAuthorize || keySign→nameAlg || keyName || wildCardName`. Just as with the `PolicySigned` assertion, if you're using a trial session to create a wild card policy, you first have to load the public key into the TPM (using the `TPM2_LoadExternal` command) and then execute the `PolicyAuthorize` command.

`TPM2_LoadExternal` returns the handle of the loaded public key, here called `aPublicHandle`. Then you can execute `TPM2_PolicyAuthorize`, passing it the handle of the trial session, `wildCardName`, and `aPublicHandle`.

`TPM2_PolicyAuthorize` is one of the most useful policies in the TPM, because it's the only way to effectively change a policy after an object has been created. This means if objects have been locked to one set of PCR values (corresponding to a particular configuration), and the configuration has to change, the objects' policy can be effectively changed to match the new set of configuration values. You see a number of other uses as well in the "Examples" section.

Command-Based Assertions

Although not strictly an assertion, it's possible to restrict a policy so that it can only be used for a particular command. For example, you can restrict a key so that it can be used for signing but not to certify other keys. If this is done, then the policy can only be used to do that one particular command. Generally this isn't done as a single assertion, but it could be. By declaring in a key's policy that it can only be used for signing, the key is prevented either from certifying another key or from itself being certified. This is because when a key is certifying or being certified, it needs to provide an authorization that can't be provided.

To create such a policy assertion, you create a policy variable of size equal to the hash algorithm and initialize it to zero. It's then extended with the value `TPM_CC_PolicyCommandCode || the command code to which the policy is to be restricted`.[4] If you're using a trial session to create this policy, you execute `TPM2_PolicyCommandCode`, passing it the handle of the trial session and the command code.

Usually, if you're restricting a TPM entity like a key to only be used in a single command, you also want to authenticate use of that key for that command. This requires that more than one restriction be placed on the key, which is the subject of multifactor authentication.

[4]The `TPM_CC` listing table is found in part 2 of the specification.

Multifactor Authentication

The TPM knows how to authenticate using assertions. It also can be told to require more than one of them. For example, it may be asked to specify that both a fingerprint and a smart card be used to provide authentication to log in to a PC.

Policies, as you'll see, build together in a way similar to the way PCRs are extended. They start with an initial value of all zeroes (the number of zeroes depends on the size of the hash algorithm used to create the policy). When a policy command is invoked, the current policy value is extended by appending a new parameter to the old value, hashing the result, and then replacing the old value with the result of this calculation. This calculation is called *extending* in PCRs. A logical AND in a policy is accomplished by extending the new assertion into the policy. Just like a PCR, the policy is initialized to all zeroes before the first assertion, but later assertions build on the value created by the previous assertion.

This means if you're using a trial session to build this kind of policy, you start and end exactly the same way—you just add more commands in the middle to correspond to the various ANDed assertions.

Example 1: Smart card and Password

If you wish to require that both a smart card that signs with a key, whose public part is `S`, and a password be used in a policy, you create a policy by first extending

```
TPM_CC_PolicySigned || SHA256(publicKey) || 0x0000 = 0x00000060 || SHA256(S)
||0x0000[5]
```

into a buffer of all zeroes. You then extend a requirement for proving knowledge of a password by extending

```
TPM_CC_PolicyAuthValue=0x00000016B
```

into the result.

If you wish to also require that the command be executed in locality 4, you extend

```
TPM_CC_PolicyLocality || locality4
```

Extending a new requirement is equivalent to a logical AND.

Using a trial session, you first load the public key into the TPM and then execute the three commands used in each of the simple assertion policies: `TPM2_PolicySigned`, `TPM2_PolicyAuthValue`, and `TPM2_PolicyLocality`.

[5] In this case you don't assign a `PolicyReference` to this signature, so the last appended value is 0x0000, which is 2 bytes of 0, which means the value of the `PolicyReference` of the signature is `EmptyBuffer`.

