Trusted Computing: Introduction & Applications

Lecture 4: Authorisation, sealing and binding

Dr. Andreas U. Schmidt

Fraunhofer Institute for Secure Information Technology SIT, Darmstadt, Germany

Literature

1. C. J. Mitchell (ed.), *Trusted Computing*. IEE Press, 2005.
2. D. C. Blight. Trusted Computing. Blackhat Windows 2004. https://www.blackhat.com/presentations/win-usa-04/bh-win-04-blight/bh-win-04-blight.pdf
3. Eimear Gallery, Graeme Proudler Lecture at Royal Holloway: http://www.isg.rhul.ac.uk/files/IY5608_-_Lecture_2_Roots_of_Trust.pdf http://www.isg.rhul.ac.uk/files/IY5608_-_Lecture_3_Roots_of_Trust.pdf
4. Kerry Maletsky: Designing in A Designing in A Trusted Platform Module (TPM), Embedded Systems Conference, Boston, 2005 https://www.trustedcomputinggroup.org/news/presentations/TCG_ESC_Atmel.pdf
5. Nihal A. D'Cunha Exploring the Integration of Memory Management and Trusted Computing. Dartmouth Computer Science Technical Report TR2007-594. MSc thesis. DARTMOUTH COLLEGEHanover, New Hampshire May 31st, 2007 ftp://ftp.cs.dartmouth.edu/TR/TR2007-594.pdf

Authorisation (from [3.])

- Required in order to:
  - Authenticate the owner of the TPM.
  - Authorise the use of/access to a TPM protected object.
  - Authorise migration of a migratable TPM key.
  - Authorise the use of a TPM capability.
- Two methods:
  - Physical presence and
  - Authorisation data.

Physical presence

- Physical presence implies direct interaction by a person with the trusted platform/TPM.
- The actual implementation is decided by the TPM and platform manufacturers.
  - (the strength of the protection mechanism is determined by an evaluation of the platform)
- Guiding principle for designers – the protection mechanism should be difficult or impossible to spoof by rogue software.
- Example – hardware switch (very difficult to circumvent by rogue software or remote attackers).
- The physical presence indication is implemented as a flag in volatile memory – PhysicalPresenceV flag.
- When it is set to TRUE – commands which can function include:
  - TPM_PhysicalEnable
  - TPM_PhysicalDisable
  - TPM_PhysicalSetDeactivated
  - TPM_ForceClear
  - TPM_SetOwnerInstall
- Precautions must be taken by designers to ensure this flag is not maskable (dedicated bus cycle could be used).

Authorisation data

- Authorisation data
  - Length: 20 bytes. For example, a hashed password, or 20 bytes from a smartcard, may be used.
- Sample types:
  - TPM owner authorisation data – owner authorised commands;
  - Protected object authorisation data;
  - Key migration authorisation data.
- The TPM neither knows or cares about the content of an authorisation value:
  - Each individual authorisation value may be unique or may be the same as another value.
- The TPM is designed, however, to conceal and protect authorisation values at all stages, i.e.:
  - During initial entry of an authorisation value;
  - When proving knowledge of a specific authorisation value;
  - When changing an authorisation value.
- There are two protocols which can be used by a requester to prove that they have knowledge of a particular authorisation value. They have been designed in order to protect against:
  - Man in the middle attacks;
  - Replay; and
  - The exposure of the authorisation data.
- Object independent authorisation protocol (OIAP)
- Object specific authorisation protocol (OSAP)

Object independent authorisation protocol (OIAP)

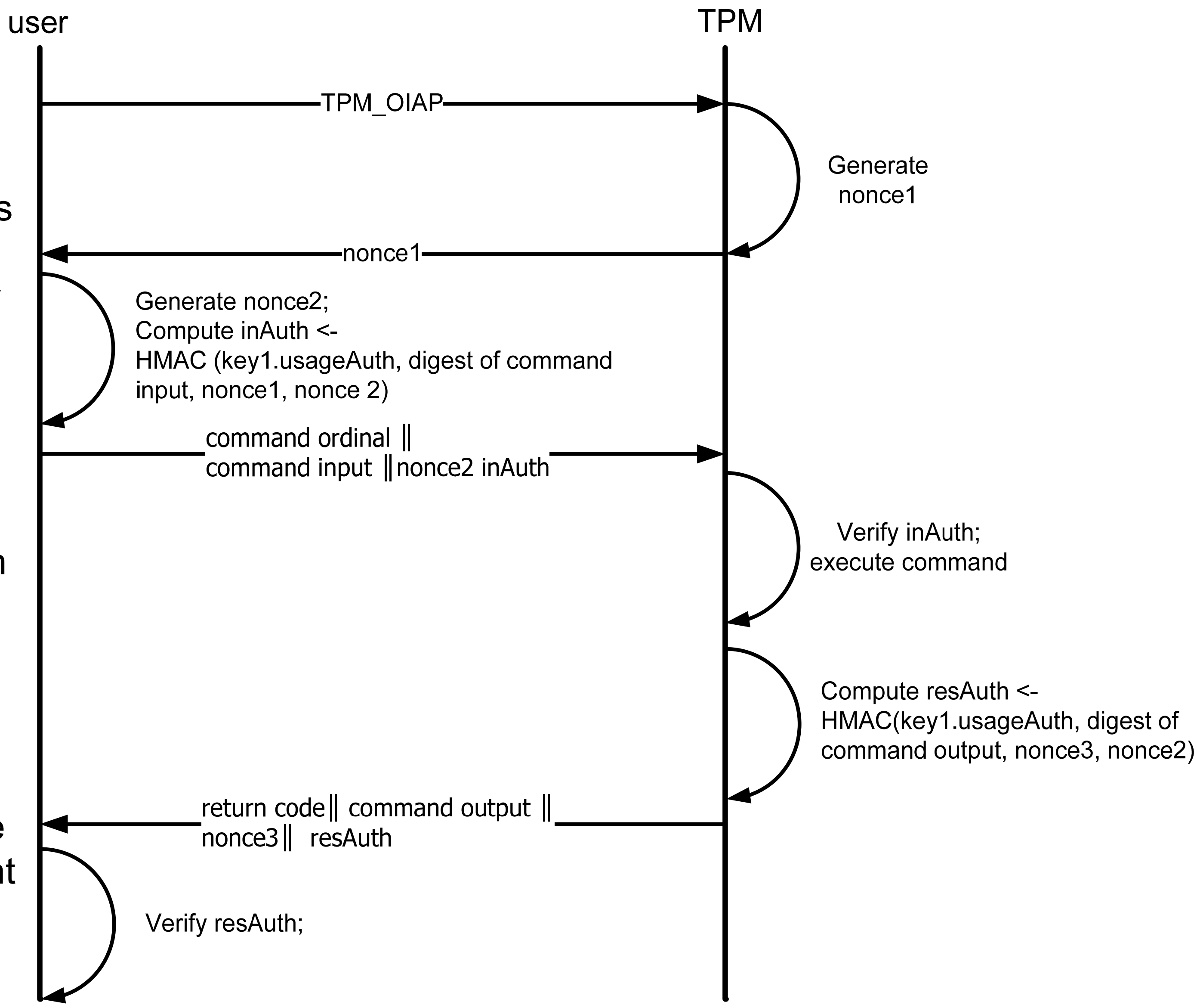
- Used by a requester to prove that it has knowledge of a particular authorisation value.
- Challenge-response based on a “rolling nonce” paradigm:
  - This requires that a nonce is sent from one side and returned in the reply.
  - Designed to prevent replay attacks and man-in-the-middle attack.
- Example:
  - Assume a TPM command that uses key1;
  - The user must know the AuthData for key1 in order to use the command;
  - Assume the caller does not need to authorise the use of key1 for more than one command.

