Trusted Computing: Introduction & Applications

Lecture 8: TSS and TC Infrastructure

Dr. Andreas U. Schmidt

Fraunhofer Institute for Secure Information Technology SIT,
Darmstadt, Germany

Literature

1. Thomas Winkler, IAIK Lectures TSS-TCG Software Stack, spring 07
2. Martin Pirker, IAIK Lectures Trusted Computing Infrastructure, spring 07

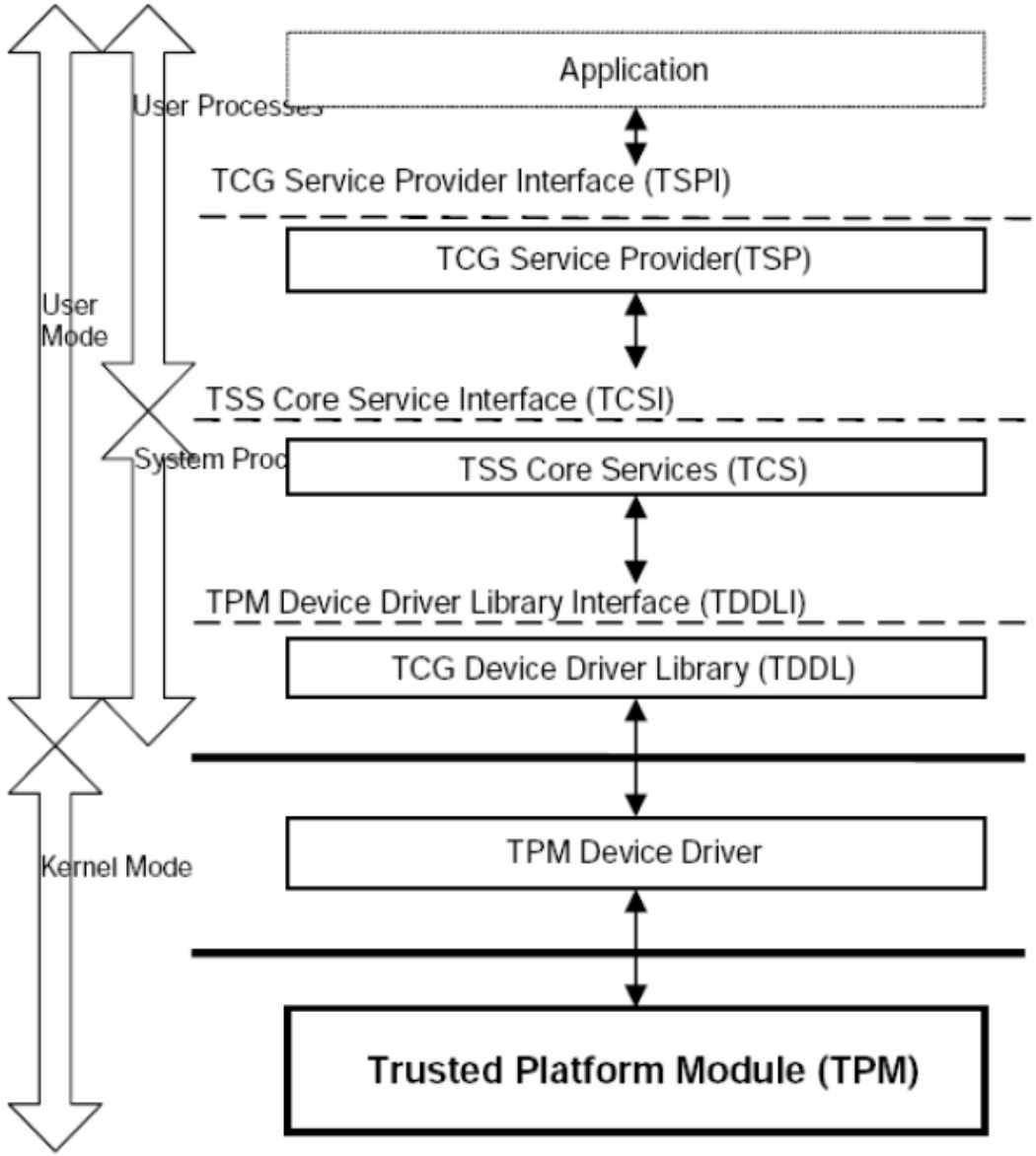
Introdicung TSS

TCG Software Stack (TSS) is the core software component for interaction with the TPM

- TSS design is provided and standardized by the TCG (TSS 1.2 spec is about 750 pages)
- TSS design goals
  - supply one single entry point to the TPM functionality (exclusive TPM access)
  - synchronize concurrent TPM access
  - TPM resource management (key slots, authorization sessions, ...)
  - building of TPM commands messages according to TPM specification
- TSS is designed as a stack of discreet modules with clearly defined interfaces between them

TSS Architecture

- TSS Service Provider (TSP)
  - op most module
  - standard API for applications
- TSS Core Services (TCS)
  - service (single instance per platform)
- TSS Device Driver Library (TDDL)
  - provides standard interface
- TPM device driver
  - kernel mode
  - TPM vendor or TIS
- TPM chip



TPM Access Linux vs. Vista

Linux Kernel drivers

- TPM drivers included in standard 2.6 kernels
- vendor specific drivers for 1.1 TPMs
  - Included in the Kernel: Infineon, Atmel, NatSemi
- 1.2 TPMs come with a generic interface (TIS – TPM Interface Specification)
- Kernel includes TIS driver that should work with all TIS compliant 1.2 TPMs
- TPM is accessed as a character device via /dev/tpmX
- very basic information is exported via SysFS (e.g. PCR contents)

previous to Windows Vista:

- vendor specific TPM device driver
- vendor specific TDDL and some (vendor supplied) TSS on top of it

Windows Vista:

- only supports 1.2 TPMs “out of the box”
- likely is using a TIS driver (yet unconfirmed)
- support for 1.1 TPMs (and maybe some 1.2 TPMs) has to be added
- by the TPM manufacturer via a driver
- Vista comes with a basic TPM abstraction layer called TPM Base Services (TBS)
  - RPC based service only accessible from the local machine

