

# Linux TCG Software Stack Low Level Design

Version 0.8

Kent Yoder  
Linux Technology Center

Document owner:	Kent E Yoder <yoder1@us.ibm.com>
-----------------	----------------------------------

Change Log		
Version	Date	Comments
0	05/13/04	Initial revision
0.1	05/14/04	Added TCS/TSP key management info
0.2	05/16/04	Table of Contents, TCS key management example, TCS misc functions
0.3	05/17/04	Formatting
0.4	05/28/04	Changes based on Tom Lendacky's review
0.5	06/01/04	Changes based on Emily Ratliff's review, typos and formatting
0.6	06/02/04	Changes based on Kylie Hall's comments, added design issues section, typos, clarification on key caching
0.7	06/04/04	Changes based on Emily Ratliff's 2nd Review
0.8	07/20/04	Changes based on review by the Security gurus

Reviewers		
Name	Required/Optional	
Emily Ratliff	R	LTC Security Team TPM project lead
Tom Lendacky	R	LTC Security Team
Doc Shankar	O	LTC Security Team
Danna Rother	O	Manager, LTC Security Team
Tony Abbatista	O	LTC Project Manager
Steve Bade	O	eServer Trusted Computing Architect
Lee Terrell	O	eServer /AIX Security Architecture and Design
Ravi Shankar	O	AIX Security Architect
Helmut Weber	O	eServer Platform Design & Architecture
Rich Guski	O	zSeries SW Security Architecture

## 1.0 Introduction

### 1.1 Overall Design Considerations

#### 1.1.0 Threading Model

#### 1.1.1 Globalization / Natural Language Support

#### 1.1.2 Performance

#### 1.1.3 Compatibility

#### 1.1.4 Installation

#### 1.1.5 Serviceability

#### 1.1.6 Build

#### 1.1.7 Dependencies

#### 1.1.8 Testing Considerations

#### 1.1.9 Documentation

#### 1.1.10 Design Problems

#### <<1.1.11 Permissions>>

## 2.0 TSS Service Provider

### 2.1 Object Management

#### 2.1.0 Data Structures

#### 2.1.1 Object Creation Functions

#### 2.1.2 Object Support Functions

### 2.2 Key Cache Management

#### 2.2.0 Data Structures

#### 2.2.1 Object Creation Functions

#### 2.2.2 Support Functions

### 2.3 Cryptographic Services

#### 2.3.0 Functions

### 2.4 Graphical User Interface

#### 2.4.0 Functions

### 2.5 Memory Management

#### 2.5.0 Data Structures

#### 2.5.1 Functions

### 2.6 Persistent Storage

#### 2.6.0 Functions

#### 2.6.1 Key Registration Functions

### 2.7 TCS Calling Interface

### 2.8 Utilities

## 3.0 TSS Core Services

### 3.1 TCS Calling Interface

#### 3.1.0 Data Structures

#### 3.1.1 Functions

### 3.2 System Persistent Storage

#### 3.2.0 Functions

### 3.3 TCS Context Handling

#### 3.3.0 Data Structures

- 3.3.1 Functions
- 3.4 Event Handling
  - 3.4.0 Data Structures
  - 3.4.1 Functions
- 3.5 Key Cache Management
  - 3.5.0 Data Structures
  - 3.5.1 Functions
- 3.6 Miscellaneous
  - 3.6.0 Functions
- <<4.0 Portability
- 5.0 References>>

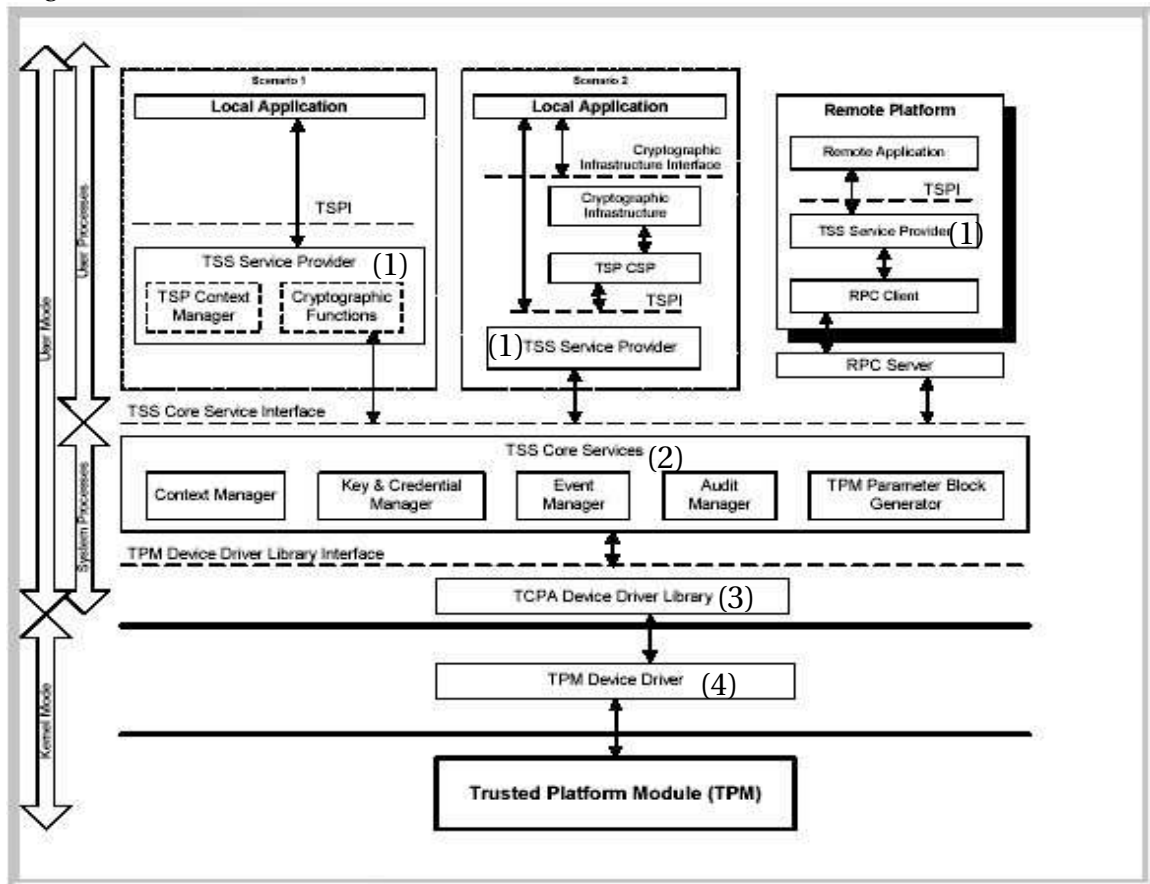
# 1.0 Introduction

The TCG Software Stack (TSS) is the set of software components that supports an application's use of a platform's TPM. The TSS is composed of a set of software modules and components that allow applications to communicate with a TPM in several different ways.

The primary design goals of the TSS are (according to the TSS spec v1.1):

- Supply one entry point for applications to the TPM functionality  
<<-Provided by the TSS Service Provider Interface>>
- Provide synchronized access to the TPM  
<<-Provided by the TPM Request Manager (see section 3.1)>>
- Hide issues such as byte ordering and alignment from the application  
<<-Provided internally by the TSS>>
- Manage TPM resources  
<<-Provided by the TSS Core Services Daemon>>

The TSS is divided into modules which are intended to be independent subsystems which communicate through interfaces defined in the TSS specification. Communication between components inside each module is meant to be implementation specific and will not affect callers of an API. The modules and components are represented in the following diagram:



<<Figure 1.0: Interaction of the pieces of the TSS Software Stack. (1) TSS Service Provider, (2)

TSS Core Services, (3) TCPA Device Driver Library, (4) TPM Device Driver>>

The modules that make up the TSS are the (1) TSS Service Provider (TSP) <<(see section 2.0)>>, the (2) TSS Core Services (TCS) <<(see section 3.0)>>, the (3) TCPA Device Driver Library (TDDL) and the (4) TPM Device Driver. <<The three pieces this low-level design will cover are the TSS Service Provider shared library, the TSS Core Services daemon and the TCPA Device Driver Library. The TCPA Device Driver Library is implemented by this TSS, but contains no additional functions other than those specified in the TSS 1.1 API. The TPM device driver itself will be maintained separately from this TSS and therefore will not be discussed in this low-level design. The following is a diagram of the interaction of the three pieces of the TSS at a high level:

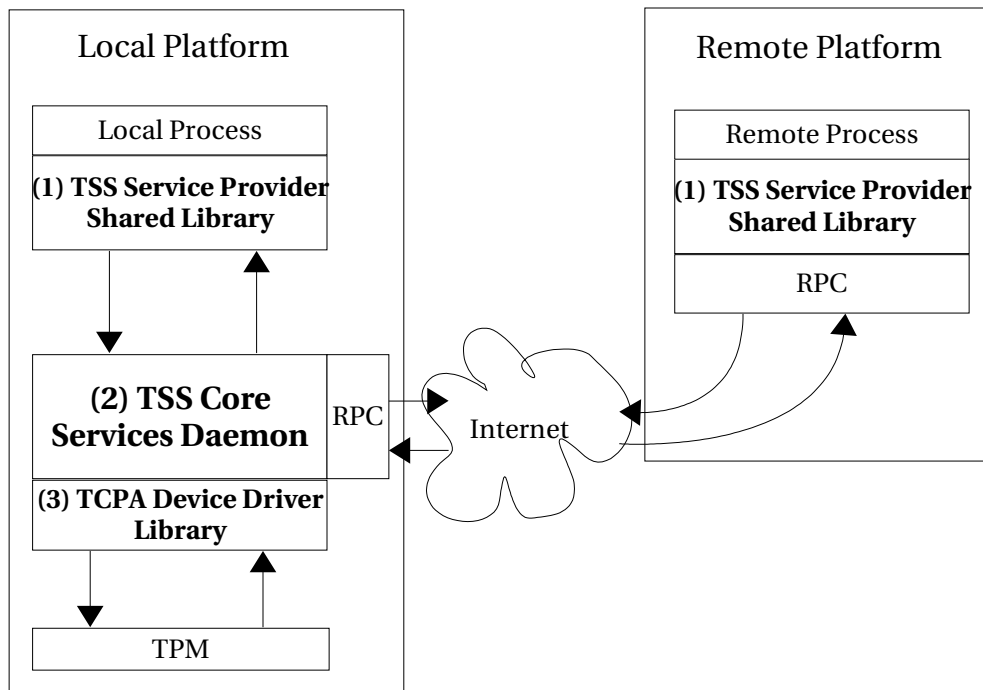


Figure 1.1: Interaction of the pieces of the TSS Software Stack implemented by functions in this document.(1) TSS Service Provider, (2) TSS Core Services, (3) TCPA Device Driver Library.

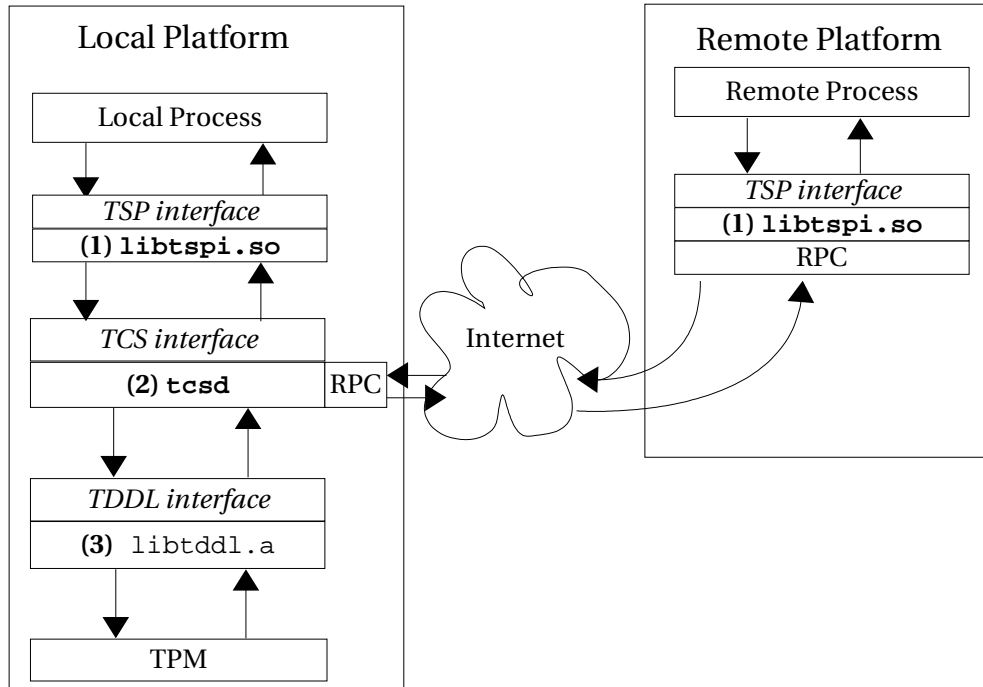


Figure 1.2: Interface interactions between the pieces of the TSS. (1) TSS Service Provider, (2) TSS Core Services, (3) TCPA Device Driver Library.

>>

The current high level design of the Linux TCG Software Stack (HLD-[TrustedComputingReferenceImplementationv21.doc](#)) documents the changes that would be made from the original PCD code base in section 4 of that document, so some detail which is not included here can be found there. Also, please refer to the TSS 1.1 specification as a companion document to this low level design.

## 1.1 Overall Design Considerations

The following sections describe details of the TSS project's overall design.

### 1.1.0 Threading Model

In keeping with the high level design document mentioned in section 1.0, the threading library used will be the POSIX pthreads library. pthreads will be used in both the TSP shared object and the TCS daemon. In the TSP, pthread mutexes will protect shared data structures from access by multiple threads in the user's application. In the TCS daemon, a new thread will be spawned when a request arrives so that the daemon can continue listening for new requests while the previous one is being processed. pthreads routines will be used to handle these threads and protect any shared data structures.

<<The maximum number of threads allowed to be running simultaneously in the TCS daemon will be configurable at build time and at run time through a configuration file. The

TCS daemon will re-read its configuration file when sent the SIGHUP signal, allowing the parameter to be changed without restarting the TCS daemon.>>

### 1.1.1 Globalization / Natural Language Support

NLS support will be enabled through the gettext package. Translation catalogs for languages other than English will not be provided, but will be accepted from the community.

### 1.1.2 Performance

The LTC-TSS can be optionally configured with `--enable-perf` to link to gprof libraries for profiling. Once test case development is complete, the test suite will be run with profiling of the TSS enabled. At this point, performance related issues will be identified. The performance of all internal routines will be considered at this time and solutions to any performance problems will be made prior to release.