Example 2: A Policy for a Key Used Only for Signing with a Password

In this example, Bob creates a key that requires a key *only* be used for signing, and only if a password is presented for the key. Start with a policy of all zeroes, and first extend it with

```
TPM_CC_PolicyCommandCode || TPM_CC_Sign = 0x0000016C || 0x0000015D
```

Then extend it again with

```
TPM_CC_PolicyAuthValue =0x000016B
```

Example 3: A PC state, a Password, and a Fingerprint

In this example, Bob creates a key which requires that PCR1 be equal to an `approvedPCRdigest`, a password, and a fingerprint. When crafting a policy that involves a PCR digest, it's generally good practice to start with that term first. This is because if it fails, there is no need to bother the user with a password and a fingerprint.

You use the TPM to create this policy value as follows:

1. Start a trial session.
2. Use `TPM2_PolicyPCR` to lock the policy to `approvedPCRdigest`.
3. Use `TPM2_PolicyAuthValue` (to require a password at execution).
4. Load the `publicKey` of the fingerprint reader.
5. Use `TPM2_PolicySigned` pointing to the public key and `stateOfRemoteDevice` (which is "Bob's finger").
6. Get the value of the policy from the TPM.
7. End the session.

Example 4: A Policy Good for One Boot Cycle

In this example, the IT administrator gives permission (for example, to a technician) for a previously created key to be used only during this boot cycle. First the administrator creates a key that has a policy controlled by a wild card. When the administrator wants to allow the technician to use the key, the admin reads the current boot counter value from the PC using `TPM2_GetCapability` and authorizes a policy for the key that states that the value of the boot counter must be its current value. The admin does this using their private key to sign this new policy, called `newPolicy`. If the key is in his own TPM, he can use the command `TPM2_Sign` to sign it. The admin sends this policy and signature to the technician.

The technician loads the public key into the TPM, using `TPM2_LoadExternal`, and then uses the `TPM2_VerifySignature` command to verify the signature of the new policy. This command returns a ticket to the technician.

The technician uses the key by starting a policy session and then executing `TPM2_PolicyCounterTimer` with an offset pointing to the boot counter. This satisfies the `newPolicy`. The technician then executes `TPM2_PolicyAuthorize`, feeding it the `newPolicy` and the ticket, and points to the admin's public key. The TPM verifies that the ticket is valid for the `newPolicy`, using the admin's public key, and then substitutes the current policy buffer with the `wildCardPolicy`. At this point, the technician can use the key during this boot cycle.

When the PC is rebooted, the boot counter is incremented. If the technician tries to use the policy again, they can never satisfy `newPolicy`, so they can't use the key.

Example 5: A Policy for Flexible PCRs

In this example, an IT administrator wants to lock a full-disk-encrypting software key to a set of PCRs that represent (among other things) the BIOS firmware. But the admin realizes that the BIOS might need to be updated and so uses `TPM2_PolicyAuthorize` to provide flexibility as to what PCR values are used to release the hard-disk encryption keys.

The admin's key is created with only `TPM2_PolicyAuthorize`, but the admin authorizes a new policy that requires the PCRs to be equal to the initial PCR values. The admin then uses `TPM2_VerifySignature` to create a ticket that can be used to validate use of that new policy.

When the disk-encryption key is to be decrypted, the machine needs to do the following:

1. Start a new policy session.
2. Use `TPM2_PolicyPCR` to replicate the new policy in the TPM.
3. Use `TPM2_PolicyAuthorize` (with the public administrator key, the new policy, and the policy ticket) to cause the TPM to change the internal policy buffer of its session to the original `PolicyAuthorize` policy.
4. Use the satisfied policy session to release the disk-encryption key.

If the admin ever needs to change the PCR values that are validated, the admin can send the user a newly signed policy corresponding to the new PCR values, and the user can use that to create a new ticket to use after the PCRs have changed.