OIAP

Once an OIAP session has been established, its nonces can be used to authorise the use of any object managed by the TPM.

The session can live indefinitely until either party requests a session termination.

This is the preferred protocol as it allows usage of the same session to provide authorisation for different objects.

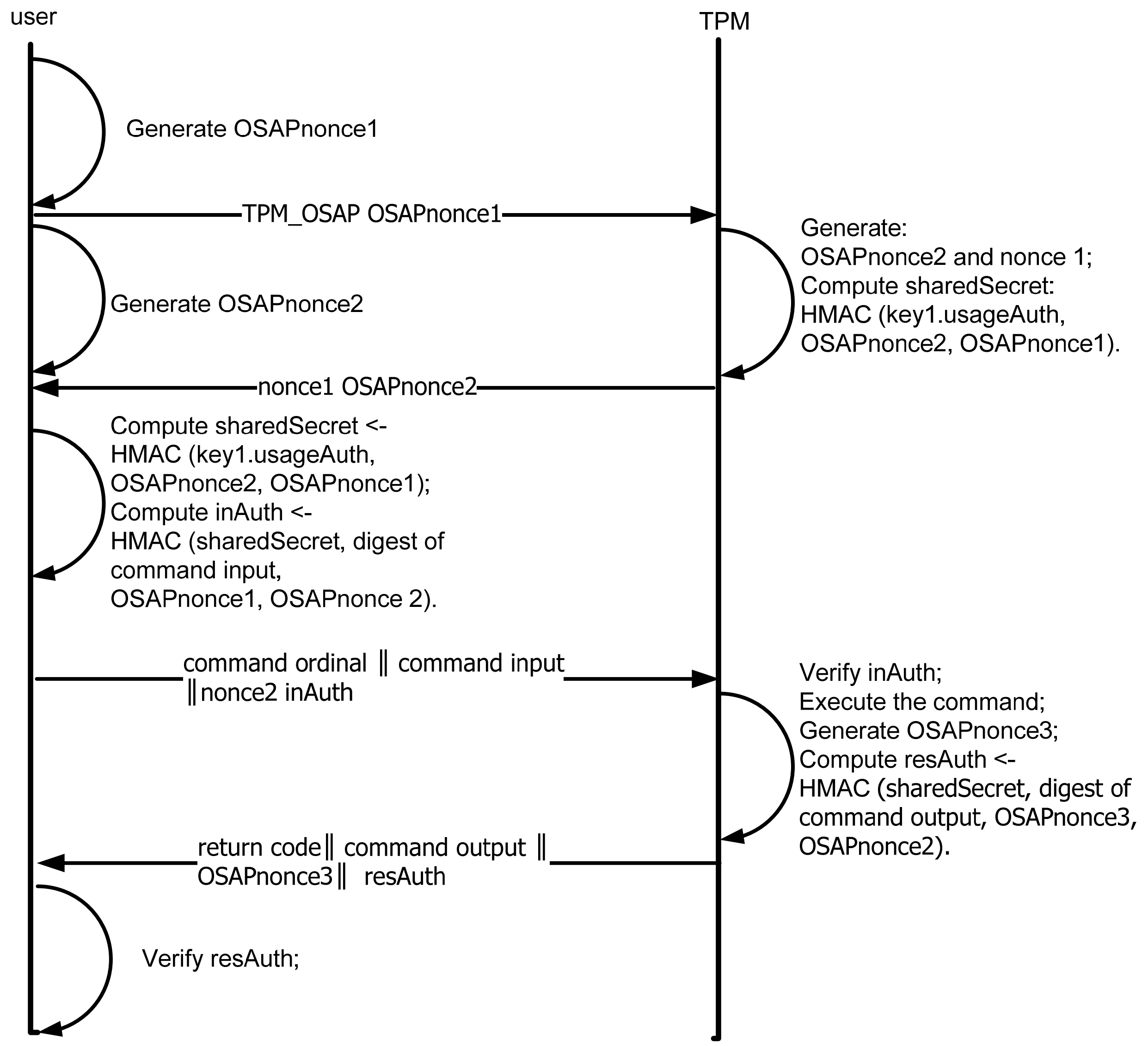

user
TPM
TPM_OIAP
Generate nonce1
nonce1
Generate nonce2;
Compute inAuth <-
HMAC (key1.usageAuth, digest of command input, nonce1, nonce 2)
command ordinal ‖ command input ‖nonce2 inAuth
Verify inAuth; execute command
Compute resAuth <-
HMAC(key1.usageAuth, digest of command output, nonce3, nonce2)
return code‖ command output ‖ nonce3‖ resAuth
Verify resAuth;