### 1.1.3 Compatibility

Compatibility with other TSS stacks will be given all possible consideration. Based on proposals by the TSS Working Group, the design/implementation of the LTC-TSS may need to be changed to ensure compatibility. Any revisions of the TSS 1.1 spec by the TSSWG will be integrated into the LTC-TSS and any TSS 1.2 spec design considerations that can be applied to this 1.1 based stack will be implemented. For instance, it is expected that the RPC design accepted by the TSSWG for the TSS 1.2 spec will be widely implemented as the RPC design for most 1.1 TSS's, despite the fact that there is no RPC design specified in the 1.1 spec. For compatibility, the 1.2 RPC design will be implemented in this TSS. If the 1.2 design cannot be approved in time for the release of this TSS, the implementation will be based on the current standing TSS 1.2 draft.

### 1.1.4 Installation

Installation of the TSP shared library will be to the `/usr/lib` directory, with the `tcscd` executable residing in `/usr/sbin`. Files specific to a distribution will be installed wherever appropriate, for instance the NLS files and `tcscd` startup script locations may vary. These variations will be reflected in the per-distribution RPM's, which will be available for all supported distributions (RedHat Enterprise Linux and SuSE Linux Enterprise Server). Separate RPM's will be available providing the documentation for this package, as is customary with SuSE and RedHat packages.

<<Although the specific locations of the files to be included in the TSS are listed in this low-level design, alternate locations are certainly acceptable. No design decisions will be made which will necessitate specific locations for any files.>>

### 1.1.5 Serviceability

Message logging will be configurable through `syslogd` for all log levels that `syslogd` supports. Separate binary RPM's will be provided with and without debug level messages, so that a configuration error (turning debug level messages on) cannot expose sensitive data. No sensitive data will be logged to any level other than debug.

Once the LTC-TSS is released by IBM externally (presumably on developerWorks or



sourceforge.net), support channels will be set up through that particular website. Users will be able to post bugs, join mailing lists, supply patches and download updates through these channels. Support will be given to anyone through these channels, however support contracts specific to IBM products will also be made available as resources allow.

### 1.1.6 Build

An automake and autoconf build environment will be provided as part of a CVS or tarball source code release. Full documentation on the build itself will be provided as part of these releases.

### 1.1.7 Dependencies

The utmost consideration will be given to keeping dependencies for the TSP's shared library and the tcscd to an absolute minimum. Existing dependencies are the libc library, OpenSSL's libcrypto, the pthreads library, a SOAP library and any libraries needed for GUI development. Since the list of libraries needed for the GUI can be quite long, every effort will be taken to minimize it.

<<It is important to note that these dependencies will not be the same on all platforms. For instance the crypto implementation has been designed so that other crypto interface libraries can be easily plugged in to the TSS at build time. Please see section 2.3 for more information on the pluggable crypto implementations.>>

### 1.1.8 Testing Considerations

A complete TSS 1.1 API test suite will be developed concurrently with the TSS. Once completed, the test suite will be integrable with the Linux Test Project's automated test environment. (The tests developed can also be run stand-alone). Once development of the stack is near completion, a daily run of the API test suite will be used to identify bugs. The test suite will be released separately from the TSS stack, available for download as a patch to the LTP tarball, as a tarball itself or from CVS. Included in the test suite package will be the TSS API programming reference and documentation for the LTC-TSS.

<< The test suite will interface to the TSP, which will then interact with the TCS interface which in turn interacts with the TPM through the TDDL interface. In this way all three interfaces will be tested. No tests specific to the TCS or TDDL interfaces will be written.>>

### 1.1.9 Documentation

A full API and programming reference as well as build and install documentation will be provided with the LTC-TSS. The API and programming reference will be packaged with the test cases (except where released in RPM form), while build and install information will be released with any source code release of the LTC-TSS.

### 1.1.10 Design Problems

The TPM spec mandates that RSA encrypted blobs will be padded using an OAEP padding parameter of the NULL terminated string "TCPA". In the existing OpenSSL implementation of RSA encryption, it is not possible to specify the OAEP padding parameter.

In order to set this parameter (and thereby decrypt data from a TPM), a change to OpenSSL will be required. Other cryptographic service providers may have a similar problem.

There are several different callbacks an application can register with the TSS. Unfortunately, the TSS 1.1 spec specifies that the addresses of these callback functions be set by the `Tspi_SetAttribUint32()` function, whose parameter is only 32bits long. This makes setting a callback function impossible on a 64bit platform. In order to fix this problem, a `Tspi_SetAttribUlong64()` will be defined and used solely to set 64 bit attributes of objects. This is still in compliance with the TSS 1.1 spec, which states in section 1.3 that “The addition of any functions do not exclude a TSS implementation from <sic> being considered a valid implementation.”

<<When the TSS specification was created, its use was considered mainly in an environment with a GUI available. This led to the decision that by default, a policy object is set to spawn a popup window to receive authorization data. This becomes a problem on platforms where no GUI is available. To solve this problem, the LTC-TSS will allow an administrator to set the default policy for secret modes to be either popup, plaintext, SHA1 or callback. This option can be set in the TCSD's configuration file.>>

~~The TSS 1.1's `Tspi_TPM_PcrExtend()` API specifies that an application can pass in an optional `TSS_PCR_EVENT` structure to be held by the TSS's PCR event log. However, if this parameter is left NULL, a PCR could be extended without leaving any logging trail. To close this hole, the LTC-TSS will deviate from the TSS 1.1 spec and require that for each PCR extend operation, the application must pass in a non-NULL `TSS_PCR_EVENT` structure to be logged.~~

### <<1.1.11 Permissions

The file access permissions will be set as follows:

File	User ID	Group ID	Permissions	Purpose
<code>tcspd</code>	<code>tss</code>	<code>tss</code>	<code>-rwx-----</code>	TSS Core Services Daemon
<code>libtspi.so</code>	<code>root</code>	<code>root</code>	<code>-r-xr-xr-x</code>	TSS Service Provider shared library
<code>system.data</code>	<code>tss</code>	<code>tss</code>	<code>-rw-----</code>	TCSD Persistent Storage file
<code>user.{pid}</code>	<code>process_pid</code>	<code>process_gid</code>	<code>-rw-----</code>	Per-process User Persistent Storage files
<code>/var/tpm</code>	<code>tss</code>	<code>tss</code>	<code>drw-rw-rwt</code>	Directory which will contain all Persistent Storage files

The TSS Core Services Daemon will be set to a unique, non-root UID (here named `tss`), to avoid making the system as susceptible to remote attacks.

Since any application should be able to link to the TSP interface, it is given user, group and other permission to read and execute.

The system persistent storage file, `system.data`, should only be readable by the TCSD, so its mode is restricted to `0600`, and its user and group ID's are set to that of the TCSD.

The user persistent storage files, shown above as `user.{pid}`, should only be readable and writable by the user who's process created them. Therefore its mode is restricted to `0600` and its user and group ID's are set to that of the creating process. Its worthwhile to

note that although child processes of a process that creates a persistent store do not inherit their parent's persistent store, there is no reason to restrict their access to the parent's persistent storage file.

The directory where all persistent storage files are written (by default `/var/tpm`) is readable and writable by user, group and others, with the sticky bit on. This will ensure that although all users can write their persistent storage files to this directory, users cannot overwrite or delete other user's files.

>>

## 2.0 TSS Service Provider

The TSS Service Provider is implemented as a C shared library (and optionally a static library) that links to the user's application. The TSP provides internal object management for keys, data blobs and all other types of objects, as well as a transparent interface to one or more TCS daemons and an interface to user persistent storage. The TSP also provides the user interface component for authentication data, an interface to the generic cryptographic routines (used internally) and a key caching mechanism.

### 2.1 Object Management

All TSP objects are maintained in one global linked list <<per process<sup>1</sup>. This list is updated and maintained as the application makes its API calls during its lifetime.>> When an application makes an API call that requires an object to be created, everything that is known about that object at object creation time is filled out. Subsequent API calls will complete the object's description and perhaps create the actual object at a later time. For example, if an application calls `Tspi_Context_CreateObject()` to create a policy object, the new object will be created internally to the TSP with type `TSS_OBJECT_TYPE_POLICY` and several default attributes. This object is added to the global linked list and is thus added to the TSP's management system. Subsequently the user may call `Tspi_SetAttribUint32()` to set the policy's secret mode or secret lifetime. The default values would then be overwritten in the TSP's object (which contained default values). Finally the user would call `Tspi_Policy_AssignToObject()` to associate the new policy with some object (usually an RSA key). The TSP's internal policy object would then have a pointer to the requested object assigned in the policy object, thereby associating it with the target object.

#### 2.1.0 Data Structures

`tspObject` - `include/spi_internal_types.h`

Each object created through the TSP exists in an entry in a linked list. Each entry in the list is an `tspObject` structure. The `tspObject` structure contains all generic object attributes such as size, type and the TSP and TCS contexts it may tie to. It also contains a pointer to the object itself.

---

<sup>1</sup> recognized as possible performance bottleneck. Implementations may be changed to work around any performance problems uncovered.

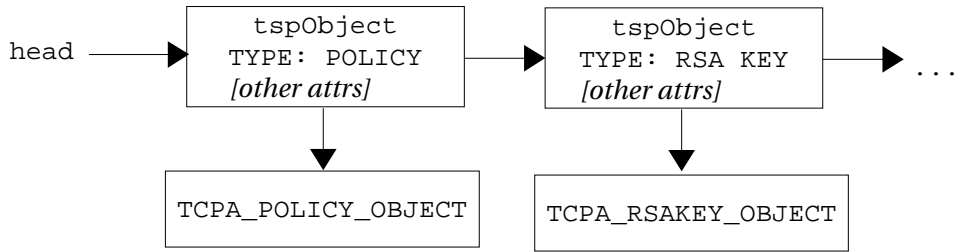


Figure 2.1.0: A sample object list

### 2.1.1 Object Creation Functions

`addObject` – *tspi/obj.c*

Synopsis:

```
TSS_HOBJECT addObject(UINT32 contextHandle, UINT32
objectType)
```

Description:

`addObject` adds a new object to the internal list of objects managed by the TSP. `contextHandle` is a TCS context handle, or 0 if a connection to a TCS does not yet exist. `objectType` is the type of TSP object to be created.

Return Value:

`addNbject` returns the handle to the newly created object.

Synchronization:

The TSP's object lock is held while the new object is added to the list.

Errors:

If an error occurs, `NULL_HOBJECT` is returned to the calling function.

`setObject` – *tspi/obj.c*

Synopsis:

```
TSS_RESULT setObject(TSS_HOBJECT objectHandle, void
*buffer, UINT32 sizeofBuffer)
```

Description:

`setObject` sets the internal memory pointer of the TSP context `objectHandle`. The TSP context's memory pointer is set to `buffer` and the size of the buffer is set to `sizeofBuffer`.

Return Value:

On success, `TSS_SUCCESS` is returned.

Synchronization:

The TSP's object lock is held while the new object is modified.

Errors:

If `objectHandle` is an invalid TSP object handle, `setObject` returns `TSS_E_INVALID_HANDLE`.

`createObject` – *tspi/obj.c*

Synopsis:

```
tspObject *createObject()
```

Description:

`createObject` calls `calloc` to create a new `tspObject` structure. The structure is freed either when an application calls `Tspi_Context_CloseObject()`, or `Tspi_Context_Close()`.

Return Value:

On success, a reference to the newly created object is returned.

Synchronization:

None.

Errors:

If `calloc` fails, `NULL` is returned and an error is logged.

## 2.1.2 Object Support functions

`getNextObjectHandle` – *tspi/obj.c*

Synopsis:

```
TSS_HOBJECT getNextObjectHandle()
```

Description:

`getNextObjectHandle` returns the next available TSP object handle.

Return Values:

The next available TSP object handle is returned.

Synchronization:

None.

Errors:

None.

`internal_GetContextObjectForContext` – *tspi/obj.c*

Synopsis:

```
TSS_RESULT internal_GetContextObjectForContext  
(TCS_CONTEXT_HANDLE tcsContext, TSS_HCONTEXT * tspContext)
```

Description:

`internal_GetContextObjectForContext` searches through the list of TSP contexts until it finds one whose TCS context matches `tcsContext`. Once it finds a match, `*tspContext` is set to the value of the matching TSP context.

Return Values:

On success, `TSS_SUCCESS` is returned.

Synchronization:

The TSP's object lock is held while the list is searched.

Errors:

If a matching context is not found, `TSS_E_INVALID_HANDLE` is returned.

`internal_GetContextForContextObject` – *tspi/obj.c*

**Synopsis:**

```
TSS_RESULT internal_GetContextForContextObject(TSS_HCONTEXT
hContext, TCS_CONTEXT_HANDLE * handleOut)
```

**Description:**

`internal_GetContextForContextObject` sets the value of `*handleOut` to the `TCS_CONTEXT_HANDLE` structure which matches the TSP context `hContext`.

**Return Values:**

On success, `TSS_SUCCESS` is returned and `*handleOut` is set.

**Synchronization:**

Calls `getAnObjectByHandle()`, which will hold the TSP object lock while searching the list.

**Errors:**

If `hContext` is not found, `TSS_E_INVALID_HANDLE` is returned.

**obj\_getPolicyOfObject – *tspi/obj.c***

**Synopsis:**

```
TSS_HOBJECT obj_GetPolicyOfObject(TSS_HOBJECT objectHandle,
UINT32 policyType)
```

**Description:**

`obj_GetPolicyOfObject` returns the policy object associated with TSP object `objectHandle` and of policy type `policyType`. If `objectHandle` is a handle to an object that has no policy associated with it, 0 is returned.

**Return Values:**

On success, the policy object that is requested is returned.

**Synchronization:**

Calls `getAnObjectByHandle()`, which will hold the TSP object lock while searching the list.

**Errors:**

If `objectHandle` is a handle to an object that has no policy associated with it, 0 is returned

**obj\_getContextForObject – *tspi/obj.c***

**Synopsis:**

```
TCS_CONTEXT_HANDLE obj_getContextForObject(TSS_HOBJECT
objectHandle)
```

**Description:**

`obj_getContextForObject` returns the TCS context associated with the TSP context `objectHandle`.

**Return Values:**

On success, the `TCS_CONTEXT_HANDLE` requested is returned.

**Synchronization:**

Calls `getAnObjectByHandle()`, which will hold the TSP object lock while searching the list.

**Errors:**

If `objectHandle` does not exist, `NULL_TCS_HANDLE` is returned.

#### `obj_getTpmObject` – *tspi/obj.c*

##### Synopsis:

```
TSS_RESULT obj_getTpmObject(TCS_CONTEXT_HANDLE tcsContext,
TSS_HOBJECT * out)
```

##### Description:

`obj_getTpmObject` searches the TSP context list for a TPM object that is associated with TCS `tcsContext`. If the search finds an object, the value of `*out` is set to it and `TSS_SUCCESS` is returned.

##### Return Values:

On success, `TSS_SUCCESS` is returned.

##### Synchronization:

Calls `getAnObjectByHandle()`, which will hold the TSP object lock while searching the list.