Example 6: A Policy for Group Admission

In this example, a group of people are given access to use a department key. But as people come and go from the department, some people's access is removed and other people's access is granted. Each member of the department has access to a private key that represents them. You can do this with a clever use of `TPM2_PolicyNV`, `TPM2_PolicyAuthorize`, and `TPM2_PolicySigned`.

First you create a NV index that has 64 bits (assuming there will never be more than 64 people in your department). Write authority is given only to the IT administrator, using the admin's private key. The admin writes it with all zeroes, noting the value of the index name. The admin then creates the department key with only a `PolicyAuthorize` policy, with the public key corresponding to the IT administrator of the department.

The IT administrator assigns each member of the group a bit in the NV space. To give a user the right to use the key, the admin creates and approves a policy that requires the corresponding bit of the NVRAM index to be a 1 (using `PolicyNV`) and that the appropriate user use their private key for authentication, using `PolicySigned`. When the admin wants to remove a user's ability to use the key, the admin changes the bit in the `NVIndex` that corresponds to that user to a 0. The admin then signs each of these new policies and gives them to the appropriate user.

When a user wants to use the key, they do the following:

1. Start a policy session.
2. Executed a `PolicyNV` command to verify the user is still in the department.
3. Execute a `PolicySigned` command to prove the user is the corresponding person.
4. Execute a `PolicyAuthorize` command to change the TPM's internal policy buffer to the `PolicyAuthorize` policy.
5. Use the key.

Example 7: A Policy for NV RAM between 1 and 100

As noted earlier, this is as simple as executing two commands: one to say the NV RAM value is greater than 1 and another to say it's less than 100. This only allows values 2, 3, 4, ...99.

Compound Policies: Using Logical OR in a Policy

The `TPM2_PolicyOR` command completes the logical constructions that can be done with policies and makes it possible to create useful policies that will do anything logically feasible. It lets you join more than one policy in multiple branches, any of which can be taken in satisfying a compound policy, as shown in Figure 14-5.

Although `TPM2_PolicyOR` commands can be used in more complicated settings, it's easiest to create individual policies for specific means of authorizing use of an entity and then use `TPM2_PolicyOR` to create a compound policy. Usually this is done by creating simple policies by ANDing assertions together to represent either a person or a role, and then ORing the simple policies together.

Suppose the following things happen:

1. Dave authorizes himself using a policy created by a fingerprint together with a password when at one machine.
2. Dave authorizes himself using a password and smart card.
3. Sally uses her smart card and an iris scanner to authorize herself.
4. The IT administrator can only use his authorization to duplicate a key and must use a smart card when the system is in a state defined by PCR0-5 having specific values.

This can be represented pictorially using circuit diagrams as follows.

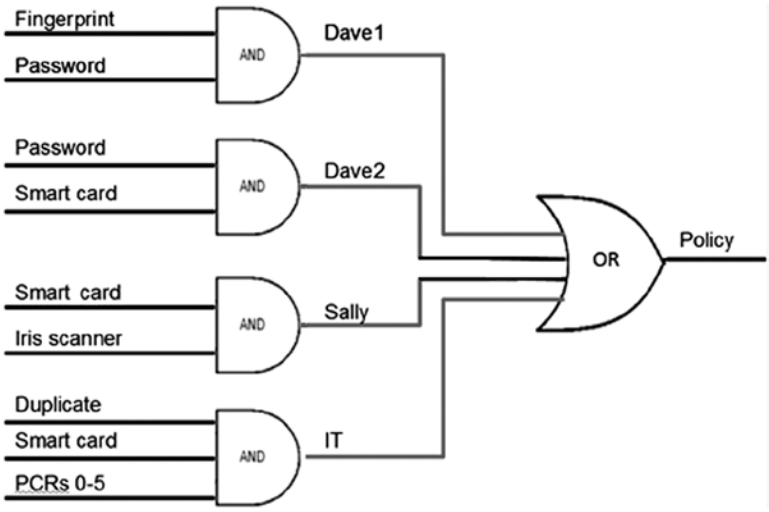

Figure 14-5. *A Compound Policy as a Circuit Diagram*

The easy way to create this compound policy is to start by creating four individual branch policies corresponding in the picture to Dave1, Dave2, Sally, and IT.