TDDL & TCS

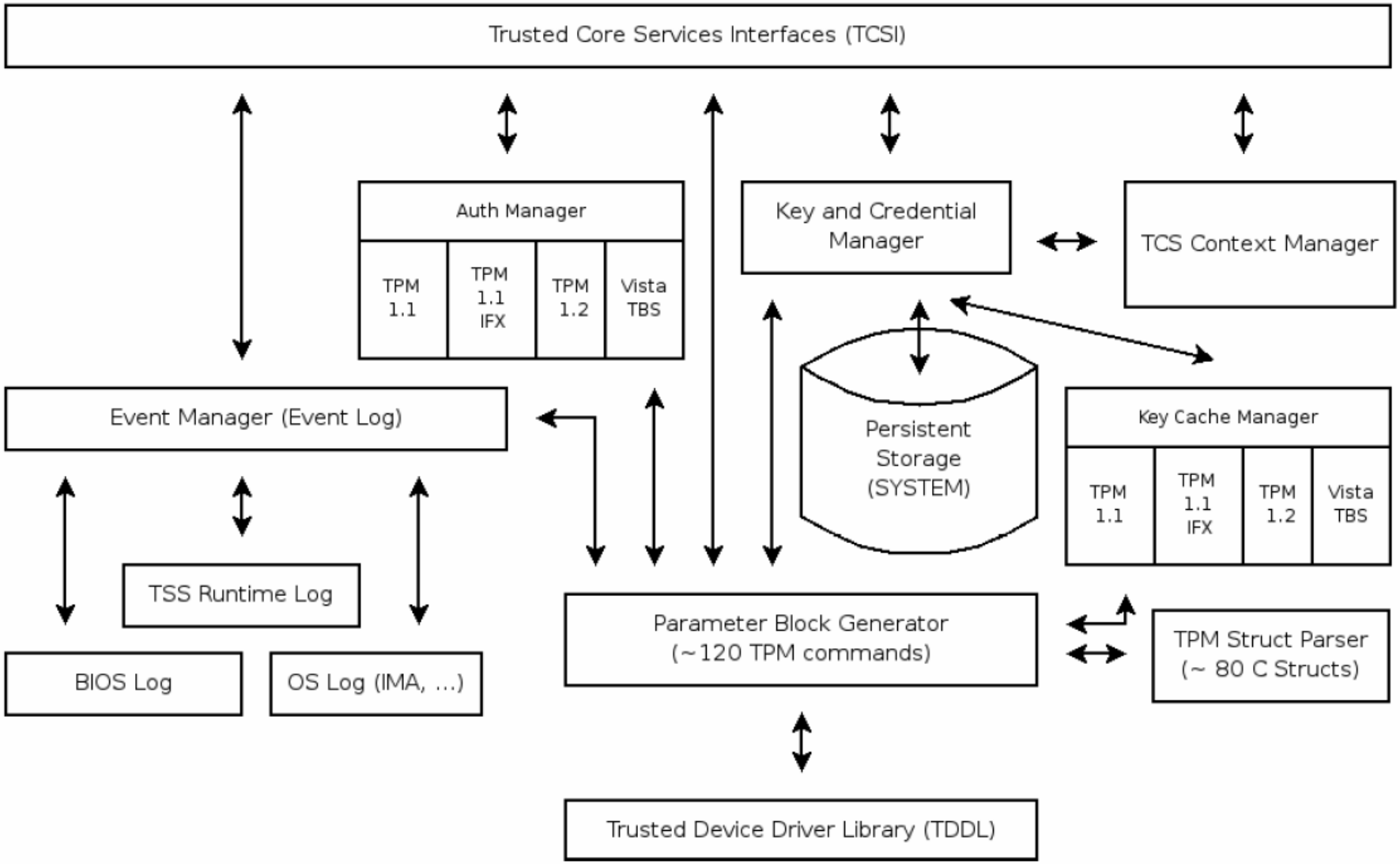
TCS – TSS Core Services

- TCS is a service provider (daemon or system service)
- one instance per system
- in TCG design, the TCS is the only entity directly accessing the TPM
- provides standardised functionality and a standard interface that is accessed by the TSS Service Provider(s)
- TCS is responsible for TPM command serialisation
- TCS builds the TPM command messages
- management of TPM resources

TDDL – TSS Device Driver Library

- first TSS component running in user space
  - standardised interface such that every TSS using the TDDL interface can communicate with the TPM regardless of the TPM manufacturer
- provides very simple abstraction layer for TPM access
  - open, close, transmit/receive
- TDDL is single-threaded (command serialisation has to be done in upper layers)
- interface between TDDL and device driver is vendor specific (at least for non-TInfraSpec compliant TPMs

TCS Architecture



TCS Functional Building Blocks 1/4

TCSI and TCS Context Management

- TCS Interface (TCSI)
  - simple C style interface
  - each operation is intended to be atomic
  - allows multi-threaded access
  - TCSI can be accessed remotely (RPC or standardized SOAP interface)
- all interaction with the TCS revolves around contexts
  - upper layers have to open a TCS context object before they can send commands to the TCS
- resources such as key handles or allocated memory belong to a context
- TCS contexts are managed by the TCS context manager

TCS Parameter Block Generator

- all commands actually send to the TPM pass through the PBG
- converts TCS function calls into byte stream oriented TPM command messages
- parses TPM response byte streams
- authorisation data (via HMAC) and command validation is not done in the TCS (typically done in TSP)

TCS Functional Building Blocks 2/4

Event Manager

- together with extending PCRs, users can add log entries to the PCR event log
  - main event log is managed by the TSS
    - events log entries are stored as TSS_PCR_EVENT entries
    - TCC_PCR_EVENT contains:
      - pcrIndex ... the PCR that was extended
      - pcrValue ... the value that was extended into the PCR
      - event ... description of the event
  - additional event log sources (not under control of TSS)
    - boot log (accessible via ACPI)
    - OS specific logs (IMA – Integrity Measurement Architecture for Linux; Kernel extension that measures loaded kernel modules, executed applications, ...)
- The event log does not need to be stored in shielded locations because tampering can be detected via the PCRs.

