TPM 2.0

David Wooten
Partner Architect
Microsoft Corp.
david.wooten@microsoft.com

AGENDA

- **TPM 2.0 Motivations**
- Key Hierarchies
- Authorizations
- Miscellaneous

Why TPM 2.0?

- Had to
- TPM 1.2 uses SHA1 and RSA 2048 as the only fully-supported algorithms
  - SHA1 is no longer considered adequate
  - RSA is not being recommended for security strengths above 112 bits

Key bits / bit of security strength

RSA Key Size in Bits	Security Strength in Bits
1024	80 (12.5)
2048	112 (18.3)
3072 SECRET	128 (24)
7680 TOP SECRET	192 (40)

From SP800-57, Table 2

TPM 1.2 Basic Capabilities

- Cryptographic signing – for PCR attestation key association
- NV memory – to store an Endorsement Key (EK), SRK, and an EK certificate
- Key exchange – for certificates from a CA
- Authorization – key usage or unseal
- Key hierarchy – for multiple AIK
- All of these features are supported in TPM 2.0 but in an algorithm agile way

Requested Additional Features for TPM 2.0

- Want to put other platform stuff in the TPM NV (like TXT-related data)
- Want a way for the platform manufacturer to use the TPM even if the system owner turns the TPM off
- Want more monotonic counters
- Do not want any monotonic counters
- Fix PCR brittleness
- Fix authorization “substitution” problem
- Want a PCR for …
- Want complex, multi-factor authorization
- Want simple authorization
- Want to be able to replace the EK
- Want to prevent the EK from being replaced
- Want to support ~~other~~ any future algorithms / my favorite algorithms / my boss’s favorite algorithms