The first policy (Dave1) defines that Dave must authenticate himself with an external device (a fingerprint reader) and have it testify that Dave has authenticated himself. Dave must then present a password to the TPM. As you have seen, this is as simple as doing the following:

1. Start a trial session.
2. Use `TPM2_PolicySigned` (with the fingerprint reader's public key and appropriate `policyRef`).
3. Use `TPM2_PolicyAuthValue`.
4. Get the value of the policy from the TPM. Call this `policyDave1`.
5. End the session.

The second policy (Dave2) has Dave present a password to the TPM and then use his smart card to sign a nonce from the TPM to prove his is the authorized owner of the smart card:

1. Start a trial session.
2. Use `TPM2_PolicyAuthValue`.
3. Use `TPM2_PolicySigned` (with the smart card's public key).
4. Get the value of the policy from the TPM. Call this `policyDave2`.
5. End the session.

The third policy states that Sally must first use her smart card to sign a nonce from the TPM to prove she is the authorized owner of her smart card and then authorize herself to an external device, an iris scanner, and have the external device testify to the TPM that Sally has authenticated herself:

1. Start a trial session.
2. Use `TPM2_PolicySigned` (with the smart card's public key).
3. Use `TPM2_PolicySigned` (with the iris scanner's public key and appropriate `policyRef`).
4. Get the value of the policy from the TPM. Call this `policySally`.
5. End the session.

Finally, the IT administrator's policy requires the administrator to use his smart card to sign a nonce produced by the TPM and then also check that PCRs 0–5 are in the expected state. Furthermore, the IT administrator can only use this authorization to duplicate the key:

1. Start a trial session.
2. Use `TPM2_PolicySigned` (with the smart card's public key).
3. Use `TPM2_PolicyPCR` (with PCRs selected and their required digest).
4. Use TPM2_PolicyCommandCode with `TPM_CC_Duplicate`.
5. Get the value of the policy from the TPM. Call this `policyIT`.
6. End the session.

Making a Compound Policy

Each of these policies, by itself, could be assigned to a TPM entity such as a key. However, you wish to allow any of the policies to be used to authenticate access to a key, and you do this using the `TPM2_PolicyOR` command:

1. Start a trial session.
2. Use `TPM2_PolicyOR`, giving it the list of policies to be allowed: `policyDave1`, `policyDave2`, `policySally`, and `policyIT`.
3. Get the value of the policy from the TPM. Call this `policyOR`.
4. End the session.

Policies created this way on one TPM will work fine on any TPM. One restriction on `PolicyOr` is that it can only be used to OR together up to eight policies. However, just as with electronic circuit design, `PolicyOR`s can be compounded together to create the equivalent of an unlimited number of ORs. For example, if X is the result of 8 policies ORed together with `TPM2_PolicyOR`, and Y is the result of a different 8 policies ORed together with `PolicyOR`, you can apply `TPM2_PolicyOR` to X and Y to create the equivalent of a `PolicyOr` of 16 different policies.

Example: A Policy for Work or Home Computers

John has a home PC with a fingerprint reader and a work PC with a smart-card reader. He wants to authorize reading his cloud-based encrypted data from either computer. He does this by locking a key to a policy that requires a fingerprint reader from his home computer and his smart card (using his work PC's smart-card reader) for work.

He first creates a policy for his home computer. He gets the public key of the fingerprint reader and sets it up to sign "John's fingerprint" when he swipes his finger on that reader:

1. Start a trial session.
2. Use `TPM2_LoadExternal` to load the fingerprint reader's public key into the home computer's TPM.
3. Use `TPM2_PolicySigned` (with the fingerprint reader's public key and appropriate `policyRef`).
4. Get the value of the policy from the TPM. Call this `HomeFingerprintPolicy`.
5. End the session.