##### Errors:

If `tcsContext` is not an existing TCS context handle or there are no objects of type `TSS_OBJECT_TYPE_TPM`, `TSS_E_INVALID_HANDLE` is returned and `*out` is not touched.

#### `getObjectTypeByHandle` – *tspi/obj.c*

##### Synopsis:

```
UINT32 getObjectTypeByHandle(TSS_HOBJECT objectHandle)
```

##### Description:

`getObjectTypeByHandle` searches the TSP's object list for `objectHandle` and returns that object's type.

##### Return Values:

On success, the type of object `objectHandle` is returned.

##### Synchronization:

Calls `getAnObjectByHandle()`, which will hold the TSP object lock while searching the list.

##### Errors:

If `objectHandle` is not a valid TSP object handle, `NULL_HOBJECT` is returned.

#### `destroyObjectsByContext` – *tspi/obj.c*

##### Synopsis:

```
void destroyObjectsByContext(TCS_CONTEXT_HANDLE tcsContext)
```

##### Description:

`destroyObjectsByContext` searches through the TSP's list of objects, removing each one that's bound to the TCS represented by `tcsContext`. If no objects are bound to the TCS represented by `tcsContext`, no action is taken.

##### Return Values:

None.

Synchronization:

Holds the TSP object lock while searching the list.

Errors:

None.

`removeObject` – *tspi/obj.c*

Synopsis:

```
void removeObject(TSS_HOBJECT objectHandle)
```

Description:

`removeObject` searches the TSP's list of object for object handle `objectHandle` and removes that object from its list. If no object has the handle `objectHandle`, no action is taken.

Return Values:

None.

Synchronization:

Holds the TSP object lock while searching the list.

Errors:

None.

`destroyObject` – *tspi/obj.c*

Synopsis:

```
void destroyObject(tspObject *object)
```

Description:

`destroyObject` frees memory associated with the `tspObject` structure.

Return Values:

None.

Synchronization:

None.

Errors:

None.

`getObject` – *tspi/obj.c*

Synopsis:

```
TCPA_RESULT getObject(TSS_HOBJECT objectHandle, void  
**outBuffer, UINT32 * outSize)
```

Description:

`getObject` searches the TSP's internal list of objects for `objectHandle` and returned references to that object and its size.

Return Values:

On success, `*outSize` is set to the size of the found object, `*outSize` bytes are malloc'd for `*outBuffer` and a copy of the requested object is copied into `*outBuffer`. `TSS_SUCCESS` is then returned. The caller is expected to handle freeing of the object returned in `outBuffer`.

Synchronization:



Calls `getAnObjectByHandle()`, which will hold the TSP object lock while searching the list.

**Errors:**

If `malloc` fails, `TSS_E_OUTOFMEMORY` is returned. If `objectHandle` does not exist, `TSS_E_INVALID_HANDLE` is returned.

`concatObjects` – *tspi/obj.c*

**Synopsis:**

```
tspObject *concatObjects(tspObject ** first, tspObject
*second)
```

**Description:**

`concatObjects` appends the object pointed to by `second` immediately after the object pointed to by `first` in the TSP's internal list of objects.

**Return Values:**

`*first` is always returned.

**Synchronization:**

The TSP object lock is held while manipulating the list.

**Errors:**

None.

## 2.2 Key Cache Management

At the TSP level, the data structure used to keep track of loaded (cached) keys is based on the key's parent. (Here “loaded” is in the TSS sense of the word, meaning ready to use by a TSS application). Since each key loaded by the TSP was loaded in reference to a wrapping key (which ultimately led back to some key resident in the TCS), a list of linked lists is maintained in the TSP where the head of each list contains the wrapping key handle. All TSP keys created which had a certain wrapping key as its parent are then added to the list descending from that wrapping key's list head. A data structure such as the following will be built as keys are created:

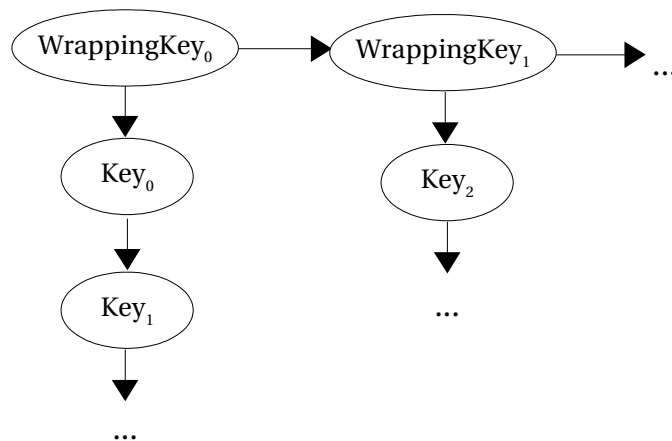


Figure 2.2a: TSP Key Cache objects.  $WrappingKey_0$  is an object internal to the TSP that contains the key handle of the key used to wrap  $Key_0$  and  $Key_1$ . Likewise,  $WrappingKey_1$  was used to wrap  $Key_2$  when it was created. These lists are strictly used for accounting of currently loaded keys, there is no direct link between the object list and this list (see figure 2.2b).

The above objects are very simple linked lists containing only a key handle and a pointer to the next list. The purpose of this data structure is merely to keep track of which keys are loaded at a given moment in the TSP. When the user calls `Tspi_Key_LoadKey()` or `Tspi_Key_UnloadKey()`, this data structure would be updated.

The object manager and key cache work together in the following way. When an application wants to create a new key, it calls `Tspi_Context_CreateObject()` with arguments to create an object of type `TSS_OBJECT_TYPE_RSAKEY`. `Tspi_Context_CreateObject()` internally calls `addObject()`, which creates the object and sets the TSP context and object type (implicitly, policy objects are also created for the RSA key for both migration and usage). `Tspi_Context_CreateObject()` then allocates and completes as much of a `TCPA_RSAKEY_OBJECT` as it can. (Since the key data is not yet known, it cannot be filled out, but the key's attributes are passed in, and so are already known). When the user then calls `Tspi_Key_CreateKey()`, passing in the handle to the created RSA key object, it does some sanity checking of the handle (and the policies for the key with that handle) and calls the TCS daemon to create the key. Once the call from the TCS daemon returns with the key data, that data is copied into the object's `TCPA_RSAKEY_OBJECT` structure and the key object creation is finished. The key is not yet in the TSP's key cache, however. Once the user calls one of the `XX_LoadKey` functions, `addKeyHandle()` is called and the key is considered cached.

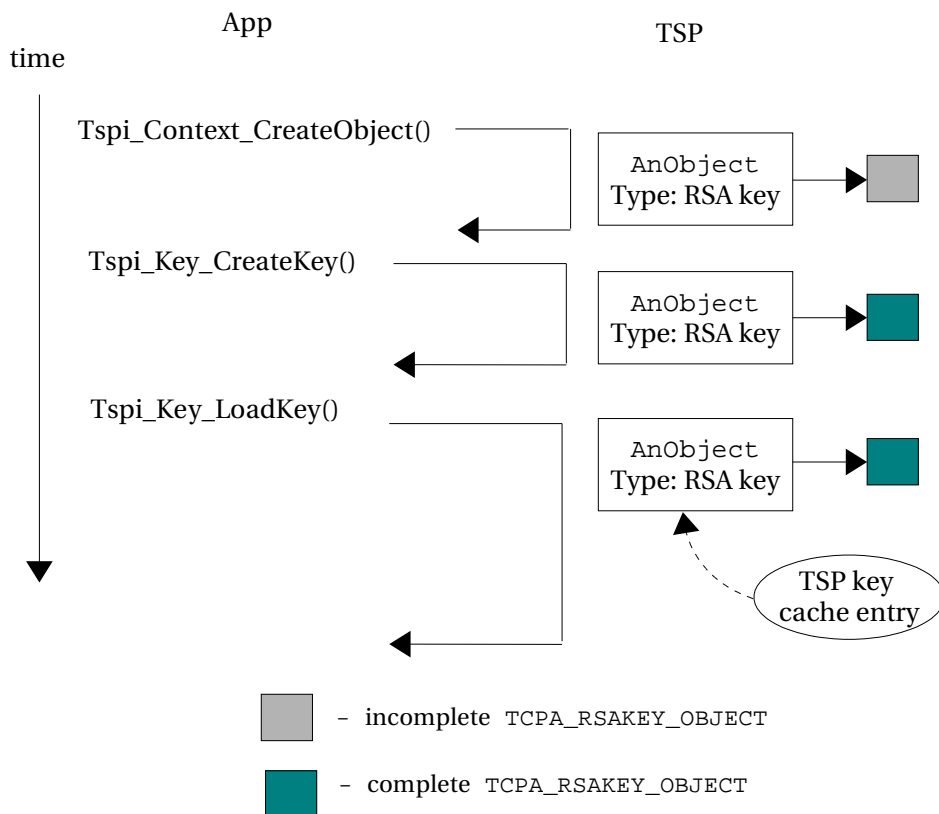


Figure 2.2b: TSP Key creation. The dotted arrow linking the cache entry to the object

represents the fact that there is no direct memory reference between the two, but both entities contain a handle to the same key.

## 2.2.0 Data Structures

TSPKeyHandleContainer - *include/spi\_internal\_types.h*

TSPKeyHandleContainer is used to create the linked list of TSP key handles associated with each TCSKeyHandleContainer.

TCSKeyHandleContainer - *include/spi\_internal\_types.h*

TCSKeyHandleContainer is a list of TCS keys that the TSP has knowledge of. For each key that the TSP creates, a TSPKeyHandleContainer is added to the TCSKeyHandleContainer for that key's parent. (See figure 2.2)

## 2.2.1 Object Creation Functions

createTSPKeyHandleContainer - *tspi/obj.c*

Synopsis:

```
TSPKeyHandleContainer *createTSPKeyHandleContainer()
```

Description:

createTSPKeyHandleContainer calloc's space for a new TSP key cache entry. The entry is freed when the corresponding key object is destroyed.

Return Values:

On success, a reference to the key cache entry is returned.

Synchronization:

None.

Errors:

If calloc fails, an error is logged and NULL is returned.

createTCSKeyHandleContainer - *tspi/obj.c*

Synopsis:

```
TCSKeyHandleContainer *createTCSKeyHandleContainer()
```

Description:

createTCSKeyHandleContainer calloc's space for a new TCS key cache entry. The entry is freed when the corresponding key object is destroyed.

Return Values:

On success, a reference to the key cache entry is returned.

Synchronization:

None.

Errors:

If calloc fails, an error is logged and NULL is returned.

## 2.2.2 Support Functions

`concatTSPKeyHandleContainer` – *tspi/obj.c*

Synopsis:

```
TSPKeyHandleContainer *concatTSPKeyHandleContainer
(TSPKeyHandleContainer ** first, TSPKeyHandleContainer
*second)
```

Description:

`concatTSPKeyHandleContainer` appends the object pointed to by `second` immediately after the object pointed to by `first` in the TSP's internal TSP key cache.

Return Values:

`*first` is always returned.

Synchronization:

The TSP key cache lock is held while manipulating the list.

Errors:

None.

`removeTSPKeyHandle` – *tspi/obj.c*

Synopsis:

```
void removeTSPKeyHandle(TSS_HKEY tspHandle)
```

Description:

`removeTSPKeyHandle` removes the TSP key referenced by `tspHandle`. If `tspHandle` does not exist, no action is taken.

Return Values:

None.

Synchronization:

The TSP key cache lock is held while manipulating the list.

Errors:

None.

`concatTCSKeyHandleContainer` – *tspi/obj.c*

Synopsis:

```
TCSKeyHandleContainer *concatTCSKeyHandleContainer
(TCSKeyHandleContainer ** first, TCSKeyHandleContainer
*second)
```

Description:

`concatTCSKeyHandleContainer` appends the object pointed to by `second` immediately after the object pointed to by `first` in the TSP's internal TCS key cache.

Return Values:

`*first` is always returned.

Synchronization:

The TSP key cache lock is held while manipulating the list.

Errors:

None.

`getTCSKeyHandleContainerByTCSHandle` – *tspi/obj.c*

Synopsis:

```
TCSKeyHandleContainer *getTCSKeyHandleContainerByTCSHandle
(TCS_KEY_HANDLE tcsHandle)
```

Description:

`getTCSKeyHandleContainerByTCSHandle` searches through the TSP's key cache for the key cache container whose TCS key handle matches `tcsHandle`.

Return Values:

On success, a reference to the key cache container requested is returned.

Synchronization:

The TSP key cache lock is held while the list is searched.

Errors:

If `tcsHandle` is not a valid TCS key handle, NULL is returned.

`addKeyHandle` – *tspi/obj.c*

Synopsis:

```
TSS_RESULT addKeyHandle(TCS_KEY_HANDLE tcsHandle, TSS_HKEY
tspHandle)
```

Description:

`addKeyHandle` creates a new TSP key cache entry bound to `tspHandle`. If a key cache container already exists for the TCS `tcsHandle`, the new TSP key cache entry is added to that, otherwise a new TCS key cache entry is created for `tcsHandle`. Allocated memory will be freed if the application explicitly calls `Tspi_Context_CloseObject()`, `Tspi_Context_Close()` or if the key is evicted.

Return Values:

On success, `TSS_SUCCESS` is returned.

Synchronization:

The TSP key cache lock is held while the new item is added.

Errors:

If `malloc` fails, `TSS_E_OUTOFMEMORY` is returned, and an error is logged.

`removeTCSKeyHandle` – *tspi/obj.c*

Synopsis:

```
void removeTCSKeyHandle(TCS_KEY_HANDLE tcsHandle)
```

Description:

`removeTCSKeyHandle` searches for the TCS key cache container that matches `tcsHandle` and removes it. All TSP key cache handles that were associated with the removed TCS key cache container are removed as well.

Return Values:

None.

Synchronization:

The TSP key cache lock is held while manipulating the list.

Errors:

None.

`getTCSKeyHandle` – *tspi/obj.c*

Synopsis:

```
TCS_KEY_HANDLE getTCSKeyHandle(TSS_HKEY tspHandle)
```

Description:

`getTCSKeyHandle` searches the TSP key cache list `tspHandle` and returns the TCS key handle associated with it.

Return Values:

On success, `getTCSKeyHandle` returns the TCS key handle requested.

Synchronization:

The TSP key cache lock is held while the list is searched.

Errors:

If no key handle matches `tspHandle`, `NULL_TCS_HANDLE` is returned.

## 2.3 Cryptographic Services

Cryptographic services are provided by a cryptographic library implementing the functions below (section 2.3.0). Crypto implementations will be found in their own directories under `tspi/crypto`, the current implementation being provided by `openssl`, in `tspi/crypto/openssl`. The decision on which library will be the cryptographic provider will be made at build time, based on which library is available.