Event Log Sample (IMA)

Event Log verification (e.g. in attestation):

- Compare individual log entries with reference database
- Replay extend sequence on a (virtual) PCR
- Compare with actual PCR value
- Verify signature (TPM_Quote value)

	Measurement Value (fingerprint == SHA1)	Measurement Hook	File Name	
#000	9797EDF8D0EED36B1CF92547816051C8AF4E45EE	ima-init	boot-aggregate	Aggregate
#001	F7A0BF5A67CE98BC06316F77CA1F404A2D447534	mmap-file	init	Executable
#002	38C5D31E5DAD3F1B012FDD35B4E011E783CE6FD8	mmap-file	ld-2.3.2.so	Library
#003	42F79603219922016713BB8AAFC9E37F6936B226	mmap-file	libc-2.3.2.so	Library
#004	A4DC5EDF06698646CD76916F16E95C37E55DC12B	mmap-file	bash	Executable
#005	F4F6CB0ACC2F1BEE13D60330011DF926D24E5688	mmap-file	libtermcap.so.2.0.8	Library
#006	AE1BC1746AFD2AC1ECD1D9EEEAEBD125A6A9EB8D	mmap-file	libdl-2.3.2.so	Library
#007	CFBC7EC3302145AB78A307C0D41DBB9A4251377B	mmap-file	libnss_files-2.3.2.so	Library
#008	805572455CF5BF50A7EE42E3CC6B0EDA65AF17A4	mmap-file	initlog	Executable
#009	C95CBC5625719649103E0D1C3595967474842F7B	mmap-file	hostname	Executable
#010	0CAA342424F420FF29B7FB2FCF278F973600681B	mmap-file	mount	Executable
#011	5E45D898530F31BADEF5E247EBCF4AB57A795366	bash-source	functions	Bash Source
#012	A253AF3AB981711A13AE45D6B46462386E628076	mmap-file	consoletype	Executable
#013	2E37B839BC4EC1B6BE1BDF5BACD1E7B56567D8D9	bash-source	i18n	Bash Source
#014	C9D1B3E2CD0995E16AE6DD98B388FD873324740D	bash-source	init	Bash Source
#015	590F75EE97E0FC560F07FCB07A8646FADEC88C2A	mmap-file	uname	Executable
#016	5E851EFA4601B3AFCA9EAE75ED53688606630BFA	mmap-file	grep	Executable
#017	32798F58C4F1B4CD017B09BCAAF2A22D345E7E4F	mmap-file	sed	Executable
#018	CE516DE1DF0CD230F4A1D34EFC89491CAF3D50E4	mmap-file	libpcre.so.0.0.1	Library
#019	22EAF1B6009B23150367F465694AC63314866558	bash-script	setsysfont	Bash Comma
#020	8B15F3556E892176B03D775E590F8ADF9DA727C5	bash-script	unicode_start	Bash Comma
#021	A4C5F9D457DA16E47768423A68F135259F7180D7	mmap-file	kbd_mode	Executable
#022	497ED7F80C33AF25307DFC80970571C51006CE6A	mmap-file	dumpkeys	Executable
#023	04A0599405EBD306CEF2447679C8F4B5159A55C7	mmap-file	loadkeys	Executable
#024	AE327AD27D02BF2DE96557A1B4053D02129B1394	mmap-file	setfont	Executable
#025	7334B75FDF47213FF94708D2862978D0FF36D682	mmap-file	gzip	Executable
#026	93D65AB85CF5EE1ACD9E6BE5057D622D80AB5E10	mmap-file	dmesg	Executable
#027	B6E90C3A25B69C3B1D3B643DB7D9504FBC36C1D1	mmap-file	minilogd	Executable

TCS Functional Building Blocks 3/4

Key Management

- TPM keys are created (and used) inside the TPM, but do not survive power cycles (volatile memory) - to store such keys permanently, the TCS provides a persistent key storage
- keys managed by the TCS have to be assigned an identifier called UUID (Universally Unique Identifier)
- keys can be registered in persistent storage using this UUID
- special keys such as the SRK have a predefined UUID
- keys can be retrieved from the persistent storage using their UUID
- remember: To load a key into the TPM, its parent key has to be loaded previously. If the parent has not yet been loaded the TSS returns an error.
- keys remain in the persistent storage until the are unregistered

Key Cache

- loaded TPM keys are assigned a TPM keys handle
- TPM key slots are limited – key swapping is required
  - not to be mistaken with TPM unloading/reloading!
  - when swapping in a swapped-out key, the parent key secret does not have to be supplied (was already supplied when key was loaded)
  - swapped-out keys can only be loaded into the TPM of origin
  - swapped-out keys become invalid upon TPM power cycles
  - TPM 1.1:
    - optional command: TPM_SaveKeyContext / TPM_LoadKeyContext
    - TPM_EvictKey / TPM_LoadKey problems: re-supply parent secret; changed PCRs for PCR bound keys
  - TPM 1.2:
    - mandatory command: TPM_SaveContext / TPM_LoadContext
    - TCS maps TPM key handles to (stable) TCS key handles