John now goes to his work computer:

1. Start a trial session.
2. Use `TPM2_LoadExternal` to load the smart card's public key into the work computer's TPM.
3. Use `TPM2_PolicySigned` (with the smart card's public key and NULL `policyRef`).
4. Get the value of the policy from the TPM. Call this policy `WorkSmart cardPolicy`.
5. End the session.

Now John can create the combined policy, which can be satisfied with both computers:

1. Start a trial session.
2. Use `TPM2_PolicyOr` with both `HomeFingerprintPolicy` and `WorkSmart cardPolicy` listed.
3. Get the value of the policy from the TPM. Call this policy `WorkOrHomePolicy`.
4. End the session.

This is the policy John uses when creating a key that he will use to identify himself to the cloud. He duplicates this key to his other computer, and then he can securely use this key on either computer.

Considerations in Creating Policies

In most cases, policies should be considered to represent roles when using TPM entities—and usually there are only a few possible roles.

End User Role

This represents the authentication that is satisfied for a user to use an entity. *Using an entity* means doing something like one of the following:

- Signing with a key
- Reading a NV location
- Writing an NV location
- Quoting with a key
- Creating keys

Administrator Role

An administrator of an entity may do different things for different entities. For NVRAM, they may be given the responsibility of managing the limited resource of available NVRAM. This would include the following:

- For NV:
  - Creating and destroying NV indexes
- For keys:
  - Authorizing duplication
  - Changing authorization with `PolicyAuthorize`

Understudy Role

In the event that the user of a key leaves the company or is unable to use a key necessary to obtain some enterprise data, it's important that another person (for example, the user's manager) be able to use the key. This is an *understudy role*.

Office Role

An office role consists of a combination (`PolicyOr`) of an enterprise administrator role and the user's role.

Home Role

A home role consists of a combination of a user acting as an administrator and acting as an end user. It may also include using different roles for using an entity on different machines, because different forms of authentication may be available on different machines. (For example, one machine may have a biometric reader and another may not.)

Once the roles are defined, policies can be created for them. Once the policies are created, they can be reused whenever entities are created, obviating the need to re-create the policies each time.

Using a Policy to Authorize a Command

You've seen how to satisfy a number of simpler policies. In order to satisfy any policy so that an object that requires the policy can be used, the steps are always the same:

1. Start a policy session.
2. Satisfy the policy for that session (this can require multiple steps).
3. Execute the command.
4. End the session.

This is very similar to the way policies are created, but satisfying a policy often requires additional steps. In a high-level API, most of the grunt work of satisfying a policy is done for you; but if you're talking directly to the TPM, some details are required to achieve this.

Starting the Policy

Starting the `PolicySession` is easy, as shown in Chapter 13. It's done with the command `TPM2_StartAuthSession`. This command returns a bunch of stuff, including a session handle, here called `myPolicySessionHandle`; and a nonce, created by the TPM, here called `nonceTPM`. You need both of these variables to satisfy the policy.

Satisfying a Policy

The considerations for satisfying the different kinds of policies—simple assertions, multifactor assertions, compound assertions, and flexible assertions—are slightly different, so let's consider them separately. It's important to remember that the order in which a policy is satisfied is important. A policy constructed with a `TPM2_PolicyPCR` followed by `TPM2_PolicyPassword` is different from a policy constructed with `TPM2_PolicyPassword` followed by `TPM2_PolicyPCR`. In general, policy commands aren't commutative.

Simple Assertions and Multifactor Assertions

Most simple assertions are easy to apply to a policy. Password, PCR, locality, TPM internal state, internal state of an NV RAM location, and command-based assertions are asserted in the same way as when the policy was created, except instead of using a trial policy, you use the policy handle `myPolicySessionHandle`. Other commands that require signature verification (the `TPM2_PolicySigned` command with or without a `policyRef`) require more work.

