IAIK Java TCG Software Stack

jTSS API Tutorial

Institute for Applied Information Processing and Communications (IAIK)

Graz University of Technology
Inffeldgasse 16b
A-8010 Graz

http://www.iaik.tugraz.at

http://trustedjava.sf.net

The work presented in this Tutorial has been supported by the European Commission's research projects open_tc (FP6) andSecricom (FP7) and the Austrian FFG research project acTvSM.

This Tutorial has been compiled by Robert Stögbuchner and Ronald Tögl based on material created by Ronald Tögl, Thomas Winkler, Martin Pirker and Michael Steurer.

Contents

1. Introduction

Java is a platform with integrated security features and therefore well suited for Trusted Computing (TC) software. However, the current releases of Java do not provide support for the TC functionality which is available in today's hardware platforms equipped with a Trusted Platform Module (TPM).

This document introduces the jTSS API. The API provides TPM access within the Java environment and allows developers to make use of the concepts described by the Trusted Computing Group (TCG).

The presented software bases on the Java2 Standard Edition (J2SE) Desktop-PC system architecture and is available for download from [trustedJ] under an open source license. Please refer to the documentation available there on how to configure jTSS on your system.

1.1. About this Document

This tutorial presents the Java API and offers some general information about the TCG Software Stack (TSS) and its implementation in java. Chapter 2 provides a detailed view of the jTSS structure and the possible connections between the parts of a TSS. In Chapter 3 we are looking at the core parts of the jTSS API and their input/output data formats. Chapter 4 provides source code examples and further information for using the functionality of a TPM in a java application. Finally, Chapter 5 will point you to further resources. When you have completed this tutorial you will know how to use the jTSS API to access the TPM functions via the TSS in your java application.

2. TSS Architectures and Design

The Trusted Computing Group (TCG) specifies the Trusted Platform Module (TPM) and the accompanying software infrastructure called TCG Software Stack (TSS). This system software defines interfaces to applications written in the C language. The goal of this work is to make the TSS available to Java developers in a consistent and object oriented way.

The Trusted Computing Group (TCG) designed the TSS as the default mechanism for applications to interact with the TPM. In addition to forwarding application requests to the TPM the TSS provides a number of other services such as concurrent TPM access or a persistent storage on the hard disk for cryptographic keys generated inside the TPM.

This Chapter shows the design and structure of the software layers between Java applications and the TPM. We provide an overview of the original TSS architecture and its functionality and address aspects of making TSS functionality available to Java. This section also contains a brief description of the individual components depicted in Figure 2 and Figure 4, starting at the hardware (TPM) level up to the Trusted Core Service (TCS) and the Trusted Service Provider (TSP).

2.1. The TCG Software Stack Architecture and jTSS

TPMs are required to provide protected capabilities and at the same time are designed as low cost devices. Due to their inexpensive nature, the internal resources and external interfaces are kept to a minimum. To nevertheless provide a certain level of usability, functionality and abstraction the TCG defines the TSS with different layers (see Figure 1). Functions that require protected capabilities are implemented in the TPM while non-sensitive features which do not require hardware protection are implemented in software. To allow a common access to this Trusted Computing functionality, these software components are combined into the TSS and offer a standardized interface. This way, the TCG intends to provide a standard environment for applications. In the TCG's specification documents you can find detailed information on the TPM [TPM_Spec] and the TSS [TSS_Spec].

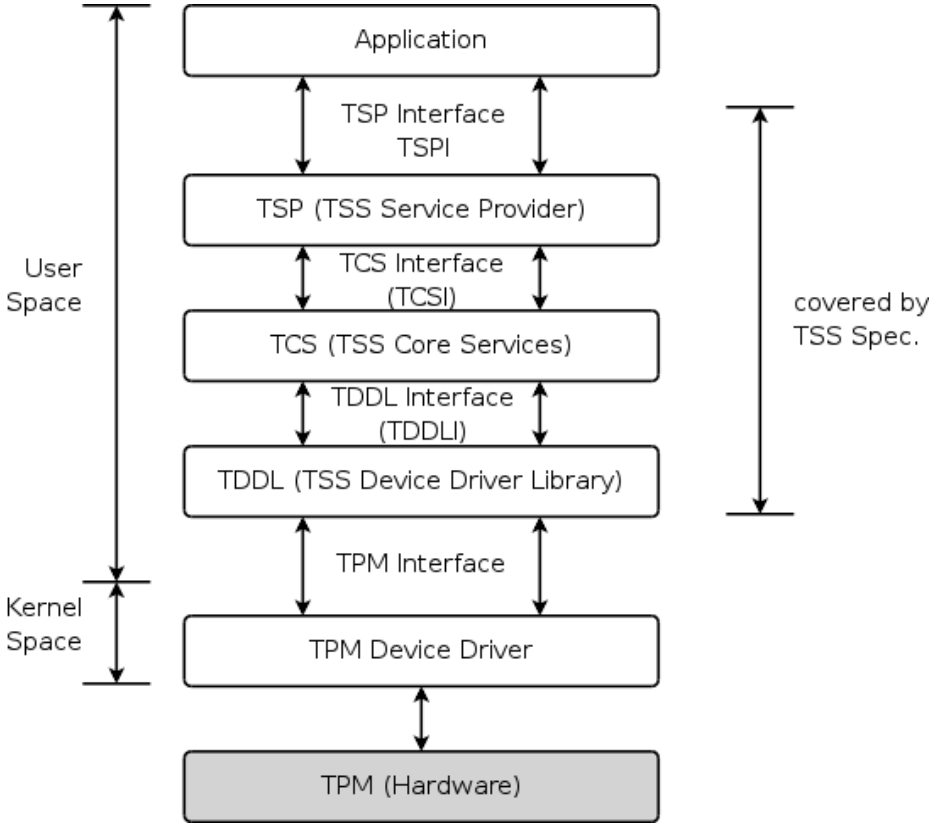


Figure 1: TCG Software Stack Layers

2.1.1 Trusted Platform Module (TPM)

In case of the PC platform, the hardware TPM is part of the mainboard and can not easily be removed or replaced without destroying the hardware. It is typically connected to the rest of the system via the LPC bus. The functionality of this hardware device resembles that of a smart card. A tamper resistant casing contains low-level blocks for asymmetric key cryptography, key generation, cryptographic hashing (SHA-1) and random number generation. With these components it is able to keep secret keys protected from any remote attacker. Additional high-level functionality consists of protected non-volatile storage, integrity collection, integrity reporting (attestation) and identity management. Of special interest is that the TPM is a passive device, a receiver of external commands. It does not measure system activity by itself but rather represents a trust anchor that cannot be forged or manipulated.