Object specific authorisation protocol (OSAP)

- Also may be used by a requester to prove that they have knowledge of a particular authorisation value and is also based upon the “rolling nonce” paradigm.
- Using OSAP, a distinct session needs to be set up for each object that requires authorisation.
- However, it enables a caller to authorise the use of a particular object multiple times without having to input the AuthData more than once.
- This protocol is required in order to set or reset AuthData within a “meta-session”
- Example:
  - Assume a TPM command that uses key1;
  - The user must know the AuthData for key1 in order to use this command;
  - Assume the caller needs to authorise the use of key1 multiple times but does not wish to input the AuthData more than once.

OSAP

If the TPM user wishes to send another command using the same session (i.e. another command on the same object) the session may be kept open.



user
TPM
Generate OSAPnonce1
TPM_OSAP OSAPnonce1
Generate OSAPnonce2
Generate: OSAPnonce2 and nonce 1; Compute sharedSecret: HMAC (key1.usageAuth, OSAPnonce2, OSAPnonce1).
nonce1 OSAPnonce2
Compute sharedSecret <- HMAC (key1.usageAuth, OSAPnonce2, OSAPnonce1); Compute inAuth <- HMAC (sharedSecret, digest of command input, OSAPnonce1, OSAPnonce 2).
command ordinal ‖ command input ‖nonce2 inAuth
Verify inAuth; Execute the command; Generate OSAPnonce3; Compute resAuth <- HMAC (sharedSecret, digest of command output, OSAPnonce3, OSAPnonce2).
return code‖ command output ‖ OSAPnonce3‖ resAuth
Verify resAuth;

Authorisation data insertion protocol (ADIP)

- This protocol is used when a caller wishes to associate AuthData with a new object.
- When the creation process is started – OSAP is used.
- The caller and the TPM generate a shared secret by calculating the HMAC of the parent.Auth and the nonces exchanged during the OSAP protocol.
- This shared secret and a nonce generated by the TPM are then used by the caller and the TPM as input to the SHA-1 hash function to generate an ephemeral secret (the hash function output).
- This ephemeral secret is then XORed with the new authorisation data in order to protect it during insertion into the TPM.

AuthData change protocol (ADCP)

- The ADCP allows an object owner to change that object’s AuthData.
- An OSAP session must first be used to authorise use of the parent object.
- This OSAP session also provides the data required to generate an ephemeral secret which can then be used to encrypt the new AuthData, as described in ADIP.
- An OIAP or an OSAP session must also be established in order to authorise access to the object whose authorisation data is being changed.
- *Note* – If the authorisation data of the parent object is known to an entity, they can snoop/eavesdrop on an AuthData change protocol for a child of that entity and learn the newly chosen child AuthData.
- Example: If SRKAuth is a known to userA and userB, userA can snoop on userB while userB is changing the AuthData for a child of the SRK and deduce the child’s new AuthData.
- In order to prevent this, it is advised that ADCP is used inside a transport session.

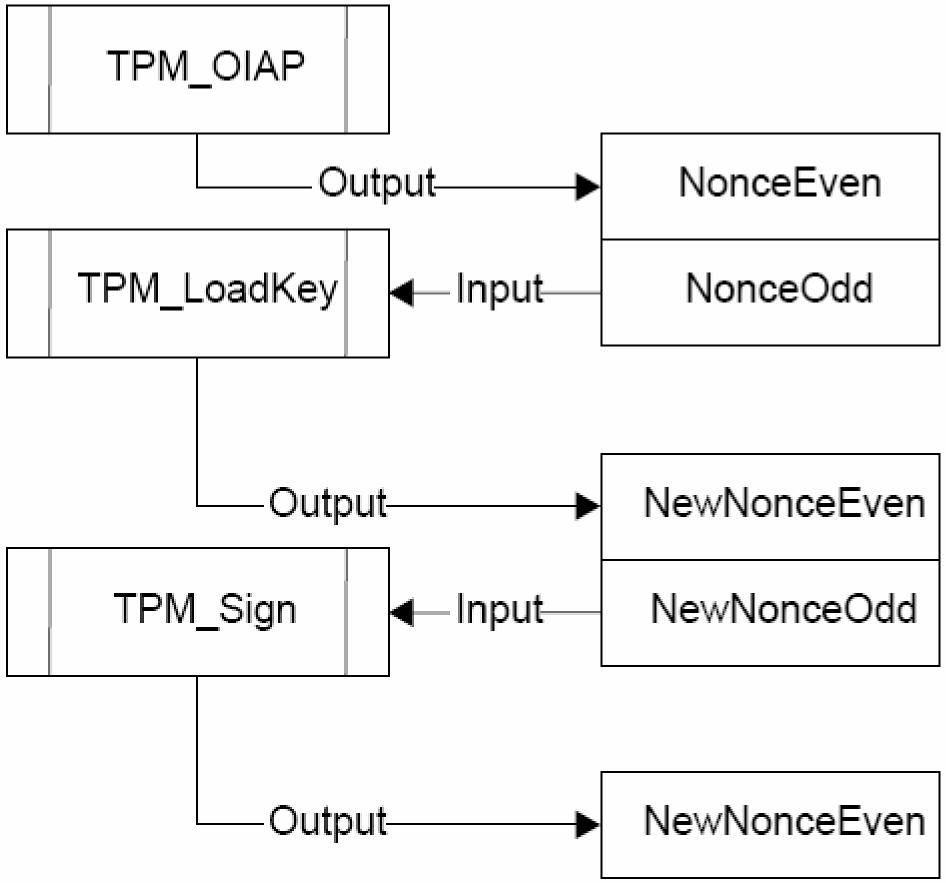
Authorisation technical detail ¼ (from [4.])