<<Adding a new crypto implementation should be fairly straightforward:

- 1) Create a new directory, e.g. `src/tspi/crypto/myCrypto`.
- 2) Inside this directory, add a file named `'crypto.c'` which implements the functions in section 2.3.0
- 3) Add a check in `configure.in` for your crypto library and headers (see the `openssl` section of `configure.in` for an example). Make sure that the build system sets the variable `"CRYPTO_PACKAGE"` to the name of the directory you created in step 1. At build time, `src/tspi/crypto/$CRYPTO_PACKAGE/crypto.c` will be built.>>

One hurdle to implementing the cryptographic operations needed to interact with a TPM is the fact that the TPM requires the OAEP padding parameter of RSA encrypt/decrypt operations to be set to the NULL terminated string `"TCPA"`.

### 2.3.0 Cryptographic functions

`TSS_Hash` – *tspi/crypto/\$CRYPTO\_PACKAGE/crypto.c*

Synopsis:

```
TCPA_RESULT TSS_Hash(BYTE HashType, UINT32 BufSize, BYTE  
*Buf, BYTE * Digest)
```

Description:

Compute the hash value of the data pointed to by `Buf`. The hash algorithm to be used is specified by `HashType`. The length of the input buffer is specified by `BufSize` and `Digest` is the location where the resulting hash value will be written. It is assumed that

Digest points to enough bytes to hold the resulting hash.

**Return Values:**

On success, the hash is written to Digest and TSS\_SUCCESS is returned.

**Synchronization:**

None.

**Synchronization:**

None.

**Errors:**

If HashType is not a supported type, TSS\_E\_BAD\_PARAMETER is returned. If a call to the underlying crypto library fails, TSS\_E\_INTERNAL\_ERROR is returned and the underlying crypto library's error printing functions are invoked.

**TSS\_HMAC** – *tspi/crypto/\$CRYPTO\_PACKAGE/crypto.c*

**Synopsis:**

```
UINT32 TSS_HMAC(BYTE HashType, UINT32 SecretSize, BYTE*
Secret, UINT32 BufSize, BYTE* Buf, BYTE* hmacOut)
```

**Description:**

TSS\_HMAC computes the HMAC of the data at Buf based on the hash algorithm HashType. BufSize should be the number of bytes pointed to by Buf. SecretSize should be the size of the secret passed in at location Secret. The resulting HMAC will be written to the address pointed to by hmacOut. It is assumed that that hmacOut points to enough bytes to hold the resulting HMAC.

**Return Values:**

On success, the HMAC is written to hmacOut and TSS\_SUCCESS is returned.

**Synchronization:**

None.

**Synchronization:**

None.

**Errors:**

If HashType is not a supported type, TSS\_E\_BAD\_PARAMETER is returned.

**TSS\_RSA\_Encrypt** – *tspi/crypto/\$CRYPTO\_PACKAGE/crypto.c*

**Synopsis:**

```
int TSS_RSA_Encrypt(unsigned char *dataToEncrypt, unsigned
int dataToEncryptLen, unsigned char *encryptedData,
unsigned int *encryptedDataLen, unsigned char *publicKey,
unsigned int keysize)
```

**Description:**

TSS\_RSA\_Encrypt encrypts dataToEncryptLen bytes pointed to by dataToEncrypt using the key data publicKey. The size of publicKey is specified by keysize. The resulting encrypted data is written to encryptedData and \*encryptedDataLen is set to the number of encrypted bytes written.

Data passed to TSS\_RSA\_Encrypt will be encrypted using PKCS#1 OAEP padding and a public exponent of 3.

**Return Values:**

On success, `*encryptedDataLen` is set to the number of encrypted bytes written to `encryptedData` and `TSS_SUCCESS` is returned.

**Synchronization:**

None.

**Errors:**

If the RSA key object cannot be created due to a `malloc` failure, `TSS_E_OUTOFMEMORY` is returned and none of the input parameters are touched. If calls to the the underlying crypto library fail, `TSS_E_INTERNAL_ERROR` is returned.

`TSS_Verify` – *tspi/crypto/\$CRYPTO\_PACKAGE/crypto.c*

**Synopsis:**

```
int TSS_Verify(UINT32 iHashLength, BYTE *pHash, int
iKeyLength, unsigned char *pModulus, BYTE *pSignature)
```

**Description:**

`TSS_Verify` decrypts the signature at `pSignature` using the key `pModulus` and compares the result to `pHash`. `iHashLength` should be the length of the data at `pHash` and `iKeyLength` should be the length of the key.

Data passed to `TSS_Verify` will be decrypted using PKCS#1 OAEP padding and a public exponent of 3.

**Return Values:**

On success the result of the comparison between `pHash` and the decryption operation will be returned.

**Synchronization:**

None.

**Errors:**

If the RSA key object cannot be created due to a `malloc` failure, `TSS_E_OUTOFMEMORY` is returned and none of the input parameters are touched. If calls to the the underlying crypto library fail, `TSS_E_INTERNAL_ERROR` is returned.

`TSS_RSA_PKCS15_Encrypt` – *tspi/crypto/\$CRYPTO\_PACKAGE/crypto.c*

**Synopsis:**

```
int TSS_RSA_PKCS15_Encrypt(unsigned char *dataToEncrypt,
unsigned int dataToEncryptLen, unsigned char
*encryptedData, unsigned int *encryptedDataLen, unsigned
char * publicKey, unsigned int keysize, BYTE* seed);
```

**Description:**

`TSS_RSA_PKCS15_Encrypt` encrypts `dataToEncryptLen` bytes pointed to by `dataToEncrypt` using the key data `publicKey`. The size of `publicKey` is specified by `keysizes`. The resulting encrypted data is written to `encryptedData` and `*encryptedDataLen` is set to the number of encrypted bytes written. `seed` is currently unused.



Data passed to `TSS_RSA_Encrypt` will be encrypted using PKCS#1 v1.5 padding and a public exponent of 3.

**Return Values:**

On success, `*encryptedDataLen` is set to the number of encrypted bytes written to `encryptedData` and `TSS_SUCCESS` is returned.

**Synchronization:**

None.

**Errors:**

If the RSA key object cannot be created due to a `malloc` failure, `TSS_E_OUTOFMEMORY` is returned and none of the input parameters are touched. If calls to the underlying crypto library fail, `TSS_E_INTERNAL_ERROR` is returned.

## 2.4 Graphical User Interface

The sole GUI components to the LTC TSS will be the pop up windows used to input authentication data for new and existing keys. Any number of different toolkits can be used to implement the underlying functionality needed by the `popup_GetSecret()` function in section 2.4.0.

<<In the same way that the cryptographic implementations are pluggable, the GUI components will be as well. `DisplayNewPINWindow` and `DisplayPINWindow` will be the abstraction point here (these are the two functions called by `popup_GetSecret()`). In order to add a new type of GUI component to drive the popup messages, do the following:

- 1) Create a new directory, e.g. `src/tspi/gui/myGui`.
- 2) Inside this directory, create the files `'main.c'`, `'support.c'`, `'interface.c'` and `'callbacks.c'` which implement `DisplayNewPINWindow()` and `DisplayPINWindow()`.
- 3) Add a check in `configure.in` for your GUI library and headers (see the GTK section of `configure.in` for an example). Make sure that the build system sets the variable "GUI\_PATH" to the name of the directory you created in step 1. At build time, `src/tspi/gui/$GUI_PATH/*.c` will be built.>>

### 2.4.0 Graphical User Interface Functions

`popup_GetSecret` - *tspi/secrets.c*

**Synopsis:**

```
TSS_RESULT popup_GetSecret(UINT32 new_pin, BYTE *message,
void *auth_hash)
```

**Description:**

`popup_GetSecret` invokes an underlying implementation to display a GUI window for accepting authentication data. If `new_pin` is non-zero, `DisplayNewPINWindow` will be invoked, otherwise `DisplayPINWindow` will be invoked. Both `DisplayNewPINWindow` and `DisplayPINWindow` are implemented based on which GUI toolkit is available at build time.

`message` will be displayed in the title bar of the window created. The SHA-1 hash of the data collected by the PIN window will be written to `auth_hash`.

`DisplayNewPINWindow` is intended to have entry boxes for a password and confirm password (to receive a password which has not previously been entered), whereas `DisplayPINWindow` will only have a box to enter one password, where that password

has previously been passed into the TSP and is only being verified.

**Return Values:**

On success, the SHA-1 hash of the data collected by the PIN window will be written to `auth_hash` and `TSS_SUCCESS` is returned.

**Synchronization:**

None.

**Errors:**

If message is not set, or if the GUI pop up dialog is canceled by the user, `TSS_E_INTERNAL_ERROR` is returned.

## 2.5 Memory Management

The memory management functions in the TSP are used when data allocated by the TSS must be returned to the application. At a later time this data will need to be free'd due to a call to `Tspi_Context_CloseObject()` or explicitly by a call to `Tspi_Context_FreeMemory()`. Each malloc'd area of memory is associated with the TCS that the current TSP context is associated with. This will enable easy cleanup of memory allocated by a TCS for one or more TSP contexts which may close unexpected.

### 2.5.0 Data Structures

`MemSlot` - *include/memmgr.h*

A `MemSlot` holds a reference to a memory area and pointer to the next `MemSlot`.

`ContextMemSlot` - *include/memmgr.h*

A `ContextMemSlot` holds a `TCS_CONTEXT_HANDLE`, a pointer to a `MemSlot` and a pointer to the next `ContextMemSlot`. For each TCS context handle the TSP gets, a linked list of memory references will be maintained.

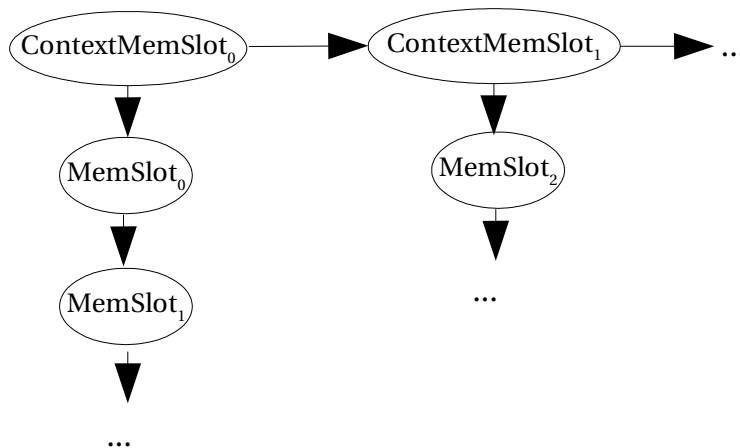


Figure 2.0.5.0: Memory references maintained by the TSP

## 2.5.1 Functions

`createMemSlot` - *tspi/memmgr.c*

Synopsis:

```
MemSlot *createMemSlot()
```

Description:

`createMemSlot` alloc's a `MemSlot` structure to be used by the memory management subsystem. The mem slot is freed when either the application closes its TSP context or explicitly calls `Tspi_Context_FreeMemory()`.

Return Values:

On success, a reference to a new, zeroed-out `MemSlot` structure is returned.

Synchronization:

None.

Errors:

If `calloc` fails, `TSS_E_OUTOFMEMORY` is returned.

`concatMemSlot` - *tspi/memmgr.c*

Synopsis:

```
MemSlot *concatMemSlot(MemSlot **first, MemSlot *second)
```

Description:

`concatMemSlot` appends the `MemSlot` pointed to by `second` immediately after the `MemSlot` pointed to by `first`.

Return Values:

`*first` is always returned.

Synchronization:

The TSP mem cache lock is held while manipulating the list.

Errors:

None.

`removeMemSlotByPointer` - *tspi/memmgr.c*

Synopsis:

```
TSS_RESULT removeMemSlotByPointer(ContextMemSlot * cms,  
void *pointer)
```

Description:

`removeMemSlotByPointer` searches through the memory list inside `cms` for `pointer`. If `pointer` is a valid memory pointer in `cms`, it is then removed.

Return Values:

On success, `TSS_SUCCESS` is returned.

Synchronization:

The TSP mem cache lock is held while removing the item from the list.

Errors:

If `pointer` is not found in `cms`'s memory list, `TSS_E_INTERNAL_ERROR` is returned.

#### `getSPIMemory - tspi/memmgr.c`

##### Synopsis:

```
void *getSPIMemory(TCS_CONTEXT_HANDLE tcsContext, UINT32
howMuch)
```

##### Description:

`getSPIMemory` attempts to allocate a `MemSlot` of size `howMuch` in the `ContextMemSlot` referenced by `tcsContext`. If a `ContextMemSlot` does not exist for `tcsContext`, a new one is created.

##### Return Values:

On success, `TSS_SUCCESS` is returned.

##### Synchronization:

Calls `concatMemSlot()`, which holds the mem cache lock while adding the `MemSlot`.

##### Errors:

If an error occurs, it is passed on to the caller.

#### `freeSPIMemory - tspi/memmgr.c`

##### Synopsis:

```
TSS_RESULT freeSPIMemory(TCS_CONTEXT_HANDLE tcsContext,
void *memPointer)
```

##### Description:

`freeSPIMemory` deallocates the memory located at `memPointer` from the `ContextMemSlot` structure associated with `tcsContext`.

##### Return Values:

On success, `TSS_SUCCESS` is returned.

##### Synchronization:

the  
Calls `removeMemSlotByPointer()`, which holds the mem cache lock while adding the `MemSlot`.

##### Errors:

If an error occurs, it is passed on through to the caller. If `tcsContext` is not a valid TCS context, `TSS_E_INVALID_HANDLE` is returned.

#### `isThisPointerSPI - tspi/memmgr.c`

##### Synopsis:

```
BOOL isThisPointerSPI(TCS_CONTEXT_HANDLE tcsContext, void
*memPointer)
```

##### Description:

`isThisPointerSPI` searches for the `ContextMemSlot` referenced by `tcsContext` for a pointer that matches `memPointer`. If it is found, `TRUE` is returned, else `FALSE`.

##### Return Values:

`TRUE` if `memPointer` is an existing reference to memory inside the

ContextMemSlot referenced by `tcsContext`, `FALSE` otherwise.

Synchronization:

Holds the mem slot pointer while searching the list.

Errors:

If `tcsContext` is not a valid TCS context, `FALSE` is returned.

## 2.6 Persistent Storage

User Persistent Storage (PS) is created to store keys on disk for loading at a later time. Stored in user PS are a key's public and encrypted private areas, the key's properties and the UUID of the key and its parent. The UUID's of keys kept in user PS can be the same as the UUID's of keys in other processes' user PS, since no one entity knows about both keys. If one process's TSP registers a key from its user PS into system PS and another process then tries to register its key into system PS with a duplicate UUID, the TSS will return `TSS_E_KEY_ALREADY_REGISTERED`.

<<The term 'persistent storage' can be confusing here, since the term persistent is usually understood to mean "across multiple processes lifetimes." Persistent storage is a more applicable term to the TCS's storage, since it does exist across processes lifetimes, TCSD restarts and platform restarts. However, the term 'persistent storage' is used here as well in order to match the language in the TSS 1.1 specification.>>

The user persistent storage maintained by a TSP is a per-process data store. Its lifetime is the same as that of the process that's using it. The user persistent storage will be created as `/var/tpm/user.{pid}`, where `{pid}` is the process id of the TSP. On `fork()`, the child process will not retain any access to the parent's TSP key store.