AGENDA

- TPM 2.0 Motivations
- **Key Hierarchies**
- Authorizations
- Miscellaneous

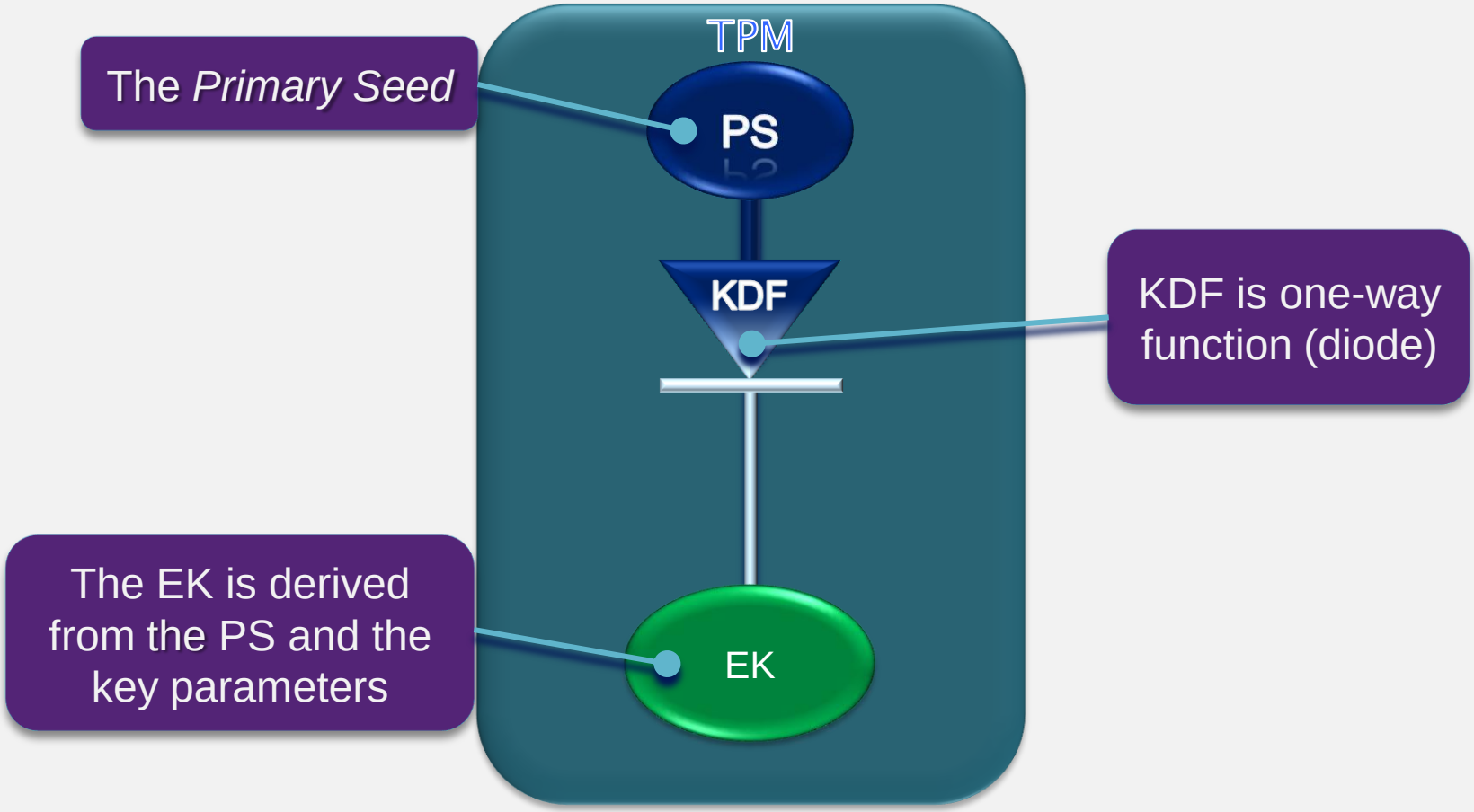
First Problem: EK Provisioning

- Need an EK for identification of the TPM
- Easy when everyone is happy using the same algorithm
- When we need different algorithms, the problem for the TPM manufacturer gets much harder
  - Each key-pair takes up NV memory
- What key pair(s) does the TPM vendor install?