- Owner and user capabilities for auth.
  - TPM Authorization Block
    - Authorization chain dependent upon the initial creation of an authorization session
    - Authorization session creates initial nonce or TPM nonce and caches parameters required to calculate authorization digest
  - TPM Entities and Authorization Secrets
    - Entities know secrets used to authorize TPM operations (User/ Owner keys)
    - Ephemeral shared secrets can be generated by software using an entity secret so that multiple operations can be performed without repeated authorisation requests.
  - TPM Nonce and TSS Nonce
  - Authorization Session Lifetimes
    - Sessions are left open when TPM commands are successful.
    - Sessions are closed when TPM commands fail or when the continue authorization byte is set to zero.
    - The continue authorization byte can eliminate the need to explicitly call the TPM_TermHandle command.
  - Authorization Digest Calculation
    - To continue session with new nonces

Authorisation technical detail 2/4

TPM to TSS nonce flow example

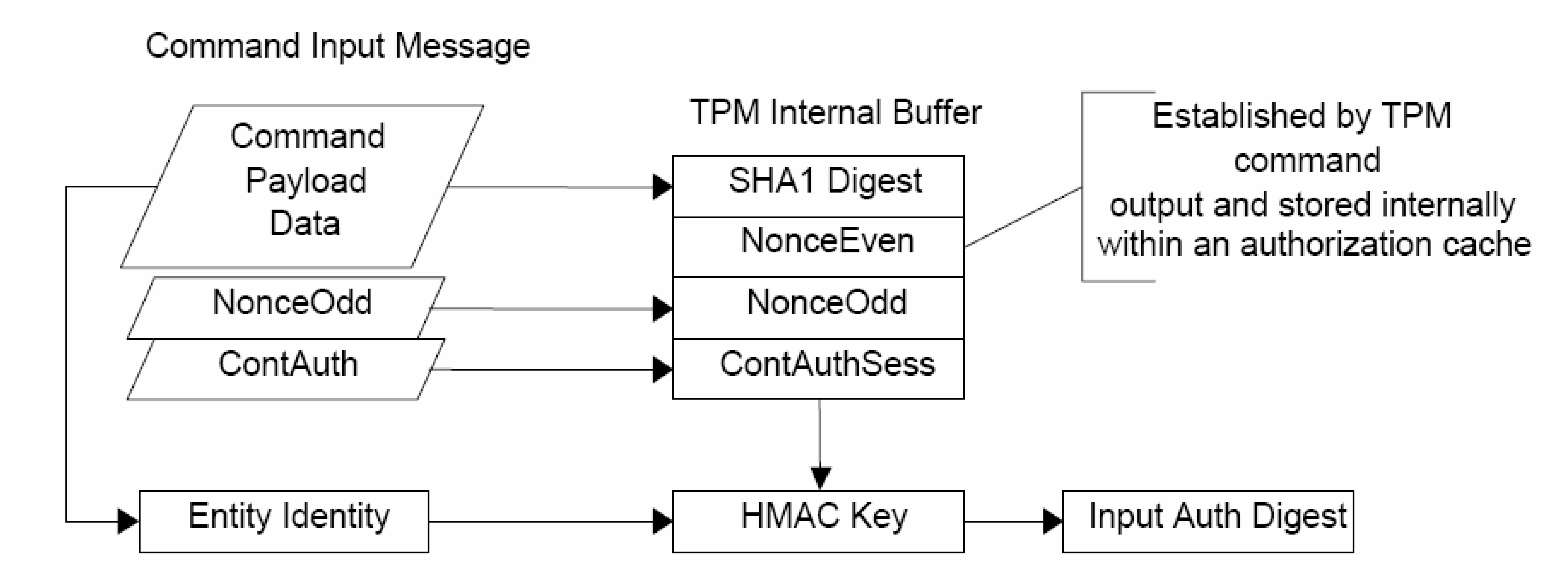
- 20-byte random value
- TPM nonce = nonce even
- TSS nonce = nonce odd
- Nonce chaining protects against message replay and man-in-the-middle attacks



Authorisation technical detail 3/4

Continuing a command session - input digest calculation

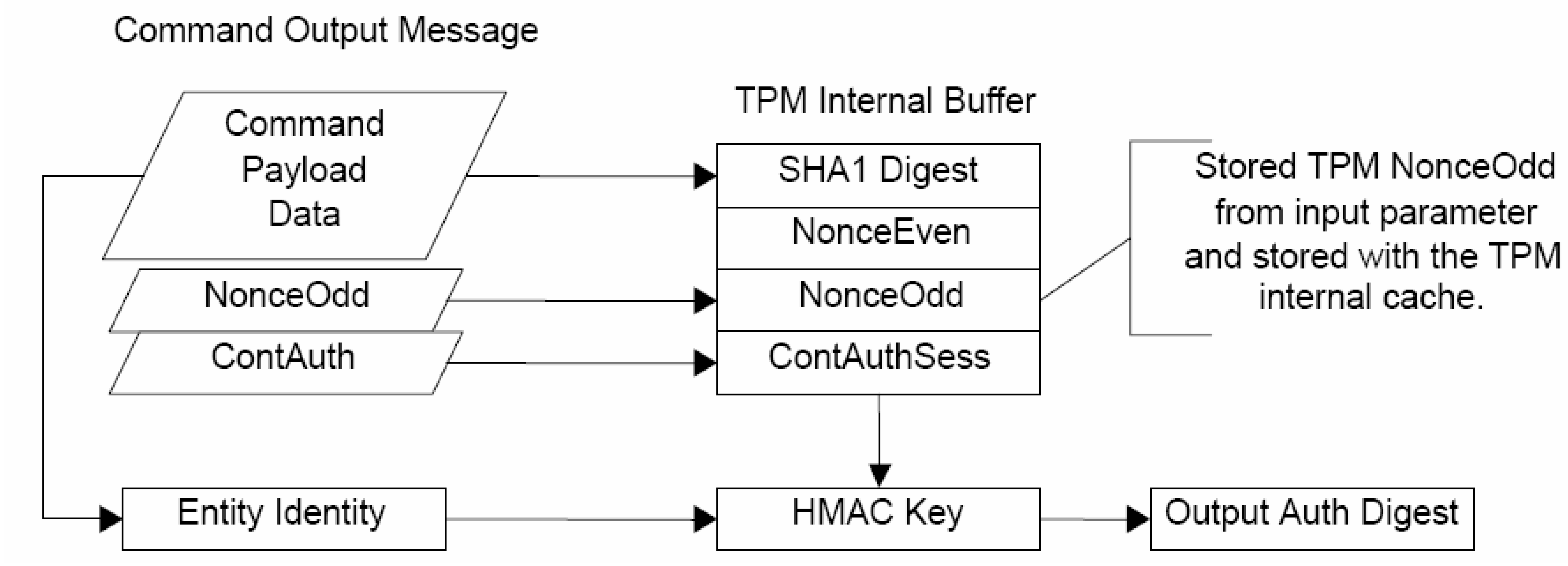
- Authorization digest calculated from selected parameters and the continuation authorisation parameter.