The user and system PS files will be binary stores of the keys and key attributes listed above. Keys will be stored in a way that makes key searches by UUID, public data and `TCPA_KEY` structure optimal (these are the 3 search types possible in explicit calls to the TCS). The format used is the following:

```
UINT32    num_keys_on_disk
TSS_UUID  UUID_key0
TSS_UUID  parent_UUID_key0
UINT16    public_key_size0
UINT16    blob_size0
UINT16    cache_flags0
BYTE[]    public_key0
BYTE[]    blob0
TSS_UUID  UUID_key1
TSS_UUID  parent_UUID_key1
UINT16    public_key_size1
UINT16    blob_size1
UINT16    cache_flags1
BYTE[]    public_key1
BYTE[]    blob1
[...]
EOF
```

The `cache_flags` variable will record where the key's parent is stored (User or System PS) and whether the key is valid or not. The valid flag is set when the key is written to disk and unset by any operation that unregisters the key.

Performance tests will determine whether its necessary to break out the public data area of each key. This should yield faster search times (since the entire key blob will not have to be read from disk to perform some searches), but will require more disk space, since the public data area is contained in the key's blob.

## 2.6.0 Persistent Storage Functions

`tspPS_GetParentUUIDByUUID - tspi/ps/tspps.c`

Synopsis:

```
TSS_RESULT tspPS_GetParentUUIDByUUID( char *szFileName,
TSS_UUID *childUUID, TSS_UUID *parentUUID )
```

Description:

`tspPS_GetParentUUIDByUUID` checks the persistent data store kept in the file `szFileName` for the UUID `childUUID` and if found, returns the parent UUID of it in `parentUUID`.

Return Values:

On success, `*parentUUID` is set to the requested UUID and `TSS_SUCCESS` is returned.

Synchronization:

The file itself is locked using `flock( )` while the file is searched.

Errors:

If `childUUID` is not found or its parent UUID is not found, `TCS_E_FAIL` is returned.

`tspPS_RemoveKeyByUUID - tspi/ps/tspps.c`

Synopsis:

```
TSS_RESULT tspPS_RemoveKeyByUUID(char* szFileName,
TSS_UUID* uuidToRemove)
```

Description:

`tspPS_RemoveKeyByUUID` checks the persistent data store kept in the file `szFileName` for the UUID `uuidToRemove` and if found, deletes the key associated with it from the store.

Return Values:

On success, the requested key is removed from persistent storage and `TSS_SUCCESS` is returned.

Synchronization:

The file itself is locked using `flock( )` while the key is removed.

Errors:

If `uuidToRemove` is not found or the remove operation fails, `TCS_E_FAIL` is returned.

`tspPS_GetKeyByUUID - tspi/ps/tspps.c`

Synopsis:

```
TSS_RESULT tspPS_GetKeyByUUID( char* szFileName, TSS_UUID
uuidToFind, BYTE* keyBuffer, UINT16* keyBufferSize )
```

Description:

`tspPS_GetKeyByUUID` checks the persistent data store kept in the file `szFileName`

for the UUID `uuidToFind` and if found, returns the key data associated with it in `keyBuffer` and the size of the key in `*keyBufferSize`.

**Return Values:**

On success, `keyBuffer` will contain the key data of key with UUID `uuidToFind` and the `keyBufferSize` is set to the size of the returned key and `TSS_SUCCESS` is returned.

**Synchronization:**

The file itself is locked using `flock( )` while the key is being read.

**Errors:**

If `uuidToFind` is not found or the operation of key extraction fails, `TCS_E_FAIL` is returned.

`tspPS_GetParentPSTypeUUID` - *tspi/ps/tspps.c*

**Synopsis:**

```
TSS_RESULT tspPS_GetParentPSTypeUUID( char* szFileName,
TSS_UUID    uuidToFind, UINT32* psTypeOut )
```

**Description:**

`tspPS_GetParentPSTypeUUID` checks the persistent data store kept in the file `szFileName` for the UUID `uuidToFind` and if found, returns the parent's persistent storage type in `*psTypeOut`.

**Return Values:**

On success, `*psTypeOut` is set to the requested persistent storage type and `TSS_SUCCESS` is returned.

**Synchronization:**

The file itself is locked using `flock( )` while the file is searched.

**Errors:**

If `uuidToFind` is not found or the operation of extracting the parent's persistent storage type from persistent storage fails, `TCS_E_FAIL` is returned.

`tspPS_IsPubRegistered` - *tspi/ps/tspps.c*

**Synopsis:**

```
TSS_RESULT tspPS_IsPubRegistered( char *szFileName,
TCPA_STORE_PUBKEY pub, BOOL *isReg )
```

**Description:**

`tspPS_IsPubRegistered` checks the persistent data store kept in the file `szFileName` for the public key data `pub` and if found, returns `TRUE` in the variable `isReg`.

**Return Values:**

On success, `*isReg` is set and `TSS_SUCCESS` is returned.

**Synchronization:**

The file itself is locked using `flock( )` while the file is searched.

**Errors:**

If no key matches `pub`, `TCS_E_FAIL` is returned and `*isReg` is set to `FALSE`.

*tspPS\_GetUUIDByPub - tspi/ps/tspps.c*

**Synopsis:**

```
TSS_RESULT tspPS_GetUUIDByPub( char* szFileName,
TCPA_STORE_PUBKEY pub, TSS_UUID* uuidOut )
```

**Description:**

tspPS\_GetUUIDByPub checks the persistent data store kept in the file szFileName for the public key data pub and if found, returns the UUID of it in \*uuidOut.

**Return Values:**

On success, \*uuidOut is set to the requested UUID and TSS\_SUCCESS is returned.

**Synchronization:**

The file itself is locked using flock( ) while the file is searched.

**Errors:**

If the public key data does not match any keys in the requested persistent store or the operation of extracting the UUID fails, TCS\_E\_FAIL is returned.

*tspPS\_WriteKeyToFile - tspi/ps/tspps.c*

**Synopsis:**

```
TSS_RESULT tspPS_WriteKeyToFile( char* szFileName, TSS_UUID
uuidOfKey, TSS_UUID uuidOfParent, UINT32* parentPS, BYTE*
tcpaKeyBlob, UINT32 tcpaKeyBlobSize )
```

**Description:**

tspPS\_WriteKeyToFile writes the key tcpaKeyBlob to the persistent store in file szFileName. uuidOfKey is set as the UUID and uuidOfParent is set as the key's parent's UUID. If parentPS is not NULL, the parent's persistent storage type is returned in parentPS.

**Return Values:**

On success, the key is written to persistent storage and TSS\_SUCCESS is returned.

**Synchronization:**

The file itself is locked using flock( ) while the key is being written.

**Errors:**

If any operation fails, TCS\_E\_FAIL is returned.

*tspPS\_ReplaceEncdata - tspi/ps/tspps.c*

**Synopsis:**

```
TSS_RESULT tspPS_ReplaceEncdata( char* szFileName, BYTE*
oldEncData, BYTE* newEncData )
```

**Description:**

tspPS\_ReplaceEncData checks the persistent data store kept in the file szFileName for a key with encrypted data matching oldEncData and replaces it with newEncData.

**Return Values:**

On success TSS\_SUCCESS is returned.

**Synchronization:**

The file itself is locked using flock( ) while the file is being manipulated.



Errors:

If oldEncData is not found or the operation fails, TCS\_E\_FAIL is returned.

## 2.6.1 Key Registration Functions

The following key registration functions sit on top of the persistent storage functions in section 2.6.0. They act as wrappers, first creating the persistent store file name based on their process ID and then returning a TSS\_RESULT based on whether the underlying persistent store function succeeds or fails.

keyreg\_IsKeyAlreadyRegistered - *tspi/keyreg.c*

Synopsis:

```
BOOL keyreg_IsKeyAlreadyRegistered(UINT32 keyBlobSize, BYTE
*keyBlob)
```

Description:

keyreg\_IsKeyAlreadyRegistered checks the persistent store for the public key data contained in keyBlob. keyBlobSize is currently unused.

Return Values:

If the key exists, TRUE is returned, else FALSE.

Synchronization:

The file lock is held by the layer below (see functions in section 2.6.0).

Errors:

If an error occurs, FALSE is returned and the error is logged.

keyreg\_WriteKeyToFile - *tspi/keyreg.c*

Synopsis:

```
TSS_RESULT keyreg_WriteKeyToFile(TSS_UUID myUUID, TSS_UUID
parentUUID, UINT32 parentPSType, UINT32 blobSize, BYTE
*blob)
```

Description:

keyreg\_WriteKeyToFile generates the persistent storage file based on the current process ID and calls the underlying persistent storage function to write the key blob into it.

Return Values:

On success TSS\_SUCCESS is returned.

Synchronization:

The file lock is held by the layer below (see functions in section 2.6.0).

Errors:

If an error occurs while generating the persistent storage filename, TSS\_E\_INTERNAL\_ERROR is returned.

keyreg\_RemoveKeyFromFile - *tspi/keyreg.c*

Synopsis:

```
TSS_RESULT keyreg_RemoveKeyFromFile(TCS_CONTEXT_HANDLE
tcsContext, TSS_UUID * myUUID, TCPA_PUBKEY * verifyKey)
```

Description:

keyreg\_RemoveKeyFromFile generates the persistent storage file based on the current process ID and calls the underlying persistent storage function to remove the key with UUID myUUID from it.

Return Values:

On success TSS\_SUCCESS is returned.

Synchronization:

The file lock is held by the layer below (see functions in section 2.6.0).

Errors:

If an error occurs while generating the persistent storage filename, TSS\_E\_INTERNAL\_ERROR is returned. If the underlying persistent storage function fails, TSS\_E\_PS\_KEY\_NOTFOUND is returned.

keyreg\_GetKeyByUUID - *tspi/keyreg.c*

Synopsis:

```
TSS_RESULT keyreg_GetKeyByUUID(TCS_CONTEXT_HANDLE
tcsContext, TSS_UUID myUUID, UINT32 * blobSizeOut, BYTE
**blob)
```

Description:

keyreg\_GetKeyByUUID generates the persistent storage file based on the current process ID and calls the underlying persistent storage function to retrieve the key \*blob from it. \*blobSizeOut is also set to the size of the retrieved key blob.

Return Values:

On success TSS\_SUCCESS is returned.

Synchronization:

The file lock is held by the layer below (see functions in section 2.6.0).

Errors:

If an error occurs while generating the persistent storage filename, TSS\_E\_INTERNAL\_ERROR is returned. If the underlying persistent storage function fails, its return code is passed through to the caller.

keyreg\_GetParentUUIDByUUID - *tspi/keyreg.c*

Synopsis:

```
TSS_RESULT keyreg_GetParentUUIDByUUID(TSS_UUID myUUID,
TSS_UUID * parentUUID)
```

Description:

keyreg\_GetParentUUIDByUUID generates the persistent storage file based on the current process ID and calls the underlying persistent storage function to retrieve the parent's UUID of the key with UUID myUUID.

Return Values:

On success TSS\_SUCCESS is returned and \*parentUUID is set.

Synchronization:

The file lock is held by the layer below (see functions in section 2.6.0).

Errors:

If an error occurs while generating the persistent storage filename, TSS\_E\_INTERNAL\_ERROR is returned. If the underlying persistent storage function fails, TSS\_E\_PS\_KEY\_NOTFOUND is returned.

keyreg\_GetParentPSTypeByUUID - *tspi/keyreg.c*

Synopsis:

```
TSS_RESULT keyreg_GetParentPSTypeByUUID( TSS_UUID
parentUUID, UINT32 * psTypeOut)
```

Description:

keyreg\_GetParentPSTypeByUUID generates the persistent storage file based on the current process ID and calls the underlying persistent storage function.

Return Values:

On success, \*psTypeOut is set to the persistent storage type of the parent UUID of the key matching parentUUID and TSS\_SUCCESS is returned.

Synchronization:

The file lock is held by the layer below (see functions in section 2.6.0).

Errors:

If an error occurs while generating the persistent storage filename, TSS\_E\_INTERNAL\_ERROR is returned. If the underlying persistent storage function fails, TSS\_E\_PS\_KEY\_NOTFOUND is returned.

keyreg\_replaceEncData\_PS - *tspi/keyreg.c*

Synopsis:

```
TSS_RESULT keyreg_replaceEncData_PS( BYTE * encData, BYTE
*newEncData)
```

Description:

keyreg\_replaceEncData\_PS generates the persistent storage file based on the current process ID and calls the underlying persistent storage function to replace encData with newEncData.

Return Values:

On success TSS\_SUCCESS is returned.

Synchronization:

The file lock is held by the layer below (see functions in section 2.6.0).

Errors:

If an error occurs while generating the persistent storage filename, TSS\_E\_INTERNAL\_ERROR is returned.

## 2.7 TCS Calling Interface

The format of the packets transferred between the TSP and the TCS layers are set to be defined by a proposal to the TSS Working Group (TSSWG) as of the writing of this document. The LTC-TSS will conform to this packet format once it is approved by the TSSWG. The format is expected to be SOAP based.

From the TSP's view, a transaction between the TCS and TSP would look like a

standard client-server interaction:



Figure 2.7: The interaction between a TSP thread and the TCS, from the TSP's perspective

The above table illustrates the program flow of a simple thread of the TSP only. The thread is blocked while waiting on `recv()` and continues once it returns. An advanced multi-threaded app may have several threads simultaneously blocked waiting for data from one or more TCS's.

Please refer to section 3.1 for information on the TCS's processing of packets.

## 2.8 Utilities

Some functions included in the LTC-TSS are not enumerated in this low level design . The use and purpose of these functions should be obvious to anyone reading the code and so are not included here. These are functions for things such as sanity checking whether a context exists or is the right type, utilities to manipulate blobs of data, routines to convert from one data type to another and so on. All functions not enumerated here are internal to the TSS only.

## 3.0 TCG Core Service

### 3.1 TCS Calling Interface

The TCG Core Services daemon is required to be implemented as a system service, being the sole access to the TPM hardware (through the TCG device driver layer). The TCS provides serialized TPM access to multiple TSP's, manages the system persistent key storage, manages requests made to the TPM as well as providing the API services specified in the TSS 1.1 specification. The TCS also manages the log of PCR events.