2.1.2 TPM Device Driver (TDD)

The TPM device driver resides in the Kernel space. For a 1.1b TPM this driver is vendor specific since it just offers a proprietary interface to upper layers whereas 1.2 TPMs support generic TPM Interface Specification (TIS) drivers. TIS provides a vendor independent interface to access TPM functionality. It depends on the platform and the operating system but the TDD may also support additional functionality such as power management. Nowadays, all major operation systems ship with TIS drivers or at least support them.

2.1.3 TSS Device Driver Library (TDDL)

The TDDL (Trusted Device Driver Library) resides in User space. From the user's point of view it exposes an OS and TPM independent set of functions that allow a basic interaction with the TPM. This includes sending commands as byte streams to the TPM and receiving the TPM's responses. The TCG specifies the TDDL Interface (TDDLI) as a required set of functions implemented in the TDDL. The intention was to offer a standardized TPM interface regardless of the TPM vendor and the accompanying TPM device driver. This ensures that different TSS implementations can communicate with any given TPM. In contrast, the communication between the TDDL and the TPM is vendor specific.

The TDDL is designed as a single-instance and single-threaded component.

The jTSS can operate on both major Operating Systems used today. The Linux OS implements the TDDL such that it opens the TPM device file *(/dev/tpm*)* provided by the underlying driver. Microsoft ships Windows Vista with a generic TIS driver that accesses the TPM via the so called TPM Base Services (TBS). For further details visit [MS_TBS]. This service interface should allow similar access to the TPM as the device file under Linux does.

2.2. jTSS Core Service (TCS)

The Trusted Core Service (TCS) is a system service and there is a single TCS instance for each TPM. The communication with the TPM relies on the TDDL and ensures that commands are properly serialized. The TCG defined the TCS Interface TCSI that specifies the communication between the TCS and the Trusted Service Provide TSP (see Figure 1).

To get an overview of the individual components that make up the TCS, Figure 2 presents the main components and their interactions.

2.2.1. Key and Credential Manager, Persistent System Storage

The TPM generates cryptographic keys but due to the low cost nature the internal memory (i.e. number of key slots) is limited. Nevertheless applications might need to store keys permanently. With the key management component of the TSS it is possible to store keys in a persistent storage (file system) outside the TPM encrypted under a parent key. To do so the user must provide this parent key before the TPM can create a new key pair. Before the TPM writes to the persistent storage it encrypts the new private key under its parent key to ensure that no unencrypted key leaves the TPM.

The root of the key hierarchy is the storage root key (SRK) which is generated at taking ownership and then stored inside the TPM permanently.

The Key Manager assigns a (possibly globally) unique identifier called UUID to every key and stores the key blobs in the persistent storage in the OS file system. I also retrieves and loads the keys. Applications can use this UUID as reference to the requested key.

Further, the TCS manages credentials (i.e. X.509 certificates) such as the endorsement or the platform credential. By that, it ensures that all applications can access these credentials by a well-defined mechanism. For example, the credentials are used as part of the CollateIdentity/ActivateIdentity cycle to create Attestation Identity Keys (AIKs).

2.2.2. Key Cache Management

To actually use a key inside the TPM it must be loaded in one of the TPM's key slots. The Key Cache Manager is responsible to transfer keys from the TPM to the persistent storage and vice versa. For example, if a user key resides in the persistent storage, the key cache manager will fetch it into the TPM the next time the user makes use of a child key. Since the key handle remains the same the entire process is transparent to the user.