Continuing a command session 2/2 input digest calculation

Continuing a command session - output digest calculation

- An Output digest calculation is similar, but uses parameters associated with the output message.



Some useful commands 1/3

Some we know already:

- TPM_TakeOwnership
  - Typically done once when the TPM (system) is purchased. Creates the SRK, stores the TPM owner authorization secret, etc.
- TPM_Extend
  - Add data to a PCR. Specifically, the input is intended to be a 20 byte hash that is hashed into the current PCR value. The system must store the sequence of elements hashed into the PCR
- TPM_OIAP/OSAP
  - Start an authorization session. OIAP is “object independent” – the secret is required for each command, OSAP is ‘object specific” and a shared secret is used for multiple commands connected to the same entity

Some useful commands 2/3

Attestation and more:

- TPM_MakeIdentity
  - Along with TPM_ActivateIdentity and a CA, permits an authorized user to generate an Attestation Identity Key (AIK). Multiple AIKs can be generated, so the user can have multiple identities that are unique from each other.
- TPM_Quote
  - Sign the current value of a PCR – provides trust that the component hashes were presented to the TPM and in the order specified.

- TPM_GetRandom
  - Get a random number from the TPM random number generator. TPM's provide a very high quality FIPs type generator that can be trusted for all practical applications.
- TPM_GetCapability
  - Get configuration or status information from the TPM.

Some useful commands 3/3

Key handling, sealing, binding:

- TPM_LoadKey
  - Load a key onto the TPM. Typically it will be retained unless explicitly removed by the system. Key can be generated by the TPM or an external system.
- TPM_Sign
  - Sign data that is presented to the TPM. Can be used to sign an outgoing email message, for instance, or sign a challenge directed to the TPM by some remote system, such as a network host.
- TPM_UnBind
  - Decrypt data that has been encrypted with a public key, the private portion of which is stored on the TPM. Useful to exchange a random session key that's been used to encrypt a larger block
- TPM_Seal/Unseal
  - Connect data to a particular TPM. Seal encrypts the data so that it can only be decrypted by this particular TPM when it is in an environment that has three properties: 1) knows the sealed authorization secret, 2) Can load and use the parent key to which the data is sealed, 3) PCRs are in the right state.
- TPM_CreateWrapKey
  - Generate a unique RSA key according to the parameters specified (key size, migration, secret value, etc), wrap (encrypt) it with the parent key and send to the external system. The system should store the key in the key cache manager database.

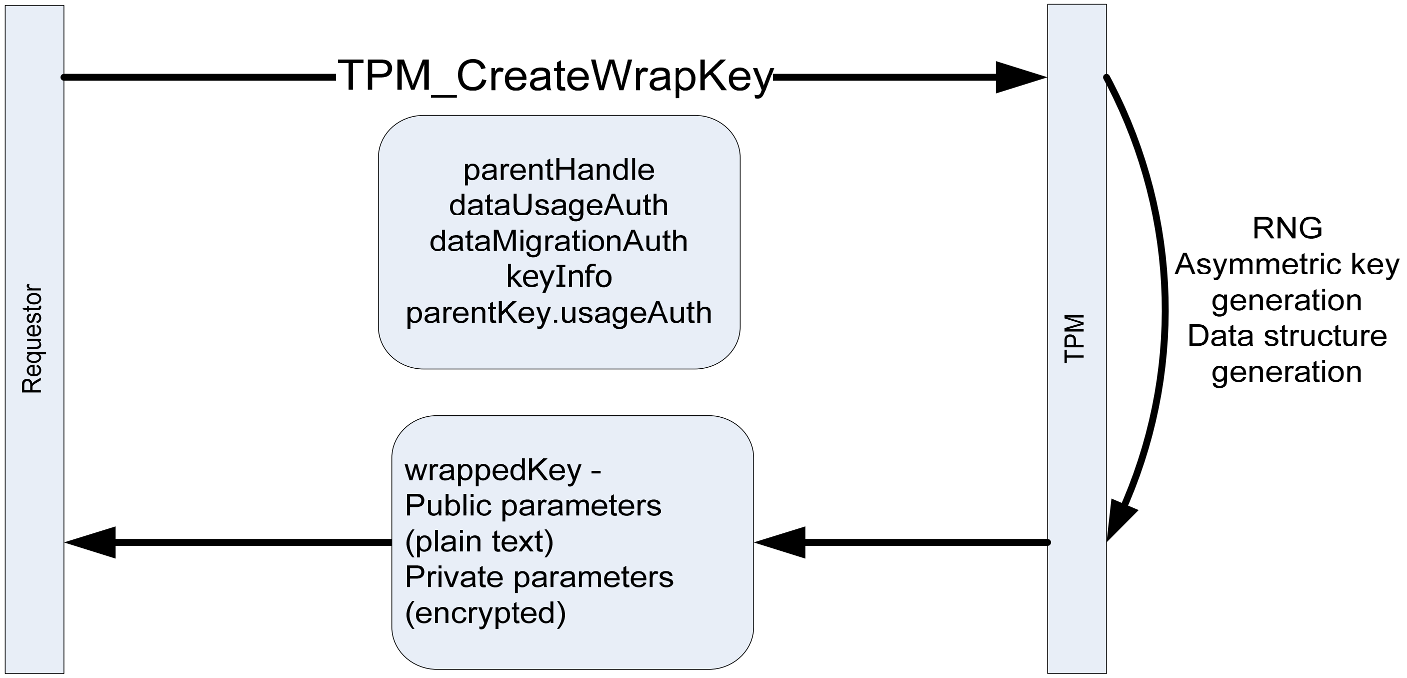
Authorisation for (non)migratable objects [3.]