TCS Functional Building Blocks 4/4

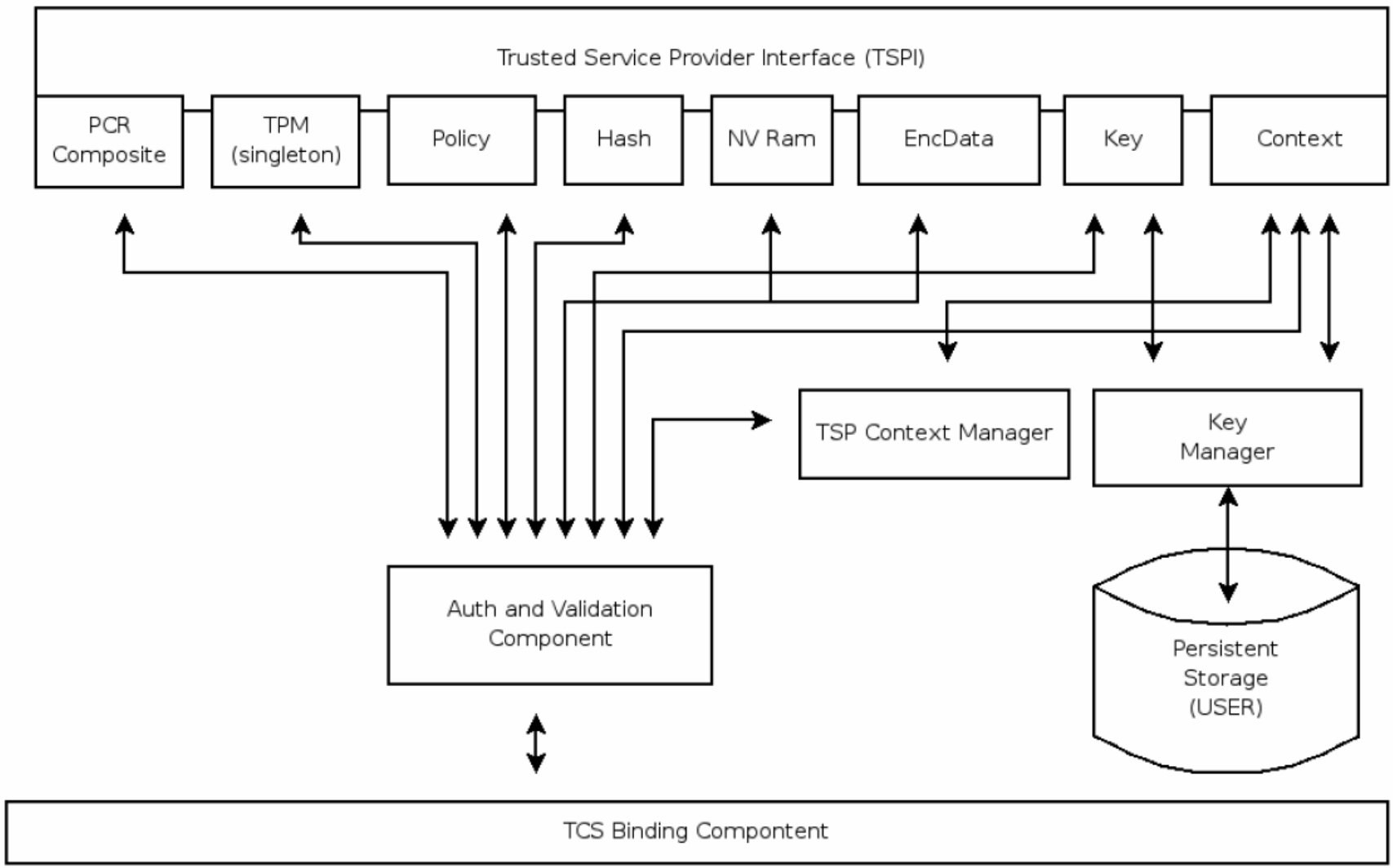
Authorisation Manager

- authorisation sessions (OIAP, OSAP) are referenced by TPM auth handles
- number of concurrently active auth sessions is limited
- auth session swapping is required
  - swapped-out auth sessions can only be loaded into the TPM of origin
  - swapped-out auth sessions become invalid upon TPM power cycles
  - TPM 1.1
    - optional command: TPM_SaveAuthContext / TPM_LoadAuthContext
    - only alternative: auth session termination
  - TPM 1.2
    - TPM_SaveContext / TPM_LoadContext
    - auth handles change when auth handles get re-loaded -> TCS has to maintain stability for upper layers

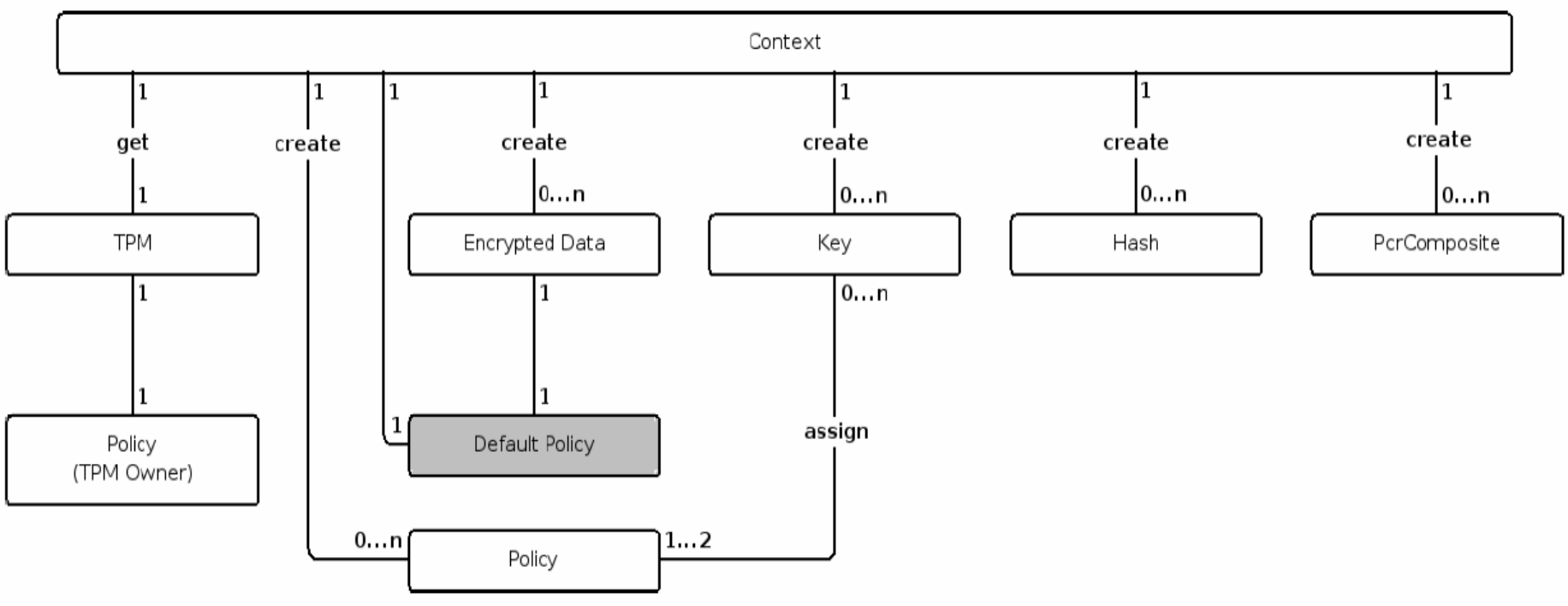
TSP – TSS Service Provider

- shared library linked to applications that require TPM access
  - application developers do not need to have in depth TPM knowledge
  - multiple instances per platform (in contrast to single-instance TCS)
- not only provides TPM access (via TCS) but also includes additional convenience functionality like signature verification
- TPM command authorisation and validation (initiating authorization sessions, ...)
- access to remote TCS via vendor specific mechanisms (RPC) or via standardised SOAP messages
- persistent user storage: persistent key store similar to persistent system storage provided by TCS but individual for every user
- provides a standardised C interface (TSPI)