For example, if you're asserting that a password must be used to satisfy the policy, you execute the command `TPM2_PolicyPassword`. The password isn't actually passed at this time. This is just telling the session that when the command is finally executed with the object, the user must prove at that time that they know the password by passing it in either as a plaintext password or as an HMAC in the session.

To satisfy `TPM2_PolicySigned`, a signature is needed, and the signature is over a hash that is formed in part from the `nonceTPM` returned by the last use of the session. Additionally, the TPM must have the public key loaded so that it can verify the signature.

Loading the public key is done exactly the same way you did it to create the session, using the `TPM2_LoadExternal` command. This returns a handle to the loaded public key, here called `aPublicHandle`. You use this when calling `the PolicySigned` command, but first you have to pass in a signature. To do this, you first need to form a hash and sign it. The hash is formed by

```
aHash = HASH(nonceTPM || expiration =0 || cpHashA = NULL || policyRef = 0x0000)
```

where nonceTPM was returned by the TPM when the session was created, expiration is all zeroes (no expiration), cpHashA = Empty Auth, and policyRef is emptyBuffer. (If you're using this for verification of a biometric reader, then policyRef is equal to the name of the person whose biometric was verified). The private key is used to sign this hash; and when signed, the result is called mySignature.

Next you execute the TPM2_PolicySigned command, passing in the handle of the session, APublicHandle, and mySignature. At this point, the TPM checks the signature internally using the public key, and if it's verified, extends its internal session policy buffer as desired. Now any command with an object whose policy that matches that policy buffer can be executed.

If the Policy Is Compound

If a policy is compound—that is, it's a logical OR of several branches—the user knows which branch they're going to try to satisfy. Once the user picks the branch, they satisfy that branch and then execute a TPM2_PolicyOR command with the TPM, which transforms the satisfied branch into the final policy, ready for execution. See Figure 14-6.

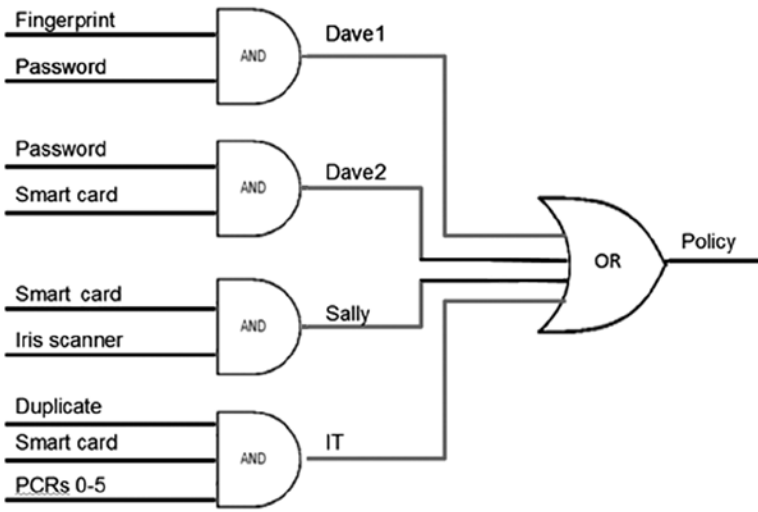

Figure 14-6. *An example of using an OR policy*

This figure shows that there are four different ways to satisfy this policy. You can satisfy it with the first branch, Dave1, by using a fingerprint reader and a password:

1. Start a policy session.
2. Satisfy the Dave1 branch of the policy:
   a. Satisfy the fingerprint assertion using `TPM2_PolicySigned`.
   b. Satisfy the password assertion using `TPM2_PolicyPassword`.
3. This sets a flag in the session, telling it that a password must be sent in when the final command is executed.
4. Transform the TPM's session policy buffer to the final session value using `TPM2_PolicyOR`.
5. Execute the command, including both the policy session and another session that satisfies the flag, by passing in the password (which can be done using the password [PWAP] permanent session).