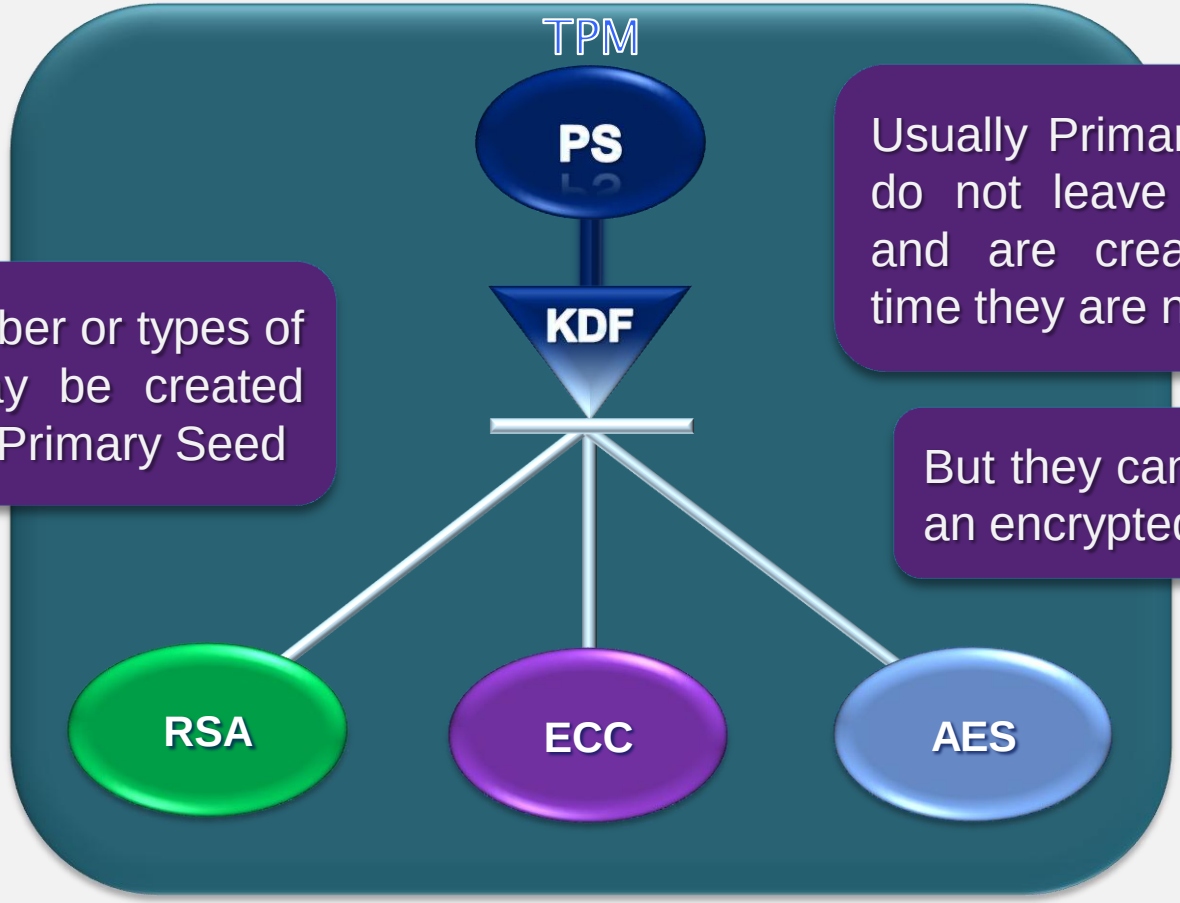
EK Provisioning – The solution

- Use one master seed value (Primary Seed) and use KDF to generate any number of (a)symmetric keys
- Allows the asymmetric key derivation to be reproducible
  - KDF is deterministic
    - When called with the same parameters, the same key is generated
  - Use Primary Seed and a caller-provided key parameters as inputs to a standard KDF
- The asymmetric keys become ‘aliases’ of the Primary Seed
  - Now TPM may have as many EKs as necessary …
  - … at the same constant price
  - Let customer decide which one they want to use

EK from a Primary Seed



Multiple EK



Usually Primary Objects do not leave the TPM and are created each time they are needed

Any number or types of keys may be created from the Primary Seed

But they can leave TPM in an encrypted context blob

These are all reproducible "aliases" for the Primary Seed and are called *Primary Keys* or *Primary Objects*