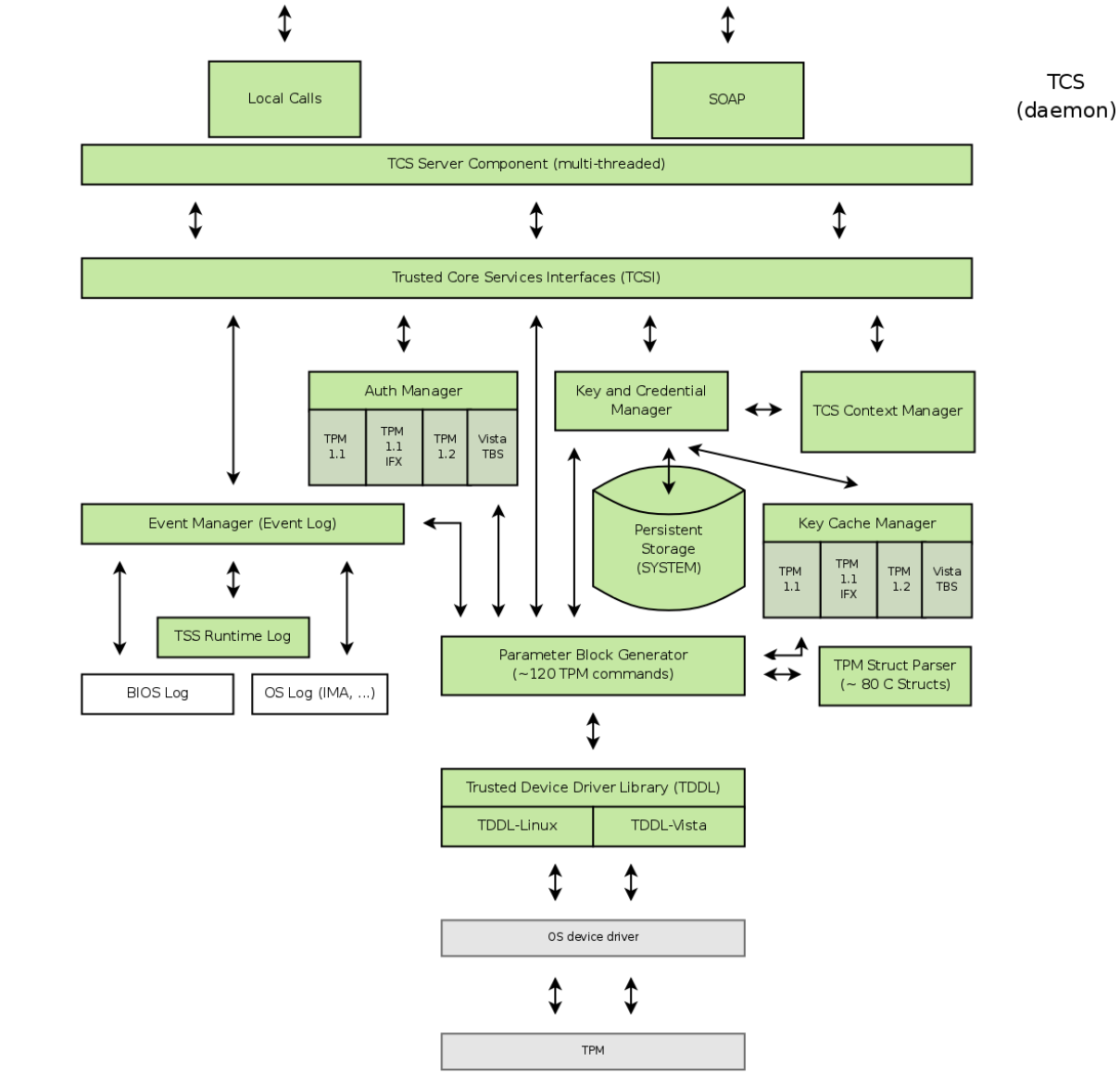


Figure 2: Trusted Core Services Components. Requests from the TSP are received via one of the communication facilities provided by the TCS server component.

2.2.3. TPM Command Generation

The TPM receives commands as byte streams. The Command Generator builds these byte streams and sends them to the TPM. It is also responsible for serialization of TPM commands in a proper way to avoid problems with concurrent accesses.

2.2.4. Parameter Block Generation and TPM Structure Parser

The TPM offers a byte stream oriented interface where TPM commands are marshaled into byte streams.

The TPM specification defines the data exchange with the TPM by C elements, so all primitive data types (e.g. UINT32) have to be mapped to equivalent Java types and the C structures have to be modeled by Java classes. To send a TPM command we extract the necessary content from the Java class instance and serialized it into a byte stream. Vice versa, we unpack the response and decode it to Java objects. Both, the TPM and JVM use big-endian.

2.2.5. Authorization Manager

The TPM is a low cost chip with a limited amount of internal resources. One of these resources are authorized sessions managed by the TSS. The authorization manager keeps track of them and swaps, evicts, or reloads TPM authorization sessions.

2.2.6. Event Manager and Event Log

The TCG specifies an event log message for the platform configuration register (PCR) extend operation. It is the task of the TCS to store, manage, and report this information.

In general, the event log is not limited to events generated by the TSS. During boot, the BIOS measures the start and stores all log information in the ACPI memory. To provide a complete event log, the TSS collects this and other log entries (e.g. Integrity Measurement Architecture [Sailer]) stored by the OS. Currently, only a basic memory based event log is implemented.

2.2.7. TCS Context Manager

Applications allocate resources with every call to the TSS. The Context Manager is responsible for these resources and binds them to an application specific context.

2.2.8. TCSI API

This layer represents the standardized API of the TCS. The TSS specification defines this API in a C style fashion which was mapped to Java in a straightforward way.

2.2.9. TCS Server Module

The TCS is designed as a system service or system daemon. The TCS server component allows multiple applications to concurrently access the TSS. It accepts incoming connection requests from applications and passes them to the underlying layer. It ensures proper synchronization and allows only one thread to enter critical sections of the TCS.

2.3. jTSS Communication Mechanism

The standard access to the TSS for applications is the TSP interface. Applications can directly link to the TSP library and use this interface to access the TCS. The TSP and the TCS can communicate either via local method calls or via the Simple Object Access Protocol (SOAP) interface. Local calls are mainly used for testing purposes whereas the SOAP communication covers a larger range of applications.

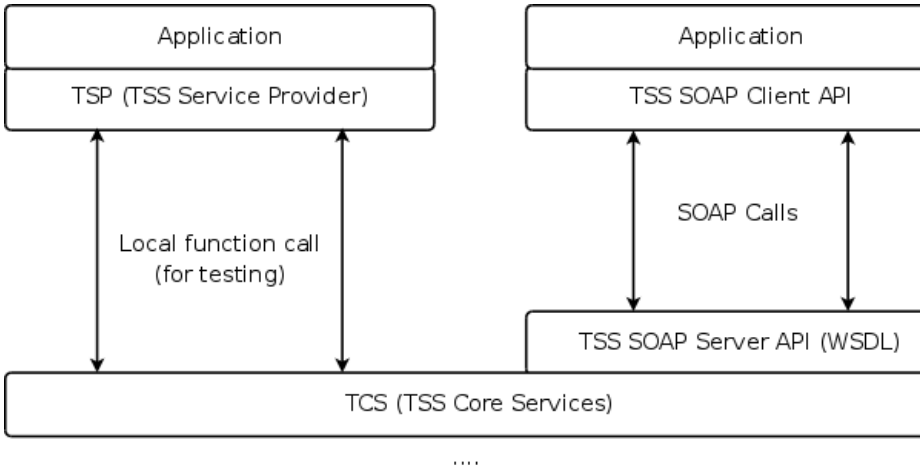


Figure 3: TSS Access methods: local and SOAP

2.3.1. Local

Local TCS method calls from TSP talking into account that admin or root privileges are needed to access the TPM driver directly.