TSP Architecture



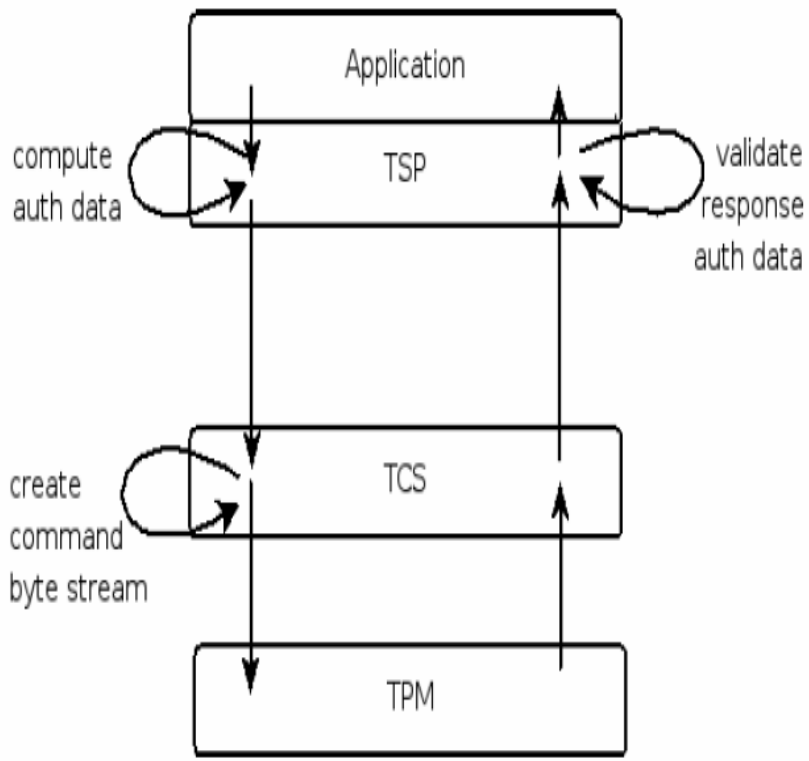
TSP Object Relationships



- TSP objects are created via the context object
- authorized objects are, by default, assigned to the default policy upon creation

TSP – TPM Command Authorisation

- For authorized entities, the TSP computes the authorization data.
- remember: authData is HMAC over parts of the input parameters, nonceEven, nonceOdd and contAuthSession; HMAC key is the entity secret (e.g. key usage secret)
- The command, together with the authData, is sent to the TCS. The PBG builds the command message and sends it to the TPM.
- Result message is sent to the TSP where the response is validated. again: HMAC over parts of the result, nonceEven, nonceOdd and contAuthSession; HMAC key is the entity secret.



TSP Context

- The TSP context object is the main entry point when interacting with the TPM
- holds basic information about environment configuration
- connection establishment to TCS
- allows access to the default policy
- provides memory management mechanisms (FreeMemory)
- allows to query the capabilities of the TCS implementation
- central point for registering and retrieving keys from the TSS' persistent storage (RegisterKey, LoadKeyByUUID, UnregisterKey)
- used to create all other TSP objects
  - TPM, Policy, Key, Hash, EncData, PcrComposite, NvRam
  - TSP objects are configured via init flags

TSP Context Java Code Samples

```
// create a context object
TcIContext context = new TcTssJniFactory().newContextObject();

// connect to TCS (null = localhost:30003)
context.connect(null);


// create other TSP objects
TcIRsaKey key = context.createRsaKeyObject(...); // init flags for key
    type, ...
TcIHash hash = context.createHashObject(TcTssDefines.TSS_HASH_SHA1);
TcIPcrComposite pcrComp = context.createPcrCompositeObject(0); // no init
    flags

// ...

// register key in system storage (parent SRK)
context.registerKey(key, TcTssDefines.TSS_PS_TYPE_SYSTEM, uuidKey,
TcTssDefines.TSS_PS_TYPE_SYSTEM, TcUuidFactory.getInstance().getUuidSRK());

// load key with given UUID from system storage
context.loadKeyByUuidFromSystem(uuidKey);
```

TSS Policy Object

- TPM entities such as keys or encrypted data require the knowledge of a usage secret
- at TSP level, these secrets are managed by the Policy object
- secrets can have a limited lifetime or a usage count
- one policy object can be assigned to multiple TSP objects
  - therefore all those objects use the same secret
  - changing the policy secret affects all assigned TSP objects
- the context object holds a default policy object
  - all new objects are assigned to this default policy upon creation
  - to set an individual secret for an object, create a new policy object and assign this policy to the object
  - remember: changing the secret of a policy affects all assigned objects!
- one exception: the TPM object is not assigned to the default policy upon creation but has an own policy
- default policy can be accessed via the GetDefaultPolicy method of the context
- policies of other authorised objects (keys, encData, ...) can be accessed via the GetPolicyObject function
- secrets of authorised objects can be changed using the ChangeAuth method

TSP TPM Object

- provides access to administrative TPM functions like
  - TakeOwnership/ClearOwnership
  - CollateIdentity/ActivateIdentity for AIK creation
  - querying TPM capabilities and manipulating TPM status
    - TPM version and manufacturer
    - number of PCRs provided by the TPM
    - ...
  - getting random numbers from the TPMs hardware RNG
  - PCR access (PcrExtend/PcrRead), event log access
  - Quote operation for attestation
- TPM object is assigned to one specific policy object (owner policy)
- implemented as singleton
- represents the owner of the TPM

Java Code Sample

```
// get TPM object
TcITpm tpm = context.getTpm();

// read TPM capability (number of PCRs)
TcBlobData subCap =
TcTssStructFactory.newBlobData().initUINT32((int)
    TcTssDefines.TSS_TPMCAP_PROP_PCR);
tpm.getCapability(TcTssDefines.TSS_TPMCAP_PROPERT
    Y, subCap);

// get 128 bytes of random data
TcBlobData randomData = tpm.getRandom(128);

// extend PCR 10 (without adding an event log
    entry)
TcBlobData data =
    TcTssStructFactory.newBlobData().initString("
    some arbitrary data");
tpm.pcrExtend(10, data.sha1(), null);

// read contents of PCR 10
TcBlobData pcrValue = tpm.pcrRead(10);
```