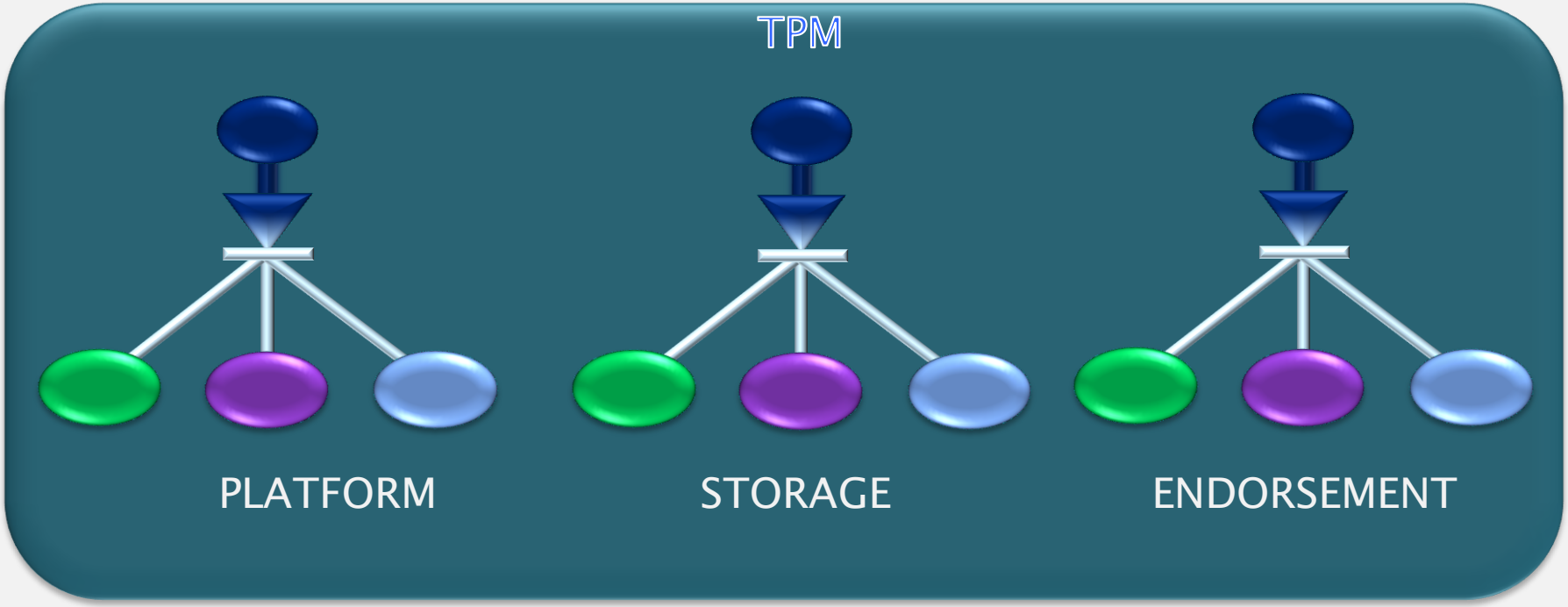
User Provisioning

- TPM manufacturer
  - Is allowed to inject a primary seed
  - Generates EKs for various markets and certifies them
- TPM is shipped without EK installed
- User creates EK using the same template as TPM vendor
  - Same parameters yield the same key
  - User will have an EK that the TPM vendor have certified
  - User could tell the TPM to make the key persistent
    - Store in NV
- This process is enabled, but NOT mandated, by the spec

The Hierarchies

- A TPM can have from one to three hierarchies
  - *Platform* hierarchy for use by the BIOS/SMM/UEFI
  - *Storage* hierarchy for general purpose use by the OS (on behalf of the various users)
  - *Endorsement* hierarchy for TPM identification
- Each hierarchy has separate resources
  - A primary seed
  - An enable flag
  - An authorization value (authValue)
  - An authorization policy (authPolicy)
- When the enable for the hierarchy is CLEAR, neither the *authValue* nor the *authPolicy* for that hierarchy can be used

Three Hierarchies

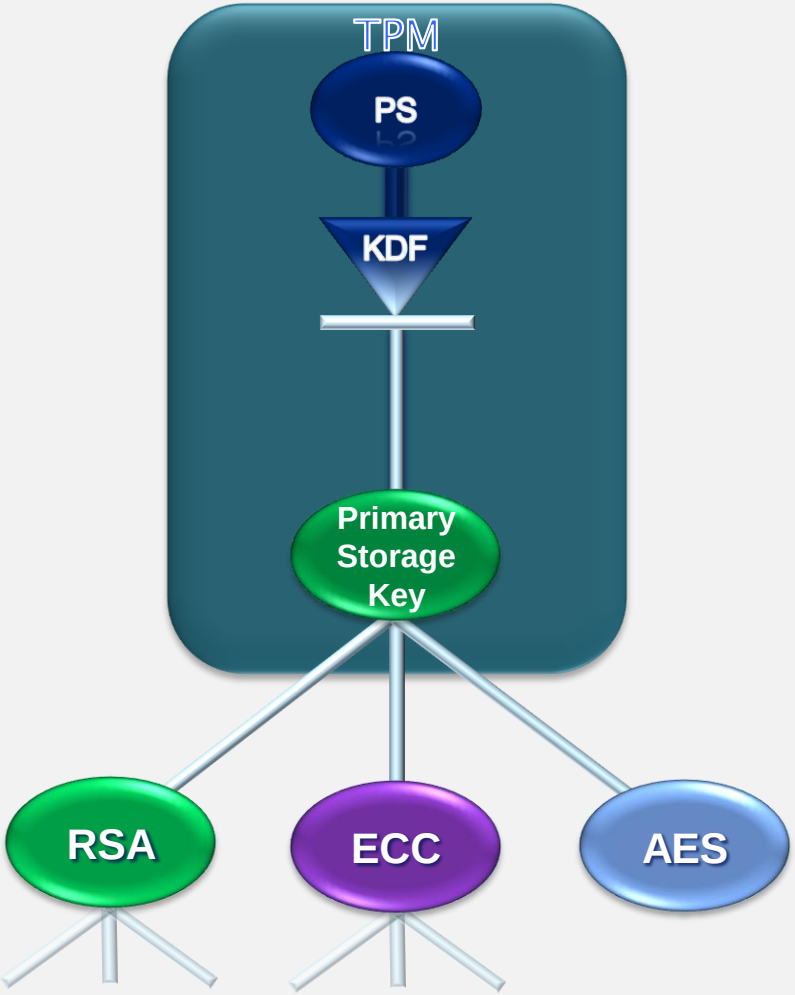


To support these three hierarchies, the only persistent storage required in the TPM are the three, Primary Seeds – 256-/512-bits each

Hierarchy Lifetimes

- Hierarchies have different lifetimes
  - Determined by when their Primary Seed changes.
- Storage Primary Seed (***SPS***) changes when system owner changes
- Endorsement hierarchy is created by the TPM vendor and may or may not change over the lifetime of the TPM
  - One OEM said they'd like to change the Endorsement Primary Seed (***EPS***) when the system comes back for refurbishment
  - Other OEM said they want to change the EPS when hell freezes over
  - The command to change the EPS requires *platformAuth* so the OEM gets to decide when/if the EPS ever changes
- Platform Primary Seed (***PPS***) is installed by the OEM when the platform is manufactured