- Non-migratable objects (keys):
  - Locked to an individual TPM;
  - Never duplicated;
  - The private keys from non-migratable key pairs are never available in plaintext outside of the TPM;
  - Non-migratable key pairs must be created inside the TPM.
- The TPM never creates any arbitrary data – in this way, arbitrary data is always generated outside the TPM and may be duplicated ad infinitum by its owner.
- In the strictest sense – TPM data objects are always migratable.

- Migratable objects:
  - Can be created outside the TPM and protected by the TPM, or created inside by TPM;
  - Can be replicated ad infinitum by its owner;
  - The extent of duplication of migratable keys is known only to the owner of that key.
- The essential architectural difference between migratable and non-migratable TPM keys is the source of their migration authorisation data:
  - The migration authorisation data for non-migratable keys is known only to the TPM - tpmProof
  - The owner of a migratable key creates the migration authorisation data.

Creating a wrapped key

- TPM_CreateWrapKey
  - Delivers a data structure in the key hierarchy
  - Depends on parent key
  - Authorisation values (optional):
    - dataUsageAuth : encrypted usage authorisation for the key created
    - dataMigrationAuth : encrypted migration authorisation for the key created



Data structure of created key

- keyInfo and wrappedKey – TPM_key type:
  - version
  - keyUsage : operations permitted with the key
  - keyFlags : migration
  - authDataUsage : conditions when it is required that authorisation be presented
  - algorithmParams : information regarding the algorithms for this key
  - PCRInfoSize
  - PCRInfo : pcrSelection, digestAtCreation, digestAtRelease
  - pubKey
  - encDataSize
  - encData : Enc(authValues, pubDataDigest (integrity check), privKey)

Loading a key

- This capability is used to load a (plaintext) public and a (ciphertext) private TPM key into the TPM.
- Must be used before a key can be used to seal, unseal, unbind, sign or perform any other such action.
- This command will fail if:
  - The version of the wrapped key is not the current TPM version;
  - There is an integrity check fail;
  - digestAtRelease for the parent TPM key does not match the current PCRs;
  - The key is non-migratable but was not generated on this platform (the tpmProof value doesn't match).

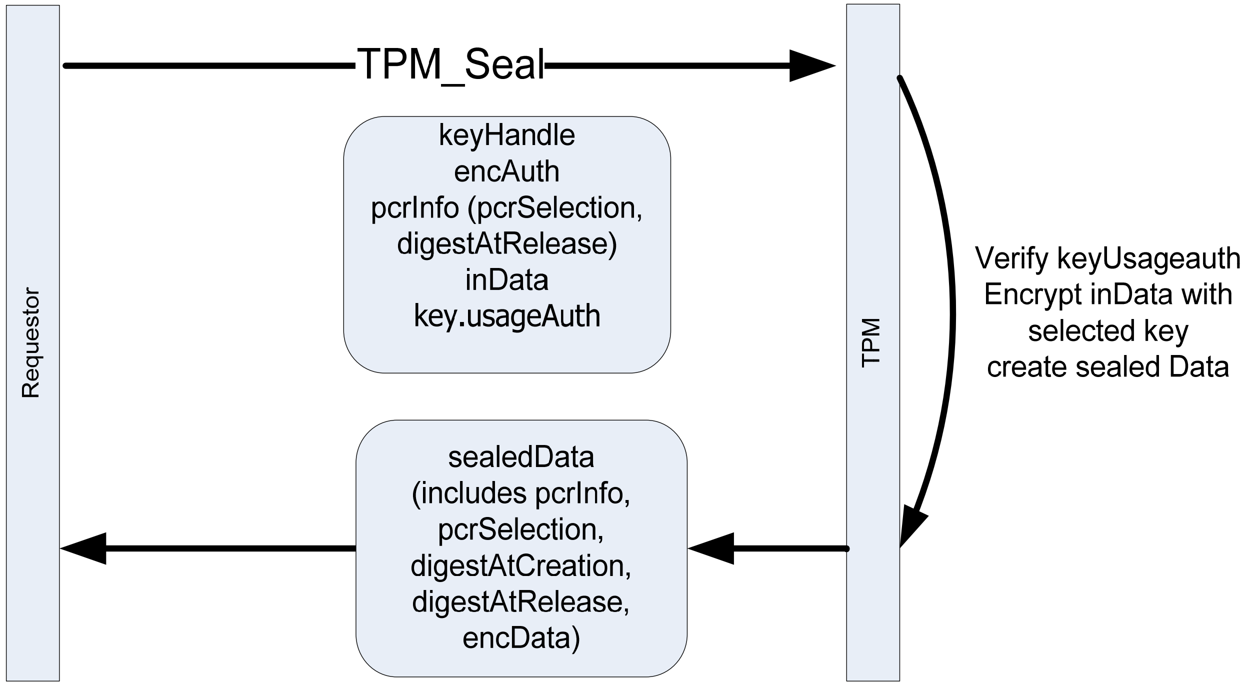
- Note the difference:
- When you are **loading a key**, you must demonstrate
  - knowledge of the AuthData for the parent key and match the DigestAtRelease associated with the parent key.
- When you are **using a key**, you must demonstrate
  - knowledge of AuthData for the key and match the DigestAtRelease associated with the key.