TSP Key Object

- TSP level representation of TPM keys
- assigned to policy objects handling key usage or migration secrets
- provides functionality to
  - create new TPM protected keys
    - key type and parameters are passed via a set of init flags
  - load/unload keys into/from TPM
  - certify TPM keys: provide evidence that a key actually is a TPM protected key
  - access to the raw TPM key blob (public key and parent-protected private key) via GetAttribData / SetAttribData functions

Java Code Sample

```
// setup storage key
TcIRsaKey storageKey =
context_.createRsaKeyObject(TcTssDefines.TSS_KEY_TYPE_
    STORAGE
    | TcTssDefines.TSS_KEY_SIZE_2048 |
TcTssDefines.TSS_KEY_NOT_MIGRATABLE);
storeKeyUsgPolicy_.assignToObject(storageKey);
storeKeyMigPolicy_.assignToObject(storageKey);

// create and load storage key
storageKey.createKey(srk_, null);
storageKey.loadKey(srk_);

// setup signing key
TcIRsaKey certifyKey =
    context_.createRsaKeyObject(TcTssDefines.TSS_KEY_S
    IZE_2048
    | TcTssDefines.TSS_KEY_TYPE_SIGNING);
signKeyUsgPolicy_.assignToObject(certifyKey);
signKeyMigPolicy_.assignToObject(certifyKey);

// create and load signing key
certifyKey.createKey(srk_, null);
certifyKey.loadKey(srk_);

// certify storage key using signing key
TcTssValidation validation =
    storageKey.certifyKey(certifyKey, null);
```

TSP PcrComposite Object

- TSP level object that allows to define a set of PCR values
- used to specify PCRs for e.g. CreateKey, Seal, ...
- SetPcrValue/GetPcrValue
  - PCR index, PCR value (can be current or “future” pcr value)
  - allows to set multiple PCRs (therefore “composite”)
- SelectPcr
  - used when not the PCR values are of interest but only the PCR indices (e.g. select set of PCRs for TPM Quote)
- GetCompositeHash
  - returns hash of PCR_COMPOSITE structure
  - composite hash is what is returned by TPM_Quote

TSP EncData Object

- TSP object for data encryption; 2 types: with or without PCRs
- without PCRs: Bind/Unbind
  - Bind: encrypt the given data blob using the public part of the key
  - Bind is a pure software (TSS) operation
  - Unbind requires the private key and therefore happens in the TPM
  - migratable vs. non-migratable binding keys
- with PCRs: Seal/Unseal
  - Seal: includes specified set of PCRs in encryption process
  - UnSeal: only releases the decrypted data if the specified set of PCRs matches the current PCR state
  - Seal/Unseal only works with non-migratable keys
- plain/encrypted data are set/retrieved using Get/SetAttribData
- input data length is limited by key size (TSS does no data blocking)

Bind / Unbind Java Code Samples

```
// create new binding key
TcIRsaKey key = context_.createRsaKeyObject(TcTssDefines.TSS_KEY_TYPE_BIND
    |
TcTssDefines.TSS_KEY_SIZE_2048 | TcTssDefines.TSS_KEY_NOT_MIGRATABLE);
keyUsgPolicy_.assignToObject(key);
keyMigPolicy_.assignToObject(key);
key.createKey(srk_, null);
key.loadKey(srk_);

// create encrypted data object
TcIEncData encData =
    context_.createEncDataObject(TcTssDefines.TSS_ENCDATA_BIND);

// bind data
TcBlobData rawData = TcTssStructFactory.newBlobData().initString("Hello
    World!");
encData.bind(key, rawData);

// get bound data
TcBlobData boundData =
    encData.getAttribData(TcTssDefines.TSS_TSPATTRIB_ENCDATA_BLOB,
TcTssDefines.TSS_TSPATTRIB_ENCDATABLOB_BLOB);

// unbind
TcBlobData unboundData = encData.unbind(key);
```