Extending the Hierarchies



As with TPM 1.2, the sensitive portion of the "child" key is only decrypted when the key is loaded into the TPM

Symmetric Hierarchy

- In TPM 1.2, the “child” keys were encrypted using the asymmetric parent key.
  - Every key load was an asymmetric key operation
- In TPM 2.0, the parent has an additional seed value that is used to generate the symmetric keys used to protect the child keys
  - Encryption and HMAC keys
  - Each set of keys is unique to the child
  - Loading a child is a pair of symmetric operations (HMAC and decrypt)
- In TPM 2.0, the asymmetric part of a storage key used for secret sharing
  - Key import
  - Activating an identity
  - Starting an authorization session

Invalidating Keys

- Each Primary Seed has a command to cause it to change:
  - TPM2_Clear() – Storage Primary Seed
  - TPM2_ChangeEPS() – Endorsement Primary Seed
  - TPM2_ChangePPS() – Platform Primary Seed
- The seeds are changed, not left at zero
  - Not safe to leave seeds in a state that would cause two TPMs to generate the same keys
- The commands that change the seeds, also flush from TPM memory any loaded Objects in the associated hierarchy
- Also removes from TPM NV any persistent Objects and Indices that belong to that hierarchy

AGENDA

- TPM 2.0 Motivations
- Key Hierarchies
- **Authorizations**
- Miscellaneous

Authorization Elements

- TPM 1.2 has only 3 authorization elements
  - Authorization value
  - PCR state
  - “Locality” – hardware privilege level
  - 8 total combinations of these simple elements
- TPM 2.0 has 12 authorization elements… so far
  - Authorization value
  - Locality
  - Asymmetric signature
  - Symmetric shared secret
  - Time Limited
  - Specific Command
  - PCR State
  - “Physical Presence”
  - Specific objects
  - Duplication
  - NV Written
  - Contents of NV
- Authorization elements can be combined using logical constructs (AND / OR) to give fine-grained access control over TPM 2.0 keys and data

Authorizing use of an Object

- Objects have an authorization value (*authValue)* and my have an authorization policy. (*authPolicy)*
  - *authValue* is in the sensitive area in an object and can be changed
  - *authPolicy* is in the public part of the object and cannot be changed
    - Its like the PCRinfo of a TPM 1.2 key
- When authorizing using the *authValue,* can use an HMAC session or provide the *authValue* as a clear-text password
- When authorizing using an *authPolicy,* the TPM compares an accumulated policy hash against the *authPolicy* of the object
  - Similar to how a TPM 1.2 creates a digest of selected PCR and compares the digest to the value in the object's *PCRinfo*
  - Difference is, the digest is an accumulation of extend values based on satisfying policy commands.

Starting an Authorization Session

- Use TPM2_StartAuthSession()
  - This command is a bit of a combination of TPM_OIAP, TPM_OSAP, and TPM_EstablishTransportSession.
  - Indicate if the session is going to be used for HMAC or policy authorizations
- For an authorization using the *authValue,* use an HMAC like the OIAP or OSAP
- For policy authorizations, build the *policyHash* in the session by executing specific policy commands
- When the proper sequence of policy commands has been executed, the *policyHash* accumulated in the session context will match the *authPolicy* of an object

Simple Policy Example

- Want to mimic the use of PCR for authorization and also provide an authorization secret
- Need to know what value to use for the *authPolicy* of the object
- Start a “trial” policy session
  - TPM2_StartAuthSession(*sessionType* == TPM_SE_TRIAL)
- Execute TPM2_PolicyPCR() with the handle of the trial policy session as *policySession*

Simple Policy Example

- TPM2_PolicyPCR() causes the TPM to update the policyDigest

$$policyDigest_{new} := \mathbf{H}(policyDigest_{old} \,||\, \text{TPM\_CC\_PolicyPCR} \,||\, pcrs \,||\, \text{digestTPM})$$

Table 121 — TPM2_PolicyPCR Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or encrypted session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyPCR
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPM2B_DIGEST	pcrDigest	expected digest value of the selected PCR using the hash algorithm of the session; may be zero length
TPML_PCR_SELECTION	pcrs	the PCR to include in the check digest