   Note: As a side note, the policy session can be told to automatically close after this command is completed. Failing that, you can close the session manually.

In order to satisfy the first assertion in the policy, you have to get the fingerprint reader to attest to the TPM that Dave's fingerprint has been matched by the reader with the public key `aPub`. To do this, you need to pass a message to sign in to the fingerprint reader, which is calculated in part from `nonceTPM`, which the TPM returned when you created the policy. This value is sent to the fingerprint reader. Then Dave swipes his finger along the fingerprint reader, and when the fingerprint reader matches his fingerprint, it signs

```
aHash = SHA256(nonceTPM || expiration=0 || cpHashA=NULL || state Of
Remote Device)
```

using its private key `aprivate`. Note here the PolicyRef is the state of the remote device. In particular, the fingerprint reader needs to sign the fact that Dave has just swiped one of his fingerprints on the device and it has matched the template the device stored. The result is called `fingerprint_Signature`.

Next you have to load the fingerprint reader's public key into the TPM. Recall that this public key's handle is `aPub`.

Finally, the TPM is sent proof that the fingerprint reader successfully identified Dave using the command `TPM2_PolicySigned`, passing in `aPub` and `fingerprint_Signature`.

Next you execute the `PolicyAuthValue` command, which promises that when you eventually ask the TPM to perform a command with an object, that user will present evidence that they know the password associated with the object. This is done by executing `TPM2_PolicyAuthValue`.

Now that you've satisfied one of the branches of the policy, you can execute `TPM2_PolicyOR` to change the internal buffer of the session to equal the compound policy by passing it a list of the ORed policies.

If the Policy Is Flexible (Uses a Wild Card)

Satisfying a wild card policy is more complicated than creating one. For one thing, when the wild card policy is created, only the public key of the party who can authorize the eventually satisfied policy is identified. When one is used, an authorized policy must have been created, and a ticket proving that it's authorized must be produced. Then a user satisfies the approved policy and runs `TPM2_PolicyAuthorize`. The TPM checks that the policy buffer matches the `approvedPolicy` and that the `approvedPolicy` is indeed approved (by using the ticket), and if it is, changes the policy buffer to the flexible policy.

Preparing a policy to be used is then a two-step process. First, the authorizing party has to approve a policy by using their private key to sign `Hash(approved Policy || wildCardName=policyRef)`. This is then sent to the user.

The user loads the public key of the authorizing party in their TPM and uses `TPM2_VerifySignature` against this signature, pointing to the handle of the public key. Upon verification, the TPM produces a ticket for this policy.

When the user wants to use this new approved policy, the user first satisfies the approved policy the way they ordinarily would and then gets the TPM to switch the approved policy to the flexible policy by calling `TPM2_PolicyAuthorize`, giving it as parameters the name of the session that has satisfied the approved policy, the approved policy, `wildCardName`, `keyName`, and the ticket. The TPM verifies that the ticket is correct and matches the approved policy in the session policy buffer. If so, it changes the session policy buffer to be the value of the flexible policy.

Thus creating a flexible policy is really a two part process.

Recapitulating: First the policy itself is created:

- Start a Trial Session
- Load the administrator's public key
- Use TPM2_PolicyAuthorize pointing to the administrator's public key
- Get the Value of the policy from the TPM. We call this policy workSmartcardPolicy
- End the session

Then an authorized policy is created using the administrator's private key

- Create a policy
- Load the administrator's private key
- Use the administrator's private key to sign the policy
- Use the TPM on which the approved policy is to be used to verify the signature (this produces a ticket)

Satisfying the Approved Policy

Satisfying the approved policy is done just as though it were the only policy you had to worry about. It doesn't matter if the approved policy is simple, compound, or flexible. After it's satisfied, it's then transformed.