2.3.2. Simple Object Access Protocol (SOAP)

SOAP is a lightweight XML based protocol for the exchange of information from a sender to a receiver [SOAP]. Within the context of this application, SOAP is used to build a framework for simple RPC calls. A client (i.e. the TSP) sends a request to a server (i.e. the TCS) and waits for a response message.

The above service-client architecture implies that we need a OS wide service that represents the TCS. We support the two major operating systems and therefore we actually need a Windows Service respectively a Linux Daemon for the server side. Both TCS representations have a predefined interface according to a WSDL file defined by the TCG. WSDL is the abbreviation for “Web Service Description Language” and specifies all remote method calls and their required parameters.

We can use the given WSDL file to derive the entire SOAP framework for the implementation of the communication. The problems with the SOAP encoding (see Section 2.1.6) forced us to

use the Axis SOAP libraries by the Apache Foundation. Axis is not maintained anymore and so this seems not to be the perfect solution.

- ***Linux Daemon and Windows Service***

  There exists several ways to start and maintain a Java application in Windows and Linux but for consistency reasons we rely on the Apache Daemon implementation for both operating systems. These libraries provide an easy to use and solid foundation for the TCS with additional functionality, e.g. Log server events.

- ***Implementation***

  The WDSL represents all methods and parameters for the communication between TCS and TSP. To actually use it within the TSP and TCS we can employ Apache Axis libraries and derive all necessary Java class files.

- ***Communication***

  We have already mentioned that there exists a local interface besides the SOAP interface for the communication between TSP and TCS. The TSP is linked to the Java application on the top of the stack and can therefore decide whether to access the TCS via local calls or via the SOAP interface. jTSS uses the standard TCP/IP connecting port 30004.

2.4. jTSS Service Provider (TSP)

The TSS Service provider (TSP) is the highest abstraction layer in the TSS and offers services defined by the TCG to applications. Due to the design as a system library, the TSP directly links to applications. For different applications several TSPs can coexists side by side and interact with one single TCS. Applications can access the TSP by a TCG defined TSP Interface (TSPI).

For the implementation, a context object serves as entry point to all functionality such as authorized and validated TPM commands, policy and key handling, data hashing, encryption, and PCR composition. The TSP can also be used to integrate the TPM in cryptographic libraries like PKCS#11. Figure 4 provides an overview of the major building blocks of the jTSS Trusted Service Provider. Starting at the bottom of the TSP, the major components are:

2.4.1. TCS Bindings

Since the TCS can support different communication mechanisms, the TSP must offer the same, too. The TCS binding component provides an abstraction layer that allows adding communication components later on without requiring changes of upper layers of the TSP.

2.4.2. Authorization and Validation Component

The access to the TPM and the TPM's resources requires proper authorization, such as a password. The TPM specification defines special authorization protocols used to establish authorization for those resources. The Authorization and Validation Component is responsible for creating and managing authorization sessions. Additionally, it validates response data from the TPM.

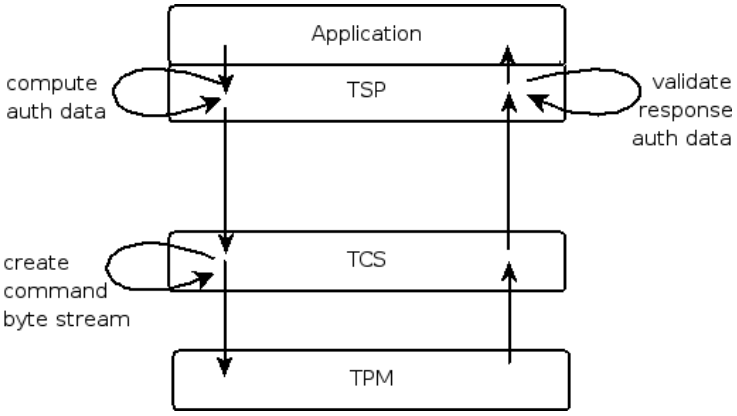


Figure 4; TPM Command Authorization

2.4.3. TSP Context Manager

Communication sessions with the TCS are mapped to contexts. The Context Manager is responsible to keep track of context sessions and the management of the associated resources.

2.4.4. Key Manager and Persistent User Storage

The TSP maintains a key hierarchy where every TPM key has a parent key in the layer above. Here, any storage key may wrap several other storage keys or keys of other types (for signatures, identity establishment, etc.). Before a user or an application can load a key from the user persistent storage the Key Manager establishes and verifies the entire key chain up to the Storage Root Key (SRK). Contrary to the system persistent storage, this TSP-layer storage is individual for every user of the system.

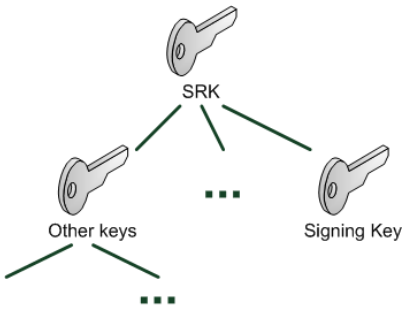


Figure 5: A simple key hierarchy.

2.4.5. TSPI and TSP Working Objects

The top level interface of the TSP is called Trusted Service Provider Interface (TSPI) and is standardized in the TSS specification.

3. The jTSS API

The Java programming language evolved in the last years to a commonly accepted environment. The main advantages are a restrictive type and memory safety ideally suited for security relevant applications. Although, Trusted Computing became more and more important over the last years there is currently no support within Java.