The TCS is multi-threaded, but the view of a transaction from the TCS side is slightly less complex:

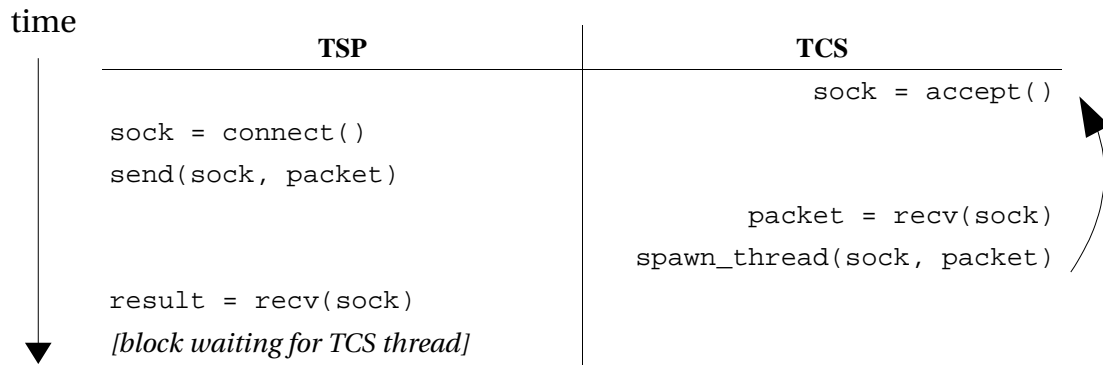


Figure 3.1: A transaction between the TCS and TSP from the TCS's perspective

In figure 3.1, the `spawn_thread()` function will be responsible for taking the received packet, decoding it, calling the correct function based on the TCS ordinal and calling `send()` to return the result of the operation to the TSP. This operation may block if the thread requires access to the TPM. Since a TPM request can take on the order of seconds to complete, a TPM Request Manager (TRM), residing inside the TCS, will block more than one thread from accessing the TPM at a time. The TRM will be responsible for maintaining the queue of current TPM requests, while the TCSKCM will dispatch threads to the TRM based on which key and auth contexts each request requires and which contexts are cached in the TPM. A simple scheduler algorithm such as round-robin would not be appropriate for the , since the TPM's resources are limited and swapping out contexts for requests from multiple threads has high overhead. Performance data from test cases will be used to identify an optimal algorithm.

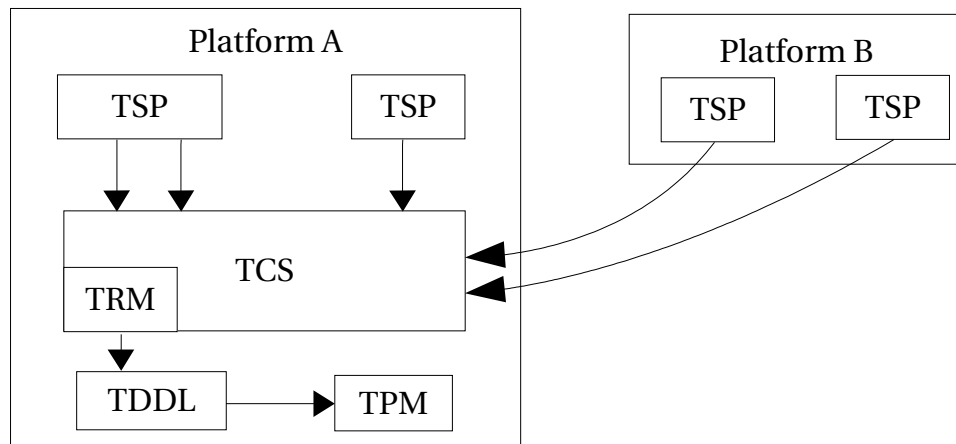


Figure 3.2: Multi-threaded access to a TPM. Each arrow represents a thread of execution.

### 3.1.0 Data Structures

#### TCSRequest

The `TCSRequest` structure will hold a decoded request from the TSP. After receiving the encoded data from the TSP, `packetDecode` will be called to convert it into a `TCSRequest` structure. The `TCSRequest` will then be handled by the TCS.

### 3.1.1 Functions

`packetDecode – tcs/main.c`

**Synopsis:**

```
struct TCSRequest *packetDecode(void *data)
```

**Description:**

`packetDecode` takes data received from a TSP and decodes it into a valid TCS request. The structure of `TCSRequest` will be described in an upcoming TSSWG proposal.

**Return Values:**

On success, `packetDecode` returns a newly allocated `TCSRequest` structure, filled out with data from `*data`.

**Synchronization:**

None.

**Errors:**

If a valid TCS request cannot be created from data, `NULL` is returned.

`packetEncode – tcs/main.c`

**Synopsis:**

```
int packetEncode(struct TCSRequest *req, char *dest)
```

**Description:**

`packetEncode` takes a `TCSRequest` to be returned to a TSP and encodes it into a format that the TSP will understand. The structure of `TCSRequest` will be described in an upcoming TSSWG proposal. `req` is encoded into the correct format and written to `dest`. The number of bytes written to `dest` is then returned.

**Return Values:**

On success, `packetEncode` returns the number of bytes written to `dest`.

**Synchronization:**

None.

**Errors:**

If a valid encoded packet cannot be created from `req`, `-TSS_E_INTERNAL_ERROR` is returned.

## 3.2 System Persistent Storage

Once a TCS daemon is started and it begins the task of managing a platform's TPM, the system PS begins to be populated with keys. Initially, only the TPM's Storage Root Key (SRK) is read from the chip and written to system persistent storage, but eventually TSP's will begin requesting that their keys be registered in system PS. Keys registered in system PS will be available until the application calls `Tspi_Context_UnregisterKey()`. The keys registered in system PS will remain on disk after the application exits and will survive system reboots/halts and TCS restarts. The amount of system PS is limited only by available disk

space.

System persistent storage is kept in the file `/var/tpm/system.data`. The system persistent storage file will have the same format as the user persistent storage files. The TCS daemon will create `/var/tpm/system.data` (including the directory) if it does not exist. The `/var/tpm` directory will be set with its sticky bit on, so that applications cannot modify or delete the system persistent storage file.

The system persistent storage file format will be the following:

```
UINT32    num_keys_on_disk
TSS_UUID  UUID_key0
TSS_UUID  parent_UUID_key0
UINT16    public_key_size0
UINT16    blob_size0
UINT16    cache_flags0
BYTE[]    public_key0
BYTE[]    blob0
TSS_UUID  UUID_key1
TSS_UUID  parent_UUID_key1
UINT16    public_key_size1
UINT16    blob_size1
UINT16    cache_flags1
BYTE[]    public_key1
BYTE[]    blob1
[...]
EOF
```

The `cache_flags` variable will record where the key's parent is stored (User or System PS) and whether the key is valid or not. The valid flag is set when the key is written to disk and unset by any operation that unregisters the key. At TCSD shutdown time, the `disk_cache` array in memory is scanned and any invalid keys are zeroized on disk. `num_keys_on_disk` keeps count of the total number of spaces on disk that keys occupy. The `num_keys_on_disk` variable is not affected by whether a key is valid or not.

### 3.2.0 Functions

`getParentUUIDByUUID - tcs/cache.c`

Synopsis:

```
TSS_RESULT getParentUUIDByUUID(TSS_UUID *uuid, TSS_UUID
**ret_uuid)
```

Description:

`getParentUUIDByUUID` checks the persistent data store for the UUID `uuid` and if found, returns the parent UUID of it in `*ret_uuid`.

Return Values:

On success, `*ret_uuid` is set to the requested UUID and `TSS_SUCCESS` is returned.

Synchronization:

The file itself is locked using `flock( )` while the file is searched.

Errors:

If `uuid` is not found, `TCS_E_FAIL` is returned.

`tcsPS_RemoveKeyByUUID - tcs/ps/tcsps.c`

**Synopsis:**

```
TSS_RESULT tcsPS_RemoveKeyByUUID(char* szFileName,
TSS_UUID* uuidToRemove)
```

**Description:**

`tcsPS_RemoveKeyByUUID` checks the persistent data store kept in the file `szFileName` for the UUID `uuidToRemove` and if found, deletes the key associated with it from the store.

**Return Values:**

On success, the requested key is removed from persistent storage and `TSS_SUCCESS` is returned.

**Synchronization:**

The file itself is locked using `flock()` while the key is removed..

**Errors:**

If `uuidToRemove` is not found or the remove operation fails, `TCS_E_FAIL` is returned.

`tcsPS_GetKeyByUUID - tcs/ps/tcsps.c`

**Synopsis:**

```
TSS_RESULT tcsPS_GetKeyByUUID( char* szFileName, TSS_UUID
uuidToFind, BYTE* keyBuffer, UINT16* keyBufferSize )
```

**Description:**

`tcsPS_GetKeyByUUID` checks the persistent data store kept in the file `szFileName` for the UUID `uuidToFind` and if found, returns the key data associated with it in `keyBuffer` and the size of the key in `*keyBufferSize`.

**Return Values:**

On success, `keyBuffer` will contain the key data of key with UUID `uuidToFind` and the `keyBufferSize` is set to the size of the returned key and `TSS_SUCCESS` is returned.

**Synchronization:**

None.

**Errors:**

If `uuidToFind` is not found or the operation of key extraction fails, `TCS_E_FAIL` is returned.

`tcsPS_GetParentPSTypeUUID - tcs/ps/tcsps.c`

**Synopsis:**

```
TSS_RESULT tcsPS_GetParentPSTypeUUID( char* szFileName,
TSS_UUID uuidToFind, UINT32* psTypeOut )
```

**Description:**

`tcsPS_GetParentPSTypeUUID` checks the persistent data store kept in the file `szFileName` for the UUID `uuidToFind` and if found, returns the parent's persistent storage type in `*psTypeOut`.

**Return Values:**

On success, `*psTypeOut` is set to the requested persistent storage type and



TSS\_SUCCESS is returned.

**Synchronization:**

The file itself is locked using flock( ) while the file is searched.

**Errors:**

If uuidToFind is not found or the operation of extracting the parent's persistent storage type from persistent storage fails, TCS\_E\_FAIL is returned.

*tcsPS\_IsPubRegistered - tcs/ps/tcps.c*

**Synopsis:**

```
TSS_RESULT tcsPS_IsPubRegistered( char* szFileName,
TCPA_STORE_PUBKEY pub, BOOL* isReg )
```

**Description:**

tcsPS\_IsPubRegistered checks the persistent data store kept in the file szFileName for the public key data pub and if found, returns TRUE in the variable isReg.

**Return Values:**

On success, \*isReg is set and TSS\_SUCCESS is returned.

**Synchronization:**

The file itself is locked using flock( ) while the file is searched.

**Errors:**

If no key matches pub, TCS\_E\_FAIL is returned and \*isReg is set to FALSE.

*tcsPS\_GetUUIDByPub - tcs/ps/tcps.c*

**Synopsis:**

```
TSS_RESULT tcsPS_GetUUIDByPub( char* szFileName,
TCPA_STORE_PUBKEY pub, TSS_UUID* uuidOut )
```

**Description:**

tcsPS\_GetUUIDByPub checks the persistent data store kept in the file szFileName for the public key data pub and if found, returns the UUID of it in \*uuidOut.

**Return Values:**

On success, \*uuidOut is set to the requested UUID and TSS\_SUCCESS is returned.

**Synchronization:**

The file itself is locked using flock( ) while the file is searched.

**Errors:**

If the public key data does not match any keys in the requested persistent store or the operation of extracting the UUID fails, TCS\_E\_FAIL is returned.

*tcsPS\_WriteKeyToFile - tcs/ps/tcps.c*

**Synopsis:**

```
TSS_RESULT tcsPS_WriteKeyToFile( char* szFileName, TSS_UUID
uuidOfKey, TSS_UUID uuidOfParent, UINT32* parentPS, BYTE*
tcpaKeyBlob, UINT32 tcpaKeyBlobSize )
```

**Description:**

tcsPS\_WriteKeyToFile writes the key tcpaKeyBlob to the persistent store in file szFileName. uuidOfKey is set as the UUID and uuidOfParent is set as the

key's parent's UUID. If `parentPS` is not NULL, the parent's persistent storage type is returned in `parentPS`.

**Return Values:**

On success, the key is written to persistent storage and `TSS_SUCCESS` is returned.

**Synchronization:**

The file itself is locked using `flock()` while the key is written.

**Errors:**

If any operation fails, `TCS_E_FAIL` is returned.

`tcsPS_ReplaceEncdata` - *tcs/ps/tcps.c*

**Synopsis:**

```
TSS_RESULT tcsPS_ReplaceEncdata( char* szFileName, BYTE*
oldEncData, BYTE* newEncData )
```

**Description:**

`tcsPS_ReplaceEncData` checks the persistent data store kept in the file `szFileName` for a key with encrypted data matching `oldEncData` and replaces it with `newEncData`.

**Return Values:**

On success `TSS_SUCCESS` is returned.

**Synchronization:**

The file itself is locked using `flock()` while the file is manipulated.

**Errors:**

If `oldEncData` is not found or the operation fails, `TCS_E_FAIL` is returned.

### 3.3 TCS Context Handling

TCS contexts are generated by the TCS daemon (not the TPM!) and are used by a TSP to tie objects and data that it creates to a specific TCS. The list of TCS contexts which have been created by the TCS and returned to some TSP is maintained internally to the TCS. When a TSP terminates its connection to the TCS, the TCS handle is destroyed.

#### 3.3.0 Data Structures

`tcsContext` - *include/tcs\_internal\_types.h*

The `tcsContext` structure contains a `TCS_CONTEXT_HANDLE` and a pointer to the next `tcsContext`. `tcsContext` structures are used to maintain a linked list of all existing `TCS_CONTEXT_HANDLE`'s.

#### 3.3.1 Functions

`createTcsContext` - *tcs/cxt.c*

**Synopsis:**

```
tcsContext *createTcsContext()
```

**Description:**

`createTcsContext` calls `calloc` to create a `tcsContext` and returns it. The TCS context is freed when the application closes its connection with the TCS.

**Return Values:**

On success, a reference to the newly created `tcsContext` is returned.

**Synchronization:**

None.

**Errors:**

If `calloc` fails, `NULL` is returned.

`getTcsContextByHandle` - *tcs/cxt.c*

**Synopsis:**

```
tcsContext *getTcsContextByHandle(TCS_CONTEXT_HANDLE  
tcsContext)
```

**Description:**

`getTcsContextByHandle` searches the TCS's internal list of `TCS_CONTEXT_HANDLE`'s for one that matches `tcsContext` and returns a reference to it.

**Return Values:**

On success, the `tcsContext` structure that holds `TCS_CONTEXT_HANDLE` `tcsContext` is returned.

**Synchronization:**

The TCS context lock is held while the list is being searched.

**Errors:**

If `tcsContext` is not found, `NULL` is returned.

`destroyTcsContext` - *tcs/cxt.c*

**Synopsis:**

```
void destroyTcsContext(TCS_CONTEXT_HANDLE tcsContext)
```

**Description:**

`destroyTcsContext` removes the internal data structures used to maintain a reference to `tcsContext`.

**Return Values:**

None.

**Synchronization:**

The TCS context lock is held while the list is being searched.

**Errors:**

None.

`makeTcsContext` - *tcs/cxt.c*

**Synopsis:**

```
TCS_CONTEXT_HANDLE makeTcsContext()
```

Description:

`makeTcsContext` creates a `tcsContext` and adds its to the TCS's internal list. In the process, a new `TCS_CONTEXT_HANDLE` is created and returned.