Binding and Unbinding

- *Binding* is the traditional operation of encrypting a message using a public key.
  - The sender uses the public key of the intended recipient to encrypt the message.
  - The message is only recoverable by decryption using the recipient's private key. When the private key is managed by the TPM as a nonmigratable key only the TPM that created the key may use it.
  - Hence, a message encrypted with the public key, "bound" to a particular instance of a TPM.
  - It is possible to create migratable private keys that are transferable between multiple TPM devices. As such, binding has no special significance beyond encryption.
- TPM_UnBind – Used for decrypting data which has been encrypted externally to the TPM using a bind key.
- TPM_UnBind takes the data blob that is the result of a TSS_Bind command and decrypts it for export to the User. The caller must authorise the use of the key that will decrypt the incoming blob.

Sealing and Unsealing

- *Sealed* messages are bound to a set of platform metrics specified by the message sender to specify platform configuration state that must exist before decryption will be allowed.
- Sealing associates the encrypted message with a set of PCR register values and a non-migratable asymmetric key.
- TPM_Seal states the authorisation data and PCR values that must be used to recover the data with TPM_Unseal

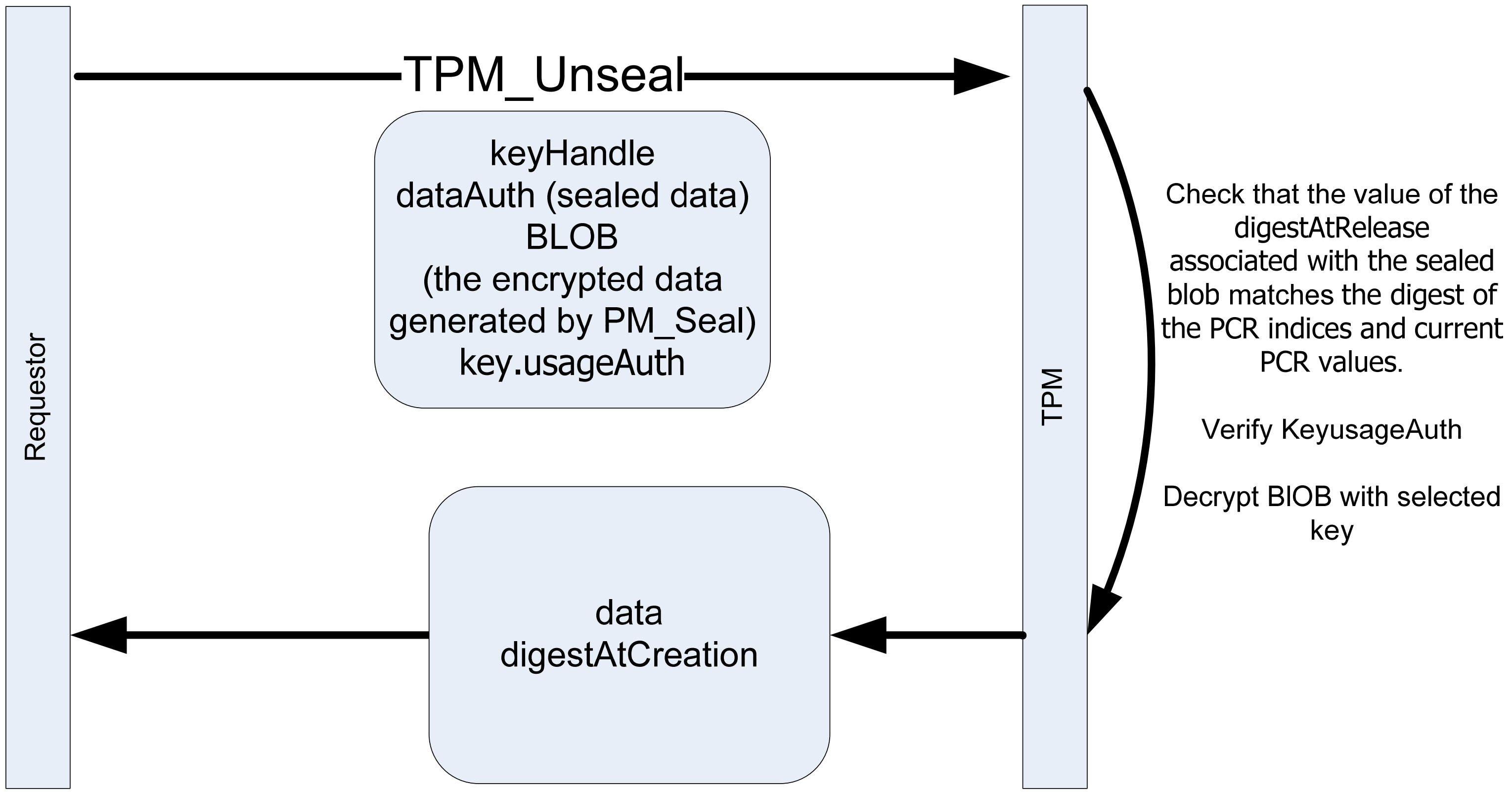


Requestor
TPM_Seal
keyHandle
encAuth
pcrInfo (pcrSelection, digestAtRelease)
inData
key.usageAuth
TPM
Verify keyUsageauth
Encrypt inData with selected key
create sealed Data
sealedData (includes pcrInfo, pcrSelection, digestAtCreation, digestAtRelease, encData)

Sealing parameters and handling

- pcrInfo:
  - pcrSelection – the selection of PCRs to which the data is bound.
  - digestAtCreation – the composite digest value of the PCR values, at the time when the sealing is performed.
  - digestAtRelease – the digest of the PCR indices and PCR values to verify when revealing Sealed Data that was associated with PCRs and encrypted.
- encData:
  - EK(authData, tpmProof, digest of pcrInfo, data)
- The application which calls the TPM_Seal or management software on the platform:
  - Collects a list of the current PCR values from the TPM (ones used to create digestAtCreation); and
  - Stores this list and a list of the selected target PCRs with the TPM protected object.