An Application Programming Interface (API) is an interface for programmers. The design of an API is always a tradeoff between a restriction of features to keep it simple and to preserve a developer's freedom. For the current purpose of the integration of TPM functionality in the Java language we have to meet a few additional requirements. The main design aspect is to cover the entire functionality envisioned by the TCG in this API without inconsistencies. Despite the complexity of the original specification we enable programmers to develop powerful applications with a clear and relatively easy to use interface.

Although, the basic concepts and functionality of the native TSP remains the same in its Java counterpart, several aspects were changed to meet the object oriented nature of Java. TSS entities such as contexts, keys, hashes, or the TPM are represented by actual Java objects. This relieves developers from object handles and memory management as required in the original TSP. Thus, developers who are already familiar with the TSPI should be able to use the Java interface without too steep learning curve because the design is similar to its C-based counterpart. The Java interface provides all the flexibility and features of the underlying stack to Java developers. Existing resources such as TSPI based C-code can therefore easily be mapped to Java.

With an abstract API, programmers can access the TSS with little knowledge of the actual implementation. This way, there should be no need to change Java applications if the underlying mechanism of the TSS changes.

3.1. Core Parts of the API

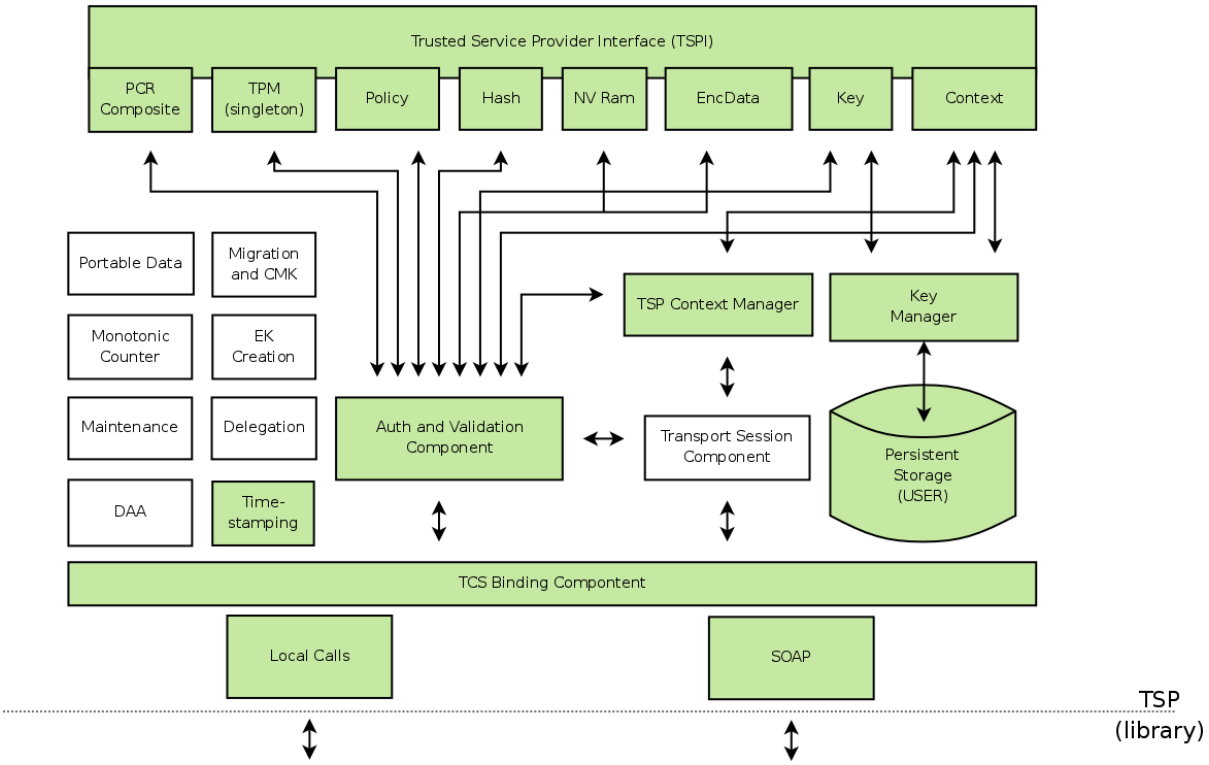


Figure 6: Trusted Service Provider Components

3.1.1. TclContext

A context represents a connection to the TSS Core Services. One can either connect to a local or a remote TCS. A context allows specifying the connection host. The context creates all further TSS objects like policy objects and registers, loads or unregisters keys from the persistent storage. The context can close objects (release their handles), get information (capabilities) about the TCS as well as free TSS memory.

3.1.2. TclTPM

This class represents the TPM and parts of its functionality. It provides methods to take or clear TPM ownership, read and set the TPM status, obtain random numbers from the TPM, access time stamping functions, or read and extend PCR registers. Aside from low level functions, *e.g.* trigger a TPM self test, it offers functions to create “attestation identities”. Further, it can do quote operations to attest the current state of the platform represented by the contents of the PCR registers.

3.1.3. TclRsaKey

Instances of this class represent keys in the TPM's key hierarchy. It provides functionality to create a new key, load a key into a key slot of the TPM, or certify keys.

3.1.4. TclEncData

This class provides access to the TPM's bind/unbind and seal/unseal functions which encrypt data with a TPM key. If this key is not migratable only the TPM that did the bind operation is able to unbind (*i.e.* decrypt) the data. It is computationally unfeasible to decrypt data if the TPM and therefore the according private key is not available any more. Sealing takes this concept a step further: This operation includes the platform configuration to encrypt data with a TPM key. By that, the sealed data can only be unsealed if the platform is in the state specified at seal time. The platform configuration is represented by the content of the TPM's PCRs.