Seal / Unseal Java Code Samples

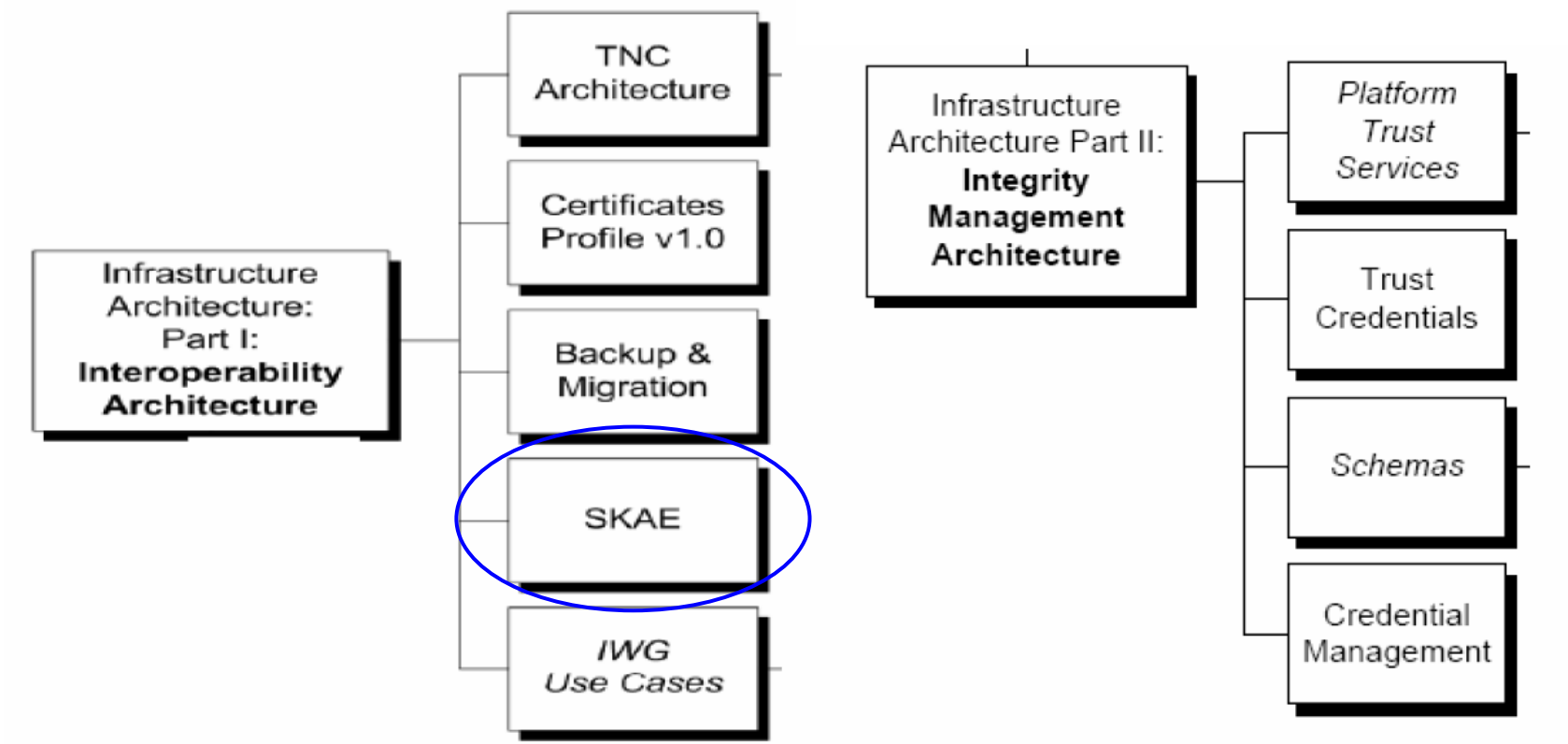
```
// create new key
TcIRsaKey key = context_.createRsaKeyObject(TcTssDefines.TSS_KEY_TYPE_STORAGE |
TcTssDefines.TSS_KEY_SIZE_2048);
keyUsgPolicy_.assignToObject(key);
keyMigPolicy_.assignToObject(key);
key.createKey(srk_, null);
key.loadKey(srk_);

// create sealed data object
TcIEncData encData = context_.createEncDataObject(TcTssDefines.TSS_ENCDATA_SEAL);

// set a secret for the sealed data
TcIPolicy encDataPolicy = context.createPolicyObject(TcTssDefines.TSS_POLICY_USAGE);
TcBlobData encDataSecret = TcTssStructFactory.newBlobData().initString("data secret");
encDataPolicy.setSecret(TcTssDefines.TSS_SECRET_MODE_PLAIN, encDataSecret);
encDataPolicy.assignToObject(encData);

// get PCR value of PCR 8
TcBlobData pcrValue = context_.getTpm().pcrRead(8);

// create PCR composite
TcIPcrComposite pcrs = context_.createPcrCompositeObject(0);
pcrs.setPcrValue(8, pcrValue);

// seal to current value of PCR 8
TcBlobData rawData = TcTssStructFactory.newBlobData().initString("Hello World!");
encData.seal(key, rawData, pcrs);

// get sealed data
TcBlobData sealedData = encData.getAttribData(TcTssDefines.TSS_TSPATTRIB_ENCDATA_BLOB,
TcTssDefines.TSS_TSPATTRIB_ENCDATABLOB_BLOB);

// unseal
TcBlobData unsealedData = encData.unseal(key);
```

TSP Hash Object

- TSP level hash object that allows to compute hash values of given data which can then be signed using TPM keys
- UpdateHashValue
  - updates the hash value with the provided data
- Set/GetHashValue
  - allows setting/retrieving the hash value represented by the object
- HashSign
  - signs the hash value held by the object using the provided TPM key
  - encryption with the private key inside the TPM
- VerifySignature
  - verifies the provided signature blob using the provided key
  - decrypts the signature blob using pub key and compares the result to the expected hash value provided via Set/GetHashValue

A glimpse on TC Infrastructure

- TCG Infrastructure WG concerns itself with with the interoperability of systems containing TCG technology (not only genuine TPs)