Transforming the Approved Policy in the Flexible Policy

Now that the TPM's buffer is equal to the approved policy, you can transform it into the flexible policy by executing `TPM2_PolicyAuthorize`, passing the current value of the session policy buffer, `PolicyTicket`, and `AdministratorPublicKeyHandle`. The TPM checks that the policy buffer matches the approved policy and that the approved policy is indeed approved (by using the ticket) and, if it is, changes the policy buffer to the flexible policy. At this point, commands can be executed on an object that requires this particular flexible policy.

Although flexible policies were introduced to the TPM in order to provide a solution to the brittleness of PCRs, they can be used to solve many more conundrums than that. They allow an administrator to decide after an object is created how the policy for that object can be satisfied. Because the name of an object (or NV index) is calculated from its policy, it isn't possible to change the policy of an NV index or a key. However, using a flexible policy, you can change the way a policy is satisfied after the fact.

Suppose a key is given a flexible policy when it's created, and later the administrator of the flexible policy wants to make it be the policy in Figure 14-6. The admin can accomplish this by signing the policy represented by Figure 14-6 and sending it to the user. Someone must do the preparatory step of creating a ticket by running `TPM2_VerifySignature`, but after that the user only has to satisfy the policy given by Figure 14-6 and then run `PolicyAuthorize` to prove that the policy has been approved.

Certified Policies

One last thing you can do with policies is prove that a policy is bound to a particular entity. When ink is used to sign a contract, the signature that is formed is irrevocably tied to the person signing it via a biometric that represents the way the person's muscles and nerves are formed. That is what produces the characteristic swirls of a signature. Electronic signatures have never been tied to a person in the same way. Typically, electronic signatures have been tied to a password (something a person knows) or a smart card (something a person has), or sometimes (usually in addition to the others) a biometric. Biometric devices can break, so in most implementations, there is always a backup password that can be used if the biometric doesn't work. (Interestingly, the ink signature has a similar problem, because people can break their hands.)

With the TPM 2.0, it's possible to tie the use of a key directly to a biometric and prove that it's so tied. First a non-duplicatable key is created, with its authValue set so that a password isn't useful for authorization. This means only the policy can be used to authorize use of the key. The policy is set to only allow use of the key when authorized by

a biometric reader, using `TPM2_PolicySign` and a `policyRef` that is produced and signed by the biometric reader when it matches the person. This produces a key that can only be used to sign something if the biometric reader is convinced the person is who it thinks the person is. We call this key A.

Next a credentialed non-duplicable restricted signing key is used with `TPM2_Certify` to produce a signature over the name of key A. This signature binds the public portion of key A (which is in the name), the authValue (which are in the name), and the policy (which is in the name). By checking the credential of the restricted signing key, an attesting agent can verify that the certificate produced by `TPM2_Certify` is valid. Then, by hashing the public data of the key, the agent can verify that the name is correct. This then validates that the only way the key could be used for signing is by satisfying the policy, not by a password. The policy is then examined and, using the public key of the biometric device, is validated to be satisfied only if the user swiped their finger over the reader.

In this way, the electronic signature with the key becomes tied to the fingerprint biometric. In a similar way, producing certificates binding policies to keys can be used to prove to an auditor that the policies being used for keys meet a corporate standard for security. This in turn satisfies the last of the problems that EA was created to solve.

Summary

This chapter has examined the new enhanced authorization in the TPM 2.0, which can be used to authorize any entity in the TPM. You have seen that EA policies can be used to create logical combinations (AND and OR) of multiple kinds of assertions—everything from passwords and smart cards to the state of the TPM or the state of a remote machine. You have looked at examples of using EA for multiple users, multifactor authorization, and the means to create policies that allow flexible management. Many examples demonstrated the ways these commands can be used to solve varied problems. Then you saw how such policies can be satisfied. Finally, you saw how a key can be bound to its policy. EA in the TPM 2.0 is one of the most complex but also most useful new capabilities in the TPM 2.0 design.