3.1.5. TclHash

This class provides access to the TSS's hash algorithm SHA1. That includes unkeyed hash calculation and verification as well as keyed hash functions, *e.g.* create signatures of data blocks with a TPM key.

3.1.6. TclPcrComposite

The platform configuration registers (PCRs) can be used to attest the state of a platform (quote operation) or to seal data to a specific configuration. Instances of this class select one or more PCRs and hand them to the quote or seal functions.

3.1.7. TclPolicy

The policy class handles authorization data for TSS objects such as keys. The authorization data consists of the SHA-1 hash of the user password. Note that different character encodings (ASCII, UTF-16LE Unicode, etc.) will hash to different values. Alternatively to setting a password, a pop-up window will ask the user to enter the appropriate secret. UTF-16LE Unicode without a zero string termination should be used.

3.1.8. TclNvRam

This class stores the attributes of a region of non volatile RAM inside the TPM. It can be used for defining, releasing, reading or writing such a region. An example is the Endorsement Key certificate shipped with Infineon TPMs.

3.2. Input/Output Data Formats and Error Handling

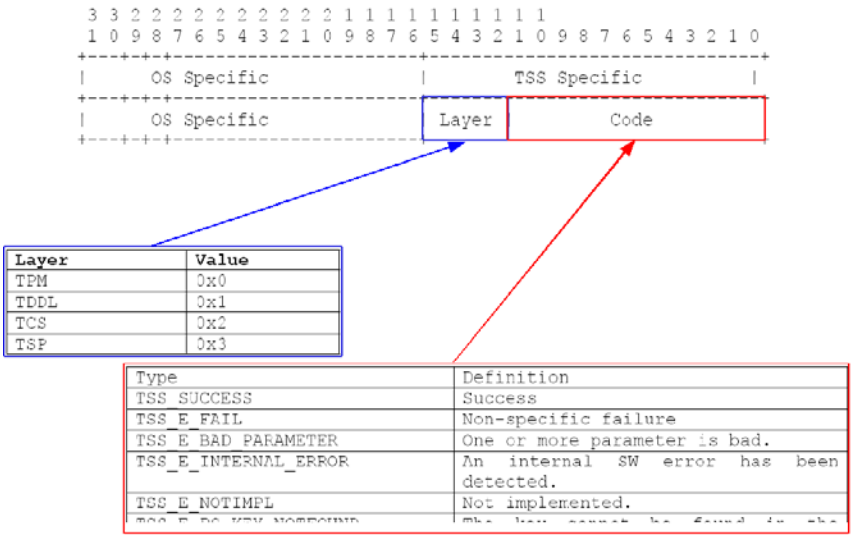
The C-based implementation of the TSP handles and identifies errors on the return code. Java replaces this concept with Exceptions which encapsulate the errors returned from the TSS. This allows handling TSS errors with conventional Java error handling mechanisms. Due to flexibility reasons, the TSS returns generic byte arrays instead of concrete structures. This allows developers to use one single function to retrieve different types of data and treat the returned data correctly. jTSS tries to convert the generic byte blobs into proper objects. If this is not possible some easy to use Java representations of the equivalent C structures are

available. They provide developers with a simple mechanism to convert byte blobs from lower layers into objects. Programmers can use this to handle structures directly received from the TPM.

The TSS return code consists of three components, OS code, TSS layer and TSS code.

- The OS code contains information regarding the OS-specific error. The details of this field are outside of the scope of the TSS specification.
- The TSS layer filed identifies the layer originating the error.
- TSS code identifies the specific cause or condition of the error.

jTSS will map these codes to `TcTSSException` class.



Layer	Value
TPM	0x0
TDDL	0x1
TCS	0x2
TSP	0x3

Type	Definition
TSS_SUCCESS	Success
TSS_E_FAIL	Non-specific failure
TSS_E_BAD_PARAMETER	One or more parameter is bad.
TSS_E_INTERNAL_ERROR	An internal SW error has been detected.
TSS_E_NOTIMPL	Not implemented.

Figure 7: TSS return code structure defined by [TSS_SpecA]

4. Code Examples

All of the example programs and source code lines given in this tutorial are not ready to be executed but can be downloaded in a version ready to be executed from [trustedJ]. In the *src* subdirectory you can find the entire source code of IAIK jTSS what contains the following tutorial code samples in the *iaik.tc.tss.tutorial package* in the *src/jtcc_tsp/src_tests/* subdirectory. For further information on jTSS Classes, their methods and variables take a look at the jTSS TSP java documentation [jTSS_TSP_JavaDoc].

4.1. Context

The Context class represents a context of a connection to the TSS Core Service (TCS) running on the local or a remote TCG system and is the central starting point for working with a TPM in jTSS. The focus of the Context object is:

- To provide a connection to a TSS Core Service. There might be multiple connections to the same or different core services.
- To provide functions for resource management and freeing of memory.
- To create working objects.
- To establish a default policy for working objects as well as a policy object for the TPM object representing the TPM owner.
- To provide functionality to access the persistent storage database.

The context object gets produced by the `TcTssContextFactory`. This factory provides the Context object. It automatically selects either *local* or *SOAP* binding, based on the configuration in the *jtss_tsp.ini* file.