Simple Policy Example

- TPM updates the *policyDigest*

$$policyDigest_{new} := \mathbf{H}_i(policyDigest_{old} \,||\, TPM\_CC\_PolicyAuthValue)$$

Table 145 — TPM2_PolicyGetDigest Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyAuthValue
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None

Simple Policy Example

- Read the computed *policyDigest* (TPM2_PolicyGetDigest()) and use that as the *authPolicy* of the created object

Table 184 — Definition of TPMT_PUBLIC Structure

Parameter	Type	Description
type	TPMI_ALG_PUBLIC	"algorithm" associated with this object
nameAlg	+TPMI_ALG_HASH	algorithm used for computing the Name of the object NOTE The "+" indicates that the instance of a TPMT_PUBLIC may have a "+" to indicate that the nameAlg may be TPM_ALG_NULL.
objectAttributes	TPMA_OBJECT	attributes that, along with type, determine the manipulations of this object
authPolicy	TPM2B_DIGEST	optional policy for using this key The policy is computed using the nameAlg of the object. NOTE Shall be the Empty Buffer if no authorization policy is present.
[type]parameters	TPMU_PUBLIC_PARMS	the algorithm or structure details
[type]unique	TPMU_PUBLIC_ID	the unique identifier of the structure For an asymmetric key, this would be the public key.

Simply Policy Example

- To use the policy do it all over again, but this time for real
  - TPM2_StartAuthSession(*sessionType* == TPM_SE_POLICY)
  - TPM2_PolicyPCR()
  - TPM2_PolicyAuthValue()
- If the PCR have the desired value, then the policyDigest will have the same value as the authPolicy of the object
- Use the policy session as the authorization session to use the object

AGENDA

- TPM 2.0 Motivations
- Key Hierarchies
- Authorizations
- **Miscellaneous**

Time

- TPM 1.2 has no notion of time
- TPM 2.0 has two time values that advance as long as the TPM is powered
  - Timer – reset on each TPM Reset
  - Clock – reset on each TPM Clear
- Time values periodically saved to NV memory
  - No less often that 2^{12} seconds (~68 minutes)

NV Memory Types

- TPM 1.2 has an NV Index that has no special capabilities other than authorized access for read or write
- TPM 2.0 adds function-specific Index types:
  - Extend – works like a PCR
  - Bits – SET but not CLEAR for revocation
  - Counter – replaces the monotonic counter in 1.2
- TPM 2.0 has “orderly” Indexes
  - RAM-backed so that they can be updated at high frequency without danger of NV wear out

Signing Keys

- TPM 1.2 has special purpose signing keys, such as AIK
- TPM 2.0 signing keys can be used more generally
  - An unrestricted key will sign anything
  - A restricted key will only sign a digest done by the TPM
    - An internally-generated data structure has a special value as it first bytes
    - When TPM does a hash of externally-provided data, it will verify that the first bytes don't have this special value
- Among other things, this lets the TPM sign a PKCS#10 certificate request with a key that is used for attestation

Credentials

- TPM 1.2 certificate protocol (TPM_ActivateIdentity()) would only work with restricted set of keys
  - EK, SRK, and an AIK that is a child of the SRK
- TPM 2.0 is more generalized
  - The decryption can be done by any key and not just the EK
  - The “credentialed” key can be any key and not just an AIK
  - Let’s any decryption key with a certificate be used to associate a credential with any other key.

Duplication / Migration

- In TPM 1.2, there were several commands relating to migration
  - TPM_CreateMigratinoBlob
  - TPM_ConvertMigrationBlob
  - TPM_AuthorizeMigrationKey
  - TPM_MIgrateKey
  - TPM_CMK_SetRestrictions
  - TPM_CMK_AppoveMA
  - TPM_CMK_CreateKey
  - TPM_CMK_CreateTicket
  - TPM_CMK_CreateBlob
  - TPM_CMK_ConvertMigration
- In TPM 2.0, there are two commands for duplication
  - TPM2_Duplicate
  - TPM2_Import

Conclusion

- TPM 2.0 development was necessitated by choice of TPM 1.2 cryptographic functions
- Engineers took the opportunity to do a major overhaul of the TPM 2.0 architecture to:
  - Reduce the chances that TPM 3.0 will be need because of cryptographic issues
  - Used the 10 years of experience with TPM 1.2 to indicate what extra functions should be added and what functions should be removed

Questions?

Thank You