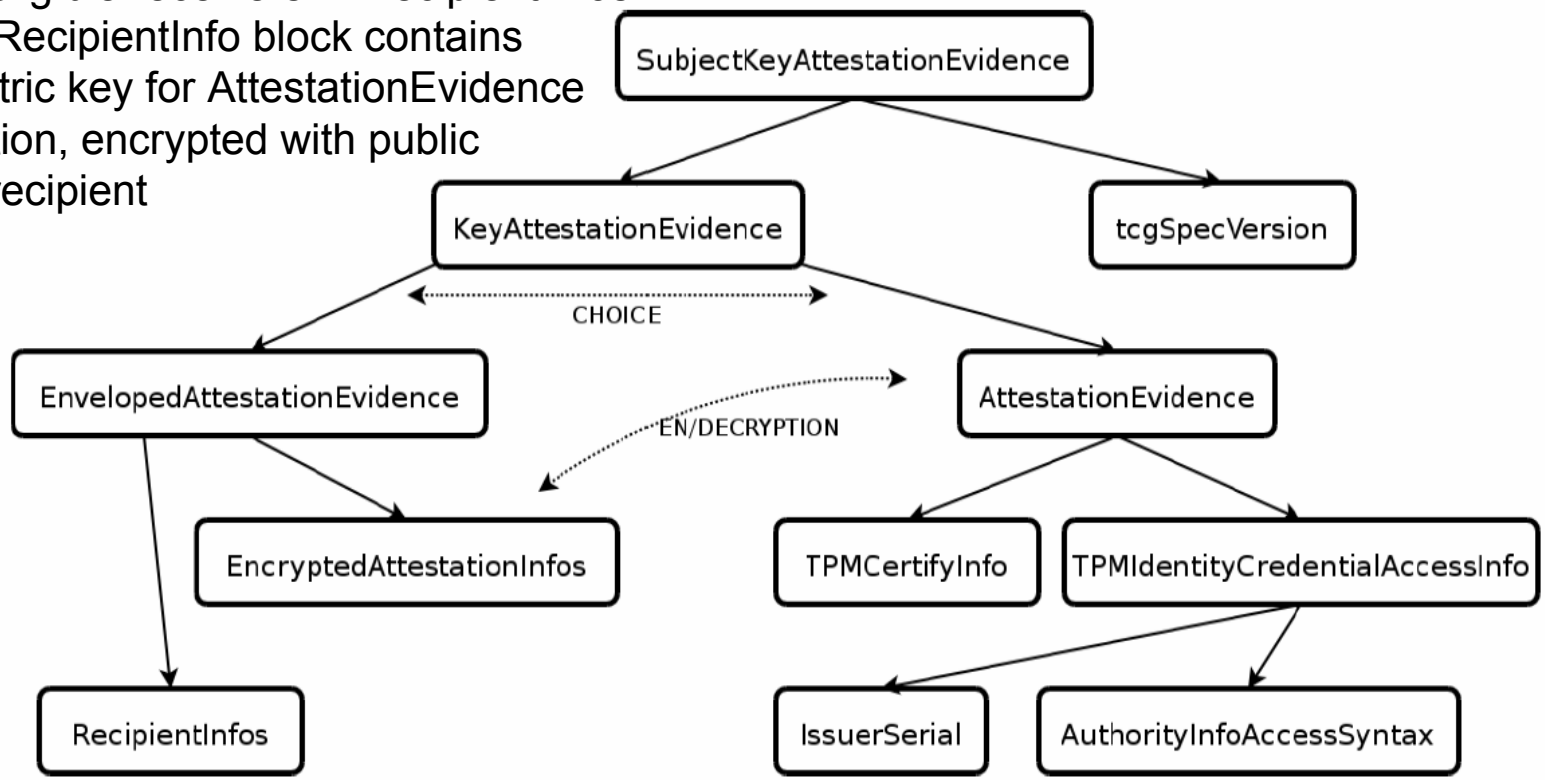
SKAE – Subject Key Attestation Evidence

- so far...
  - EK certificate proofs for hardware TPM
  - AIK certificate derived from EK certificate
- real life application?
  - nobody knows about these new certificate types
  - how to bring TCG oriented security assertions to common certificates?
- one approach: Subject Key Attestation Evidence extension
  - take standard certificate
  - add new certificate extension
  - extension contains proof that public key of certificate has corresponding private key stored in the protected storage area of a TPM

SKAE ASN.1 structure

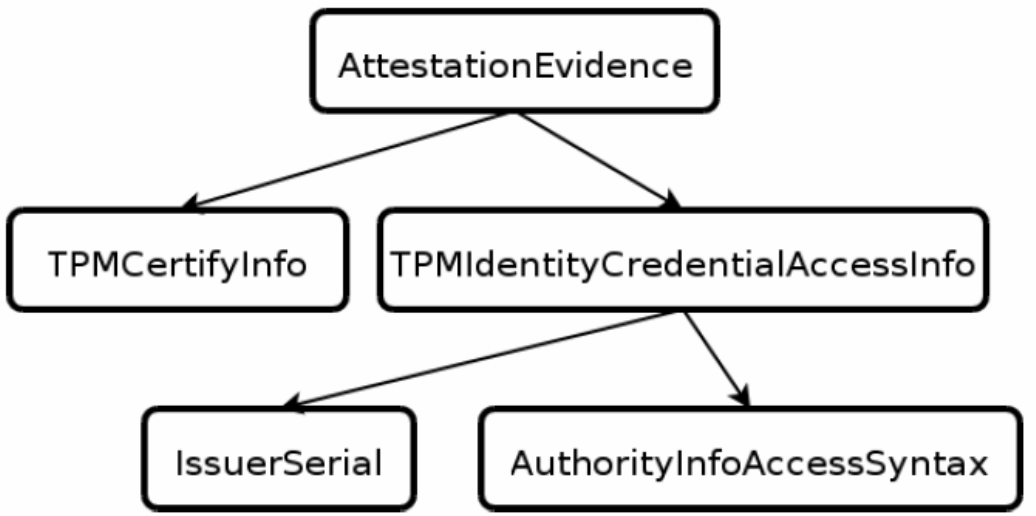
SKAE extension comes in one of two variants

- clear text
  - everyone can read AttestationEvidence contents
- encrypted
  - list of eligible receivers in RecipientInfos
  - every RecipientInfo block contains symmetric key for AttestationEvidence decryption, encrypted with public key of recipient



SKAE contents

- content of attestation evidence
  - TPMCertifyInfo: TPM key structure + signature over structure with AIK private key
  - IssuerSerial (optional component): reference to issuing authority and serial number of AIK credential
  - AuthorityInfoAccessSyntax: information how to access authority of AIK credential



SKAE creation & validation

- creating a certificate with SKAE extension
  - create TPM identity key “A”
  - obtain AIK certificate from Privacy CA for “A”
  - create new (non-migrateable) TPM key “B”
  - certify “B” with AIK key (TSS function Tspi_Key_CertifyKey)
    - result: certifyInfo structure
  - pre-assemble SKAE on client side or send all parts to CA
  - CA validates AIK certificate
  - CA validates certifyInfo structure
  - CA adds SKAE to normal certificate
  - CA builds / signs certificate
- validation steps
  - client offers certificate with SKAE extension
  - client offers proof of possession of private key, e.g. fresh signature on provided nonce
  - server validates proof of possession signature
  - server validates certificate with SKAE extension
  - server retrieves and validates AIK certificate referenced in SKAE
  - server validates certifyInfo structure in SKAE
  - if all tests ok, server is convinced that client has TPM and key in certificate with SKAE is protected by TPM

SKAE Deployment Scenario

- "old" infrastructure does not know about SKAE
- if "normal" certificate requires all extensions to be marked "critical“ (=all extensions must be known and how to read and interpret their value) then SKAE cannot always be included
- CA operation modes
  - CA includes SKAE only after successful validation of SKAE
    - must mention this in policy
  - CA does not validate SKAE at creation, just includes extension "as is"
    - SKAE validation later done by Relying party
  - CA validates SKAE, issues certificate without SKAE
    - must mention this in policy
  - certificate and SKAE always delivered in 2 separate pieces
    - trustworthy out-of-band distribution method needed, Relying party validates later

SKAE security/privacy impact

- after certification of key “B” there is no need to keep AIK private key
  - however, AIK certificate is long-term document
- SKAE contains reference to AIK
  - correlation SKAE <--> AIK <--> EK <--> TPM maybe possible
- options
  - always use one new AIK to create one new SKAE
    - maximum decoupling
  - design for trusted verifiers
    - only they can decrypt SKAE
    - need to be specified at build time
  - omit optional IssuerSerial reference
    - find trusted verifier using non-specified out-of-band mechanism