The following example program shows the minimal necessary steps to work with the TPM. It creates a context object and connects it to the local TCS. With `closeContext()` it frees all the open recourses of this session and closes the connection.

```
import iaik.tc.tss.api.exceptions.common.TcTssException;
import iaik.tc.tss.api.tspi.TcIContext;
import iaik.tc.tss.api.tspi.TcTssContextFactory;

public class ShortestjTSSProgram {

  public static void shortestjTSSProgram() {

    try {
      TcIContext context = new
        TcTssContextFactory().newContextObject();

      context.connect();

      // work with context here …
```

```
      context.closeContext();

    } catch (TcTssException e) {
      e.printStackTrace();
    }
  }
}
```

TutorialContext.java

4.2. Reading PCR, TPM Flags and general TPM Information

The context also holds the TPM Object what provides a lot of additional general methods for the TPM. `getTpmObject()` is used to obtain a TPM object that allows interaction with the system's TPM. The `TcBlobDataI` class represents a data blob (binary data object) that is received from or passed to the TSS. Passing parameters as simple using byte arrays is a common practice in the TSS APIs. This class provides an abstraction of this byte data objects.

```
try {
  //get the context via the factory and connect to it
  // ...

  TcITpm tpm = context.getTpmObject();
  TcBlobData aesSupport = tpm.getCapability(
    TcTssConstants.TSS_TPMCAP_ALG,
    TcBlobData.newUINT32(TcTssConstants.TSS_ALG_AES));

  TcTssVersion tpmVersion = tpm.getCapabilityVersion(
    TcTssConstants.TSS_TPMCAP_VERSION, null);
  Log.info("TPM version : " + tcsVersion.toString());

  TcTssVersion tcsVersion = context_.getCapabilityVersion(
    TcTssConstants.TSS_TCSCAP_VERSION, null);
  Log.info("TCS version : " + tcsVersion.toString());

  TcTssVersion tspVersion = context_.getCapabilityVersion(
    TcTssConstants.TSS_TSPCAP_VERSION, null);
  Log.info("TSP version : " + tspVersion.toString());
} catch (TcTssException e) {
    Log.err(e);
}
```

TutorialReadTPMSettings.java

Here is an example how to find out the number of available PCRs and how to read their values and write them to a string. The TPM method `getCapability` provides all the capabilities of the TPM. `ORD`, `FLAG`, `ALG`, *etc.* are valid values for the `capArea` parameter. The only difference for `getCapabilityUINT32` is that the returned data is interpreted as `UINT32(long)`.

```
try {
  //get the context ...

  TcBlobData subCap = TcBlobData.newUINT32(
    (int)TcTssConstants.TSS_TPMCAP_PROP_PCR);
```

```
    // get the number of available PCRs from the TPM
    long numPcrs = tpm.getCapabilityUINT32(
      TcTssConstants.TSS_TPMCAP_PROPERTY, subCap);

    Vector<String> msgs = new Vector<String>();
    for (int i = 0; i < numPcrs; i++) {
      StringBuffer buffer = new StringBuffer();
      if (i < 10) {
        buffer.append(0 + i + ": ");
      } else {
        buffer.append(i + ": ");
      }

      //read the value of the PCR with index i
      TcBlobData pcrValue = tpm.pcrRead(i);
      buffer.append(pcrValue.toHexStringNoWrap());
      buffer.append(Utils.getNL());
      msgs.addElement(buffer.toString());
    }

  } catch (Exception e) {
    e.printStackTrace();
  }
```

TutorialReadPcr.java

Retrieve the name of the manufacturer from the TPM and the TPM version.

```
  //get the name of the manufacturer
  TcBlobData subCap = TcBlobData
    .newUINT32(TcTssConstants.TSS_TPMCAP_PROP_MANUFACTURER);

  TcBlobData tpmMan = context_.getTpmObject().getCapability(
    TcTssConstants.TSS_TPMCAP_PROPERTY, subCap);
```

TutorialReadTPMSettings.java

Read the TPM status flags. This flags are used to get and set the TPMs status about how the endorsement key was created (either using the method `TPM_CreateEndorsmentKey()` or using a manufacturers process), Indicates whether or not the operator authorization has been set, etc. For detailed information on this flags see [TSS_SpecA].

```
  //get TPM flags
  TcBlobData flags =
    tpm.getCapability(TcTssConstants.TSS_TPMCAP_FLAG, null);
```

TutorialReadTPMSettings.java

4.3. Creating and storing Keys

In this example **(TutorialKeys.java)** we will use the TPM to create an asymmetric key pair. The TPM will guarantee that it is only used for signing operations. All keys in the key hierarchy are children to the Storage Root Key (SRK). This key hierarchy needs to be maintained by the application. If a key should be not migratable, like the key in encryption example in Chapter 4.4, the `TcTssConstants.TSS_KEY_NOT_MIGRATEABLE` flag needs to

be set and with `keyMigPolicy.setSecret()` the key migration secret hast to be set to `TcTssConstants.TSS_WELL_KNOWN_SECRET`. The `TSS_WELL_KNOWN_SECRET` is defined as 20 bytes of zero.

First get the SRK object and set the canonical secret for its policy

```
try {
  //Set up the Storage Root KEY (SRK)
  TcIRsaKey srk = context
    .createRsaKeyObject(TcTssConstants.TSS_KEY_TSP_SRK);

  //set SRK policy
  TcIPolicy srkPolicy = context
    .createPolicyObject(TcTssConstants.TSS_POLICY_USAGE);

  srkPolicy.setSecret(TcTssConstants.TSS_SECRET_MODE_SHA1,
    TcBlobData
    .newByteArray(TcTssConstants.TSS_WELL_KNOWN_SECRET));

  srkPolicy.assignToObject(srk);
```

Create a migratable key for signing operations and configure its policy accordingly and define the authentication secrets for using and migrating the key.

```
  //create an empty Singing Key object and
  //define the security policy
  TcIRsaKey mySigningKey = context
    .createRsaKeyObject(TcTssConstants.TSS_KEY_SIZE_2048
      | TcTssConstatns.TSS_KEY_TYPE_SIGNING
      | TcTssConstatns.TSS_KEY_NON_VOLATILE
      | TcTssConstatns.TSS_KEY_MIGRATABLE
      | TcTssConstatns.TSS_KEY_AUTHORIZATION);

  //Define the object's authentication secrets
  TcBlobData keyUsageSecret = TcBlobData
    .newString("Password4Signatures");
  TcBlobData keyMigrationSecret = TcBlobData
    .newString("Password4KeyBackup");
```