Unsealing

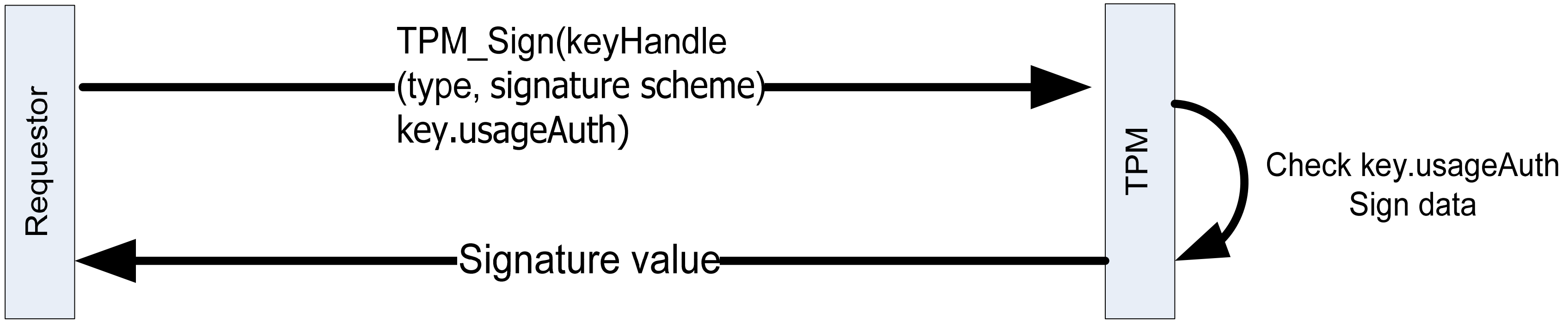

Requestor
TPM
TPM_Unseal
keyHandle
dataAuth (sealed data)
BLOB
(the encrypted data
generated by PM_Seal)
key.usageAuth
Check that the value of the digestAtRelease associated with the sealed blob matches the digest of the PCR indices and current PCR values.
Verify KeyusageAuth
Decrypt BIOB with selected key
data
digestAtCreation

Unsealing

- When the unsealed data has been returned by the TPM, the management software can verify the software state of the platform when the TPM object was created.
- The management software:
  - Collects the record of the software state at creation (digestAtCreation from the TPM object) and the list of PCR indices and PCR values associated with the TPM object.
  - Computes a digest over the list of PCR indices and PCR values.
  - Compares the digest computed to the digestAtRelease.
  - If they are the same the PCR values accurately represents the record of the software state when the object was created – otherwise it does not.
- This allows management software to ensure that an object was created in the correct software environment, else rogue software may have created the protected object.

Further security services 1/4 – signing arbitrary data

- TPM_sign – signs data and returns a digital signature.

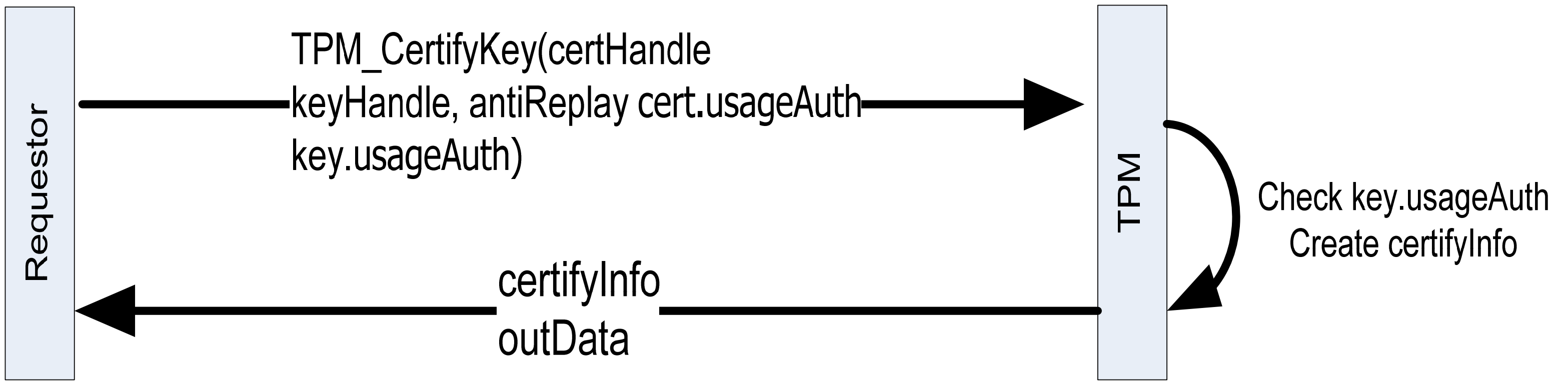


Further security services 2/4 – Sealed signing

- *Sealed-Signing*
  - Signing operations can also be linked to PCR registers as a way of increasing the assurance that the platform that signed the message meets a specific configuration requirement.
  - The verifier mandates that a signature must include a particular set of PCR registers.
  - The signer, during the signing operation, collects the values for the specified PCR registers and includes them in the message, and as part of the computation of the signed message digest.
  - The verifier can then inspect the PCR values supplied in the signed message, which is equivalent to inspecting the signing platform's configuration at the time the signature was generated.

Further security services 3/4 Certifying Keys

- TPM_CertifyKey – allows one key to certify the public portion of another key.
- A TPM attestation identity key may be used to certify nonmigratable keys but is not permitted to certify migratable keys.
- This enables a TPM to attest that a key was created on a TPM, will never be revealed outside of the TPM, and can only be used under certain conditions (given the host platform is in a specified software state).
- Signing and legacy keys may be used to certify both migratable and non-migratable keys.


Requestor
TPM_CertifyKey(certHandle keyHandle, antiReplay cert.usageAuth key.usageAuth)
TPM
Check key.usageAuth
Create certifyInfo
certifyInfo
outData

Further security services – Key certification

- certifyInfo:
  - version
  - keyUsage
  - keyFlags
  - authDataUsage
  - algorithmsParams
  - pubKeyDigest
  - data: anti replay nonce
  - parentPCRStatus : indicate if the parent key was wrapped to PCRs
  - PCRInfoSize
  - PCRInfo
- outData: a signature generated on certifyInfo