Return Values:

On success, the newly created `TCS_CONTEXT_HANDLE` is returned.

Synchronization:

The TCS context lock is held while the list is being manipulated.

Errors:

If `malloc` fails, an error is logged and `NULL_TCS_HANDLE` is returned.

### 3.4 Event Handling

A PCR event is recorded when the application passes in a `TSS_PCR_EVENT` structure to `Tspi_TPM_PcrExtend()`. The event type and internals of the `TSS_PCR_EVENT` structure are all application defined, the TSS only has to append the event to the event log for the PCR being manipulated. The event log is maintained by the TCS in order to return to the application on a call to `Tspi_TPM_GetEvent()`, `Tspi_TPM_GetEvents()` or `Tspi_TPM_GetEventLog()`.

PCR Events are logged using a linked list of `TSS_PCR_EVENT` structures per PCR. Each time an event occurs on a PCR, a new event structure is added to the linked list that corresponds to that PCR.

#### 3.4.0 Data Structures

`PCREventLog` - *include/tcs\_internal\_types.h*

`PCREventLog` structures are used to keep pointers to the head and tail of each list of events per PCR.

`PCREventContainer` - *include/tcs\_internal\_types.h*

`PCREventContainer` structures contain an event, and event number and a pointer to the next `PCREventContainer`.

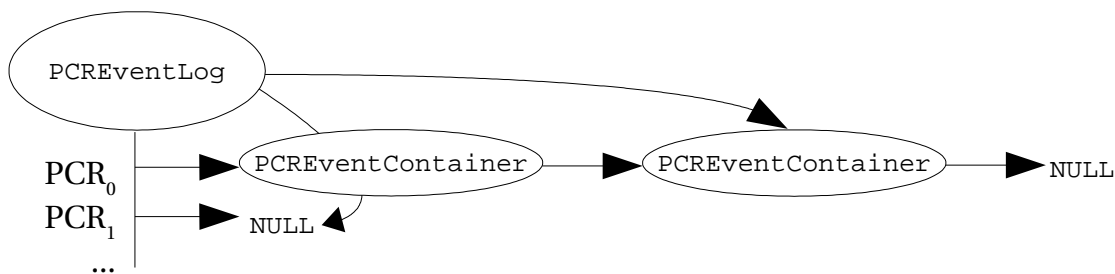


Figure 3.5.0: A PCR Event Log structure. At this point, 2 events have occurred on `PCR0`, no events have occurred on `PCR1` and `PCR's 2+` are not shown. A `PCREventLog` structure contains 2 arrays of pointers, one set to the head of each linked list of events (per PCR) and one to each tail.

### 3.4.1 Functions

`initEventLog` - *tcs/tcsem.c*

Synopsis:

```
TSS_RESULT initEventLog(TCS_CONTEXT_HANDLE tcsContext)
```

Description:

`initEventLog` queries the TPM to find out how many PCR's it supports, then `calloc`'s a new `PCREventLog` structure. Allocated PCR event log structures are not destroyed until a TCS shutdown.

Return Values:

On success, the TCS event log is created and `TSS_SUCCESS` is returned.

Synchronization:

The TCS event log lock is held while the event log is being created.

Errors:

If `tcsContext` is not valid or the query of the TPM fails, `TCS_E_FAIL` is returned.  
If `calloc` fails, `TSS_E_OUTOFMEMORY` is returned.

`copyPcrEvent` - *tcs/tcsem.c*

Synopsis:

```
TSS_RESULT copyPcrEvent(TCPA_PCR_EVENT *dest,  
                        TCPA_PCR_EVENT *source)
```

Description:

`copyPcrEvent` copies the `TCPA_PCR_EVENT` `source` to `dest`. `malloc` is called to create `dest`. The caller is expected to free the copied PCR event.

Return Values:

On success, `TSS_SUCCESS` is returned.

Synchronization:

The TCS event log lock is held while the event is copied.

Errors:

If `malloc` fails, `TSS_E_OUTOFMEMORY` is returned and the event is not copied.

`addPcrEvent` - *tcs/tcsem.c*

Synopsis:

```
TSS_RESULT addPcrEvent(TCPA_PCR_EVENT event)
```

Description:

`addPcrEvent` adds a new PCR event to the TCS's internal PCR event log. `event.PCRIndex` should contain the PCR number that the even occurred on. PCR events are not deallocated until a TCS shutdown.

Return Values:

On success, the event is added and `TSS_SUCCESS` is returned.

Synchronization:

The TCS event log lock is held while the event is being added.

Errors:

If `malloc` fails in creating the new `PCREvent` structure, `TSS_E_OUTOFMEMORY` is returned.

`getLastEventNumber` - *tcs/tcsem.c*

Synopsis:

```
UINT32 getLastEventNumber(UINT32 pcrIndex)
```

Description:

`getLastEventNumber` returns the latest event number for PCR `pcrIndex`.

Return Values:

On success, the last event number is returned.

Synchronization:

The TCS event log lock is held while the event number is being retrieved.

Errors:

If no events have occurred on the specified PCR, -1 is returned.

`getPCREventByNumber` - *tcs/tcsem.c*

Synopsis:

```
TCPA_PCR_EVENT *getPCREventByNumber(UINT32 pcrIndex, UINT32  
eventNumber)
```

Description:

`getPCREventByNumber` retrieves the `TCPA_PCR_EVENT` structure with event number `eventNumber` from the event log of the PCR with index `pcrIndex`.

Return Values:

On success, a reference to the requested `TCPA_PCR_EVENT` structure is returned.

Synchronization:

The TCS event log lock is held while the event list is being searched.

Errors:

If `pcrIndex` is out of range or the event number DNE, `NULL` is returned.

`getAllEvents` - *tcs/tcsem.c*

Synopsis:

```
TCPA_RESULT getAllEvents(TCS_CONTEXT_HANDLE hContext,  
TCS_PCR_EVENT ** list, UINT32 * howMany)
```

Description:

`getAllEvents` returns an array of `TCS_PCR_EVENT`'s, each element of which is a list of `TCPA_PCR_EVENT`'s. `*howMany` is set to the total number of `TCPA_PCR_EVENT`'s returned.

Return Values:

On success, `TSS_SUCCESS` is returned and `*list` and `*howMany` are set.

Synchronization:

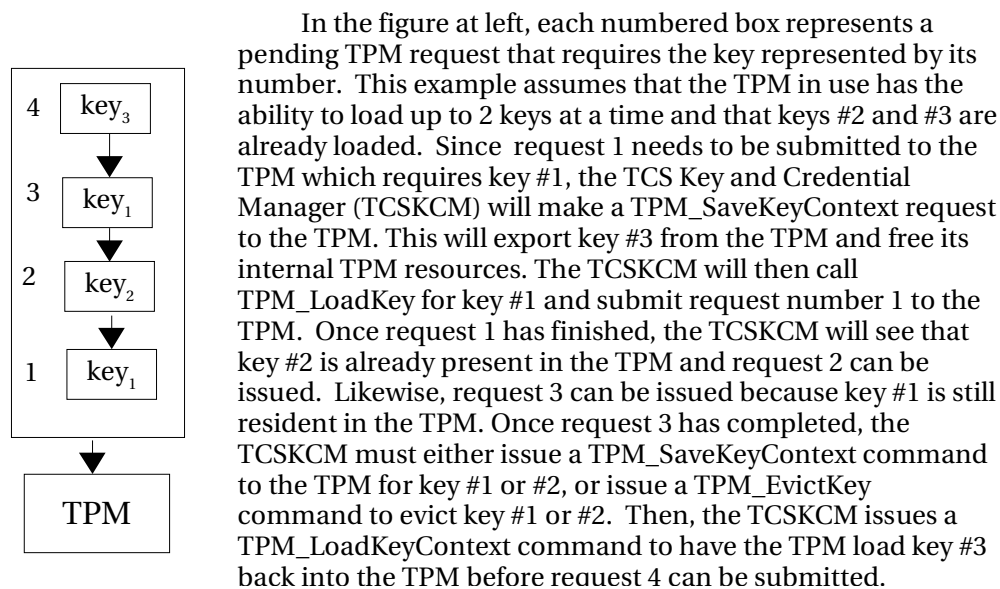
The TCS event log lock is held while the event list is being copied.

Errors:

If `malloc` fails, `TSS_E_OUTOFMEMORY` is returned. If `hContext` is not valid, `TCS_E_INVALID_CONTEXTHANDLE` is returned.

### 3.5 Key Cache Management

TCS key cache operations are initiated by a TSP. It is the TCS's job to juggle the requests of multiple TSP's at once, making the necessary TPM calls to swap key contexts. In order to facilitate access by multiple TSP's, a v1.1 TPM optionally provides the ability to swap out key and auth session contexts. This frees up the TPM's limited internal resources and allows the TCS to more easily context switch between the resources needed by multiple TSP processes. The following example has been simplified from an implementation perspective to illustrate key caching. Please see section 3.1 for information on how the TCSKCM and TRM work together.



In the figure at left, each numbered box represents a pending TPM request that requires the key represented by its number. This example assumes that the TPM in use has the ability to load up to 2 keys at a time and that keys #2 and #3 are already loaded. Since request 1 needs to be submitted to the TPM which requires key #1, the TCS Key and Credential Manager (TCSKCM) will make a `TPM_SaveKeyContext` request to the TPM. This will export key #3 from the TPM and free its internal TPM resources. The TCSKCM will then call `TPM_LoadKey` for key #1 and submit request number 1 to the TPM. Once request 1 has finished, the TCSKCM will see that key #2 is already present in the TPM and request 2 can be issued. Likewise, request 3 can be issued because key #1 is still resident in the TPM. Once request 3 has completed, the TCSKCM must either issue a `TPM_SaveKeyContext` command to the TPM for key #1 or #2, or issue a `TPM_EvictKey` command to evict key #1 or #2. Then, the TCSKCM issues a `TPM_LoadKeyContext` command to have the TPM load key #3 back into the TPM before request 4 can be submitted.

Figure 3.6: Key contention within the TCS

If the TPM does not support the `TPM_SaveKeyContext` / `TPM_LoadKeyContext` interface (optional for a v1.1 TPM), `TPM_LoadKey` and `TPM_EvictKey` will be used. The TCSKCM will attempt to be optimal in its calls to the TPM, ordering them as appropriate to minimize the total number of TPM requests .

#### 3.5.0 Data Structures

CacheStruct - *tcs/cache.c*

The `CacheStruct` structure contains information about the key being kept in the TCS's cache. `CacheStruct`'s contain info such as the key's UUID, its parent's UUID, its public data, the key blob itself, whether the key is actually resident in the TPM and so on.

KeyCacheEntry - *tcs/cache.c*

`KeyCacheEntry` is the structure used to maintain a list of all keys in the TCS key cache. It

contains a pointer to the next cache entry, a `CacheStruct` structure, the key's `TCS_KEY_HANDLE` and a time stamp. The time stamp is used to keep track of the least recently used key, in case we need to evict it.

### 3.5.1 Functions

`getParentPubBySlot` – *tcs/cache.c*

Synopsis:

```
TCPA_STORE_PUBKEY *getParentPubBySlot(TCPA_KEY_HANDLE  
    tpm_handle)
```

Description:

`getParentPubBySlot` searches through the TCS key cache for an entry matching `tpm_handle`. When its found, a reference to that key's parent's public key is returned.

Return Values:

On success, a reference to the requested key's parent's public key is returned.

Synchronization:

The TCS key cache lock is held while the list is being searched.

Errors:

If the slot is not found, `NULL` is returned.

`getPubBySlot` – *tcs/cache.c*

Synopsis:

```
TCPA_STORE_PUBKEY *getPubBySlot(TCPA_KEY_HANDLE tpm_handle)
```

Description:

`getPubBySlot` searches through the TCS key cache for an entry matching `tpm_handle`. When its found, a reference to that key's public key is returned.

Return Values:

On success, a reference to the requested key's public key is returned.

Synchronization:

The TCS key cache lock is held while the list is being searched.

Errors:

If the slot is not found, `NULL` is returned.

`getPubByHandle` – *tcs/cache.c*

Synopsis:

```
TCPA_STORE_PUBKEY *getPubByHandle(TCS_KEY_HANDLE  
    tcs_handle)
```

Description:

`getPubByHandle` searches through the TCS key cache for an entry matching `TCS_KEY_HANDLE tcs_handle`. When its found, a reference to that key's public key is returned.

Return Values:

On success, a reference to the requested key's public key is returned.

Synchronization:

The TCS key cache lock is held while the list is being searched.



**Errors:**

If the TCS context is not found, NULL is returned.

**setParentByHandle – *tcs/cache.c***

**Synopsis:**

```
TSS_RESULT setParentByHandle(TCS_KEY_HANDLE tcs_handle,  
TCS_KEY_HANDLE p_tcs_handle)
```

**Description:**

setParentByHandle searches through the TCS key cache for an entry matching TCS\_KEY\_HANDLE tcs\_handle. If its found, a key with a handle matching p\_tcs\_handle is searched for. If they're both found, a pointer from child to parent is set internally in the TCS key\_disk\_cache structure of the child.

**Return Values:**

On success, TSS\_SUCCESS is returned.

**Synchronization:**

The TCS key cache lock is held while the list is being searched.

**Errors:**

If the TCS context is not found, TCS\_E\_FAIL is returned.

**getUuidByPub – *tcs/cache.c***

**Synopsis:**

```
TSS_UUID *getUuidByPub(TCPA_STORE_PUBKEY *pub)
```

**Description:**

getUuidByPub searches through the TCS key cache for an entry who's public key matches pub. When its found, a reference to that key's UUID is returned.

**Return Values:**

On success, a reference to the requested key's UUID is returned.

**Synchronization:**

The TCS key cache lock is held while the list is being searched.

**Errors:**

If the cache cannot be initialized, or no cached key matches pub, NULL is returned.

**getUUIDByEncData – *tcs/cache.c***

**Synopsis:**

```
TSS_UUID *getUUIDByEncData(BYTE * encData)
```

**Description:**

getUUIDByEncData searches through the TCS key cache for an entry who's encrypted data area matches encData. When its found, a reference to that key's UUID is returned.

**Return Values:**

On success, a reference to the requested key's UUID is returned.

**Synchronization:**

The TCS key cache lock is held while the list is being searched.

**Errors:**

If the cache cannot be initialized, or no cached key matches `encData`, `NULL` is returned.

`getTCSKeyHandleByEncData` – *tcs/cache.c*

**Synopsis:**

```
TCS_KEY_HANDLE getTCSKeyHandleByEncData(BYTE * encData)
```

**Description:**

`getTCSKeyHandleByEncData` searches through the TCS key cache for an entry whose encrypted data area matches `encData`. When it's found, that key's `TCS_KEY_HANDLE` is returned.

**Return Values:**

On success, the requested key's `TCS_KEY_HANDLE` is returned.