Assign the policy to the key object.

```
  TcIPolicy keyUsgPolicy = context
    .createPolicyObject(TcTssConstants.TSS_POLICY_USAGE);
  TcIPolicy keyMigPolicy = context
    .createPolicyObject(TcTssConstatns.TSS_POLICY_MIGRATION);

  keyUsgPolicy.setSecret(TcTssConstants.TSS_SECRET_MODE_PLAIN,
    keyUsageSecret);
  keyMigPolicy.setSecret(TcTssConstants.TSS_SECRET_MODE_PLAIN,
    keyMigrationSecret);

  keyUsgPolicy.assignToObject(mySigningKey);
  keyMigPolicy.assignToObject(mySigningKey);
```

As all details of the key are set, instruct the TPM to create the RSA key pair by using its hardware random number generator (TRNG). So creating the RSA key is performed under the hardware protection of the TPM.

```
  mySignignKey.createKey(srk, null);
```

To store the key for later usage, create a unique identifier for the system-wide key database. Register the key there, indicating that it is under the protection of the SRK.

```
  //Store the created key on the HDD for later use
  TcTssUuid keyUUID1 = TcUuidFactory.getInstance()
    .generateRandomUuid();

  context.registerKey(mySignignKey,
    TcTssConstants.TSS_PS_TYPE_SYSTEM, keyUUID1,
    TcTssConstants.TSS_PS_TYPE_SYSTEM, TcUuidFactory
    .getInstance().getUuidSRK());

  Log.info("key1 registered in persistent system storage with "
    + keyUUID1.toString());
```

Load the RSA key into a key slot, that it can be used to perform signatures.

```
  //Load the key into a key slot of the TPM
  mySignignKey.loadKey(srk);

  //Now my Signing Key can be used for cryptographic operations...

  // create new hash object
  TcIHash hash = context.createHashObject(
    TcTssConstants.TSS_HASH_SHA1);

  // update
  TcBlobData data = TcBlobData.newString("Hello World");
  hash.updateHashValue(data);

  // sign
  TcBlobData signature = hash.sign(mySigningKey);

} catch (TcTssException e) {
  //...
}
```

4.4. Encrypting and decrypting Data

You can encrypt and decrypt Data using either *bind* or *seal* and using a storage key from the TPM's root of trust.

- Bind

  Bind data uses a storage key from the TPM's root of trust to encrypt (bind) data. Bound data can be encrypted and decrypted on different TPMs if a migratable key is used for encryption.

- Seal

  Sealing data is like binding it and recording the values of selected PCRs at time of encryption. Sealed data can only be decrypted if the selected PCRs have the same values as they had at the time of encryption. This

The following source code sniped gives an example sequence of the steps to perform using a TPM for binding and unbinding data.

```
//..first configure, create and load key..

//The data to bind to this TPM
TcBlobData rawData = TcBlobData.newString("Hello World!");

//create encryption data object
TcIEncData encData = context
  .createEncDataObject(TcTssConstants.TSS_ENCDATA_BIND);

//Perform encryption within the TPM
encData.bind(key, rawData);

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

//..
//store encrypted data somewhere
//..

TcIEncData remoteBoundData = context
  .createEncDataObject(TcTssConstants.TSS_ENCDATA_BIND);

remoteBoundData.setAttribData(
  TcTssConstants.TSS_TSPATTRIB_ENCDATA_BLOB,
  TcTssConstants.TSS_TSPATTRIB_ENCDATABLOB_BLOB, boundData);

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

TutorialEncryptData.java

5. Further Steps

After completing this tutorial you may wish to get more information about the jTSS functionalities. You can find a description of all the methods in the jTSS java documentation on [trustedJ]. Additional example programs can be downloaded from [trustedJ]. In the *src* subdirectory you can find the entire source code of IAIK jTSS what contains test case code samples in the *iaik.tc.tss.test package* of the *src/jtcc_tsp/src_tests/* subdirectories.

6. Resources

[trustedJ] M. Pirker, R. Toegl, T. Winkler, et al. Trusted Computing for the JavaTM Platform, http://trustedjava.sourceforge.net/. 2007.

[Sailer] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In Proceedings of the 13th USENIX Security Symposium, 223-238, 2004.

[SOAP] IBM. Simple Object Access Protocol (SOAP), http://publib.boulder.ibm.com/infocenter/radhelp/v6r0m1/index.jsp?topic=/com.ibm.etools.webservice.doc/concepts/csoap.html, 2005.

[SOAPENC] W3C Recommendation. SOAP Encoding, http://www.w3.org/TR/soap12-part2/#soapenc, 2007.

[WSI] WS-I Organization. Web Service Standards, http://www.ws-i.org/, 2008.

[SOAPIM] IBM. Discover SOAP encodings impact on Web service performance, http://www.ibm.com/developerworks/webservices/library/ws-soapenc/. 2003.

[AXIS2] Apache Software Foundation. Axis 2, http://ws.apache.org/axis2/, 2008.

[SOAPPRE] Jorgen Thelin, The Death of SOAP Encoding, http://www.thearchitect.co.uk/weblog/archives/2002/11/000008.html, 2002.

[jTSS_TSP_JD] T. Winkler, R. Toegl, et al., IAIK jTSS – TCG Software Stack for the Java™ Platform, http://trustedjava.sourceforge.net/jtss/javadoc_tsp/index.html, 2011.

[TPM_Spec] http://www.trustedcomputinggroup.org/developers/trusted_platform_module/specifications, 2011

[TSS_SpecA] http://www.trustedcomputinggroup.org/files/resource_files/6479CD77-1D09-3519-AD89EAD1BC8C97F0/TSS_1_2_Errata_A-final.pdf, 2007

[TSS_Spec] http://www.trustedcomputinggroup.org/developers/software_stack, 2011

[MS_TBS] http://msdn.microsoft.com/en-us/library/aa446796%28v=VS.85%29.aspx, 2011