**Synchronization:**

The TCS key cache lock is held while the list is being searched.

**Errors:**

If the cache cannot be initialized, or no cached key matches `encData`, `NULL_TCS_HANDLE` is returned.

`replaceEncData_knowledge` – *tcs/cache.c*

**Synopsis:**

```
void replaceEncData_knowledge(BYTE * encData, BYTE  
*newEncData)
```

**Description:**

`replaceEncData_knowledge` searches through the TCS key cache for an entry with encrypted data matching `encData` and replaces that encrypted data with `newEncData`.

**Return Values:**

On success, the requested `encData` is replaced.

**Synchronization:**

The TCS key cache lock is held while the list is being manipulated.

**Errors:**

If the cache cannot be initialized, or if a key matching `encData` cannot be found, no action is taken.

`add_tcs_cache_entry` – *tcs/cache.c*

**Synopsis:**

```
TSS_RESULT add_tcs_cache_entry(TCS_KEY_HANDLE tcs_handle,  
TCPA_KEY_HANDLE tpm_handle, TCPA_KEY *blob)
```

**Description:**

`add_tcs_cache_entry` creates a new TCS key cache entry for the key with handle `tcs_handle` if no entry exists with the same TCS key handle. `tpm_handle` and `tcs_handle` are set in the new cache object and its time stamp is set. New memory is also allocated for a `TCPA_KEY` and `blob` is copied into the object of the new key cache

entry. Memory allocated is freed if the key is explicitly evicted by a call to `Tspi_Key_EvictKey()`.

**Return Values:**

On success, or if a cache entry with the same TCS handle exists, `TSS_SUCCESS` is returned.

**Synchronization:**

The TCS key cache lock is held while the list is being searched.

**Errors:**

If `malloc` fails, `TSS_E_OUTOFMEMORY` is returned.

**`setSlotBySlot` – *tcs/cache.c***

**Synopsis:**

```
TSS_RESULT setSlotBySlot(UINT32 old_handle, UINT32
new_handle)
```

**Description:**

`setSlotBySlot` searches the TCS's key cache for an entry with a `TCPA_KEY_HANDLE` matching `old_handle`. If found, it replaces the `TCPA_KEY_HANDLE` in the cache entry with `new_handle` and updates the slot's time stamp. `setSlotBySlot` is used by the TCSKCM to update a cache entry when a key is loaded into the TPM that has been previously loaded.

**Return Values:**

On success, `TSS_SUCCESS` is returned.

**Synchronization:**

The TCS key cache lock is held while the list is being manipulated.

**Errors:**

If no cache entry has a `TCPA_KEY_HANDLE` matching `old_handle`, `TCS_E_FAIL` is returned.

**`setSlotByHandle` – *tcs/cache.c***

**Synopsis:**

```
TSS_RESULT setSlotByHandle(TCS_KEY_HANDLE tcs_handle,
TCPA_KEY_HANDLE tpm_handle)
```

**Description:**

`setSlotByHandle` searches the TCS's key cache for an entry whose TCS key handle matches `tcs_handle`. If found, the entry's `TCPA_KEY_HANDLE` is set to `tpm_handle` and its time stamp is updated.

**Return Values:**

On success, `TSS_SUCCESS` is returned.

**Synchronization:**

The TCS key cache lock is held while the list is being manipulated.

**Errors:**

If no key cache entry has a `TCS_KEY_HANDLE` matching `tcs_handle`, `TCS_E_FAIL` is returned.

`remove_tcs_cache_entry` – *tcs/cache.c*

Synopsis:

```
TSS_RESULT remove_tcs_cache_entry(TCS_KEY_HANDLE
tcs_handle)
```

Description:

`removetcs_cache_entry` searches through the TCS key cache for an entry with a TCS key handle matching `tcs_handle`. If found, the entry is removed from the cache.

Return Values:

None.

Synchronization:

The TCS key cache lock is held while the list is being manipulated.

Errors:

If no cache entry matches `tcs_handle`, `TCS_E_FAIL` is returned.

`setUuidsByPub` – *tcs/cache.c*

Synopsis:

```
TSS_RESULT setUuidsByPub(TCPA_STORE_PUBKEY *pub, TSS_UUID
*uuid, TSS_UUID *p_uuid)
```

Description:

`setUuidsByPub` searches through the TCS key cache for an entry with a public key matching `pub`. If found, the cache entry's UUID and parent UUID are set to `uuid` and `p_uuid` respectively.

Return Values:

On success, `setUuidsByPub` returns `TSS_SUCCESS`.

Synchronization:

The TCS key cache lock is held while the list is being manipulated.

Errors:

If no key's public data matches `pub`, `TCS_E_FAIL` is returned.

`getSlotByHandle` – *tcs/cache.c*

Synopsis:

```
TCPA_KEY_HANDLE getSlotByHandle(TCS_KEY_HANDLE tcs_handle)
```

Description:

`getSlotByHandle` searches through the TCS key cache for an entry with a TCS handle matching `tcs_handle`. If found, the entry's `TCPA_KEY_HANDLE` is returned.

Return Values:

On success, the entry's `TCPA_KEY_HANDLE` is returned.

Synchronization:

The TCS key cache lock is held while the list is being manipulated.

Errors:

If no cache entry's TCS handle matches `tcsHandle`, `NULL_TPM_HANDLE` is returned.

*getSlotByPub – tcs/cache.c*

Synopsis:

```
TCPA_KEY_HANDLE getSlotByPub(TCPA_STORE_PUBKEY *pub)
```

Description:

*getSlotByPub* searches through the TCS key cache for an entry with a public key matching *pub*. If found, the entry's *TCPA\_KEY\_HANDLE* is returned.

Return Values:

On success, the entry's *TCPA\_KEY\_HANDLE* is returned.

Synchronization:

The TCS key cache lock is held while the list is being searched.

Errors:

If no cache entry's public key matches *pub*, *NULL\_TPM\_HANDLE* is returned.

*getTCSKeyHandleByPub – tcs/cache.c*

Synopsis:

```
TCS_KEY_HANDLE getTCSKeyHandleByPub(TCPA_STORE_PUBKEY *pub)
```

Description:

*getTCSKeyHandleByPub* searches through the TCS key cache for an entry with a public key matching *pub*. If found, the entry's TCS key handle is returned.

Return Values:

On success, the requested TCS key handle is returned.

Synchronization:

The TCS key cache lock is held while the list is being searched.

Errors:

If no cache entry's public key matches *pub*, *NULL\_TCS\_HANDLE* is returned.

*getParentPubByPub – tcs/cache.c*

Synopsis:

```
TCPA_STORE_PUBKEY *getParentPubByPub(TCPA_STORE_PUBKEY  
*pub)
```

Description:

*getParentPubByPub* searches through the TCS key cache for an entry with a public key matching *pub*. If found, a reference to the entry's parent's public key is returned.

Return Values:

On success, a reference to the requested key's parent's public key is returned.

Synchronization:

The TCS key cache lock is held while the list is being searched.

Errors:

If no cache entry's public key matches *pub*, *NULL* is returned.

*isKeyInMemCache – tcs/cache.c*

Synopsis:

```
BOOL isKeyInMemCache(TCS_KEY_HANDLE tcs_handle)
```

**Description:**

`isKeyInMemCache` searches through the TCS key cache for an entry with a TCS key handle matching `tcs_Handle`. If found, `TRUE` is returned. If the key is not found, `FALSE` is returned.

**Return Values:**

The existence of the key in the TCS key cache is returned.

**Synchronization:**

The TCS key cache lock is held while the list is being searched.

**Errors:**

None.

`getBlobByPub` – *tcs/cache.c*

**Synopsis:**

```
TSS_RESULT getBlobByPub(TCPA_STORE_PUBKEY *pub, TCPA_KEY
**ret_key)
```

**Description:**

`getBlobByPub` searches through the TCS key cache for an entry with a public key matching `pub`. If found, a reference to the key's blob is copied into `*ret_key`.

**Return Values:**

On success, `*ret_key` is set and `TSS_SUCCESS` is returned.

**Synchronization:**

The TCS key cache lock is held while the list is being searched.

**Errors:**

If the requested key cannot be found, `TCS_E_FAIL` is returned.

`getBlobBySlot` – *tcs/cache.c*

**Synopsis:**

```
TSS_RESULT getBlobBySlot(TCPA_KEY_HANDLE tpm_handle,
TCPA_KEY **ret_key)
```

**Description:**

`getBlobByPub` searches through the TCS key cache for an entry with a `TCPA_KEY_HANDLE` matching `tpm_handle`. If found, a reference to the key's blob is copied into `*ret_key`.

**Return Values:**

On success, `*ret_key` is set and `TSS_SUCCESS` is returned.

**Synchronization:**

The TCS key cache lock is held while the list is being searched.

**Errors:**

If the requested `TCPA_KEY_HANDLE` cannot be found, `TCS_E_FAIL` is returned.

`getAnyHandleBySlot` – *tcs/cache.c*

**Synopsis:**

```
TCS_KEY_HANDLE getAnyHandleBySlot(TCPA_KEY_HANDLE
tpm_handle)
```

**Description:**

`getAnyHandleBySlot` searches through the TCS key cache for an entry with a `TCPA_KEY_HANDLE` matching `tpm_handle`. If found, the entry's TCS key handle is returned.

**Return Values:**

On success, the requested entry's TCS key handle is returned.

**Synchronization:**

The TCS key cache lock is held while the list is being searched.

**Errors:**

If no cached key matches `tpm_handle`, `NULL_TCS_HANDLE` is returned.

`getKeyHandleByUuid` – *tcs/cache.c*

**Synopsis:**

```
TCS_KEY_HANDLE getKeyHandleByUuid(TSS_UUID *uuid)
```

**Description:**

`getKeyHandleByUuid` searches through the TCS key cache for an entry with a UUID matching `uuid`. If found, the entry's TCS key handle is returned.

**Return Values:**

On success, the requested entry's TCS key handle is returned.

**Synchronization:**

The TCS key cache lock is held while the list is being searched.

**Errors:**

If no cached key matches `uuid`, `NULL_TCS_HANDLE` is returned.

`refreshTimeStampBySlot` – *tcs/cache.c*

**Synopsis:**

```
TSS_RESULT refreshTimeStampBySlot(TCPA_KEY_HANDLE
tpm_handle)
```

**Description:**

`refreshTimeStampBySlot` searches through the TCS key cache for an entry with a key slot of `tpm_handle`. If found, the entry's time stamp is updated.

**Return Values:**

On success, the requested entry's time stamp is updated.

**Synchronization:**

The TCS key cache lock is held while the list is being searched.

**Errors:**

If no cached key has a key slot matching `tpm_handle`, `TCS_E_FAIL` is returned.

## 3.6 Miscellaneous

The miscellaneous functions and data structures listed below do not fall into any of the categories above, but are worth mentioning.

### 3.6.0 Functions

`getTPMMetrics()` - *tcs/cxt.c*

Synopsis:

```
TSS_RESULT getTPMMetrics()
```

Description:

`getTPMMetrics` queries the TPM on TCS startup to find out information about it. The `TPM_GetCapability` command is issued to the TPM to get values for the number of PCR's it contains, whether it supports the `Save/LoadKeyContext` and `Save/Load AuthContext` ordinals, etc. These values are then used to create the necessary data structures managed by the TCS.

Return Values:

On success, `getTPMMetrics` returns `TSS_SUCCESS`.

Synchronization:

None.

Errors:

If a particular query to the TPM fails, a default value is used in its place if possible. If the TPM cannot be queried, the error value is passed back to the caller.

## <<3.7 TCSD Configuration File

The TCSD will read a configuration file at startup time and again when it receives the `SIGHUP` signal. The configuration file will contain the following parameters:

1. The port that the TCSD will listen on for remote connections
2. The default behavior for secret handling inside policy objects
3. The maximum number of threads allowed to be running in the TCSD simultaneously.
4. The location and names of the persistent storage files

## 4.0 Portability

In order to make this document easier to navigate, all portability related items will be echoed in this section.

As stated in the TSS High Level Design document, the LTC TSS will make every attempt to use only POSIX header files and interfaces in order to be as portable as possible.

Crypto:

Adding a new crypto implementation should be fairly straightforward:

- 1) Create a new directory, e.g. `src/tspi/crypto/myCrypto`.
- 2) Inside this directory, add a file named `'crypto.c'` which implements the functions



in section 2.3.0

3) Add a check in `configure.in` for your crypto library and headers (see the openssl section of `configure.in` for an example). Make sure that the build system sets the variable “CRYPTO\_PACKAGE” to the name of the directory you created in step 1. At build time, `src/tspi/crypto/$CRYPTO_PACKAGE/crypto.c` will be built.>>

One hurdle to implementing the cryptographic operations needed to interact with a TPM is the fact that the TPM requires the OAEP padding parameter of RSA encrypt/decrypt operations to be set to the NULL terminated string “TCPA”.

GUI:

In the same way that the cryptographic implementations are pluggable, the GUI components will be as well. `DisplayNewPINWindow` and `DisplayPINWindow` will be the abstraction point here (these are the two functions called by `popup_GetSecret`). In order to add a new type of GUI component to drive the popup messages, do the following:

- 1) Create a new directory, e.g. `src/tspi/gui/myGui`.
- 2) Inside this directory, create the files `'main.c'`, `'support.c'`, `'interface.c'` and `'callbacks.c'` which implement `DisplayNewPINWindow()` and `DisplayPINWindow()`.
- 3) Add a check in `configure.in` for your GUI library and headers (see the GTK section of `configure.in` for an example). Make sure that the build system sets the variable “GUI\_PATH” to the name of the directory you created in step 1. At build time, `src/tspi/gui/$GUI_PATH/*.c` will be built.

Memory Pinning:

The TSS 1.1 API specification states that secrets should be pinned in memory and zeroed out prior to being freed. Since memory pinning is an OS specific operation, the `pin_memory()` function will need to be reimplemented.

Internationalization:

NLS support will be enabled through the gettext package. Since gettext may not be available for all OS's, all gettext calls will be made part of the logging macros, found in `src/include/log.h`. This should make changing the library used for internationalization easier. Also, visit `configure.in` and `src/tspi/Makefile.am` for examples of adding new internationalization library support at build time.

## 5.0 References

1. *TCG Software Stack (TSS) Specification, Version 1.1*. Trusted Computing Group, Incorporated. August 20, 2003. (C) 2003.  
[http://www.trustedcomputinggroup.org/downloads/TSS\\_Version\\_\\_1.1.pdf](http://www.trustedcomputinggroup.org/downloads/TSS_Version__1.1.pdf)
2. *Trusted Computing Platform Alliance (TCPA) Main Specification Version 1.1b*. Trusted Computing Group, Incorporated. (C) 2003  
[http://www.trustedcomputinggroup.org/downloads/Main\\_TCG\\_Arcitecture\\_v1\\_1b.zip](http://www.trustedcomputinggroup.org/downloads/Main_TCG_Arcitecture_v1_1b.zip)>>