GLOBALPLATFORM®
THE STANDARD FOR SECURE DIGITAL SERVICES AND DEVICES

GlobalPlatform Technology

TEE Management Framework (TMF) including ASN.1 Profile

Version 1.0.1.12 (target v1.1)

Public Review

April 2020

Document Reference: GPD_SPE_120

THIS SPECIFICATION OR OTHER WORK PRODUCT IS BEING OFFERED WITHOUT ANY WARRANTY WHATSOEVER, AND IN PARTICULAR, ANY WARRANTY OF NON-INFRINGEMENT IS EXPRESSLY DISCLAIMED. ANY IMPLEMENTATION OF THIS SPECIFICATION OR OTHER WORK PRODUCT SHALL BE MADE ENTIRELY AT THE IMPLEMENTER'S OWN RISK, AND NEITHER THE COMPANY, NOR ANY OF ITS MEMBERS OR SUBMITTERS, SHALL HAVE ANY LIABILITY WHATSOEVER TO ANY IMPLEMENTER OR THIRD PARTY FOR ANY DAMAGES OF ANY NATURE WHATSOEVER DIRECTLY OR INDIRECTLY ARISING FROM THE IMPLEMENTATION OF THIS SPECIFICATION OR OTHER WORK PRODUCT.

Contents

Figures

Tables

1 Introduction

This document describes the security model for the administration of Trusted Execution Environments (TEEs) and the administration of Trusted Applications (TAs) and corresponding Security Domains (SDs). In particular, it presents the roles and responsibilities of the different stakeholders involved in the administration of a TEE and TA, the life cycle of administrated entities, and the mechanisms involved in administration operations.

This document also describes an ASN.1 implementation (referred to as the ASN.1 Profile). Other profiles may be defined in separate specifications.

If you are implementing this specification and you think it is not clear on something:

1. **Check with a colleague.**

And if that fails:

2. **Contact GlobalPlatform at TEE-issues-GPD_SPE_120_v1.1@globalplatform.org**

1.1 Audience

This specification is intended for:

- Trusted Execution Environment Implementers (TEE Implementers)
- Trusted Execution Environment Issuers (TEE Issuers)
- Trusted Application Providers (TA Providers)
- Service Providers (SP)
- Trusted Service Managers (TSM)

1.2 IPR Disclaimer

Attention is drawn to the possibility that some of the elements of this GlobalPlatform specification or other work product may be the subject of intellectual property rights (IPR) held by GlobalPlatform members or others. For additional information regarding any such IPR that have been brought to the attention of GlobalPlatform, please visit https://globalplatform.org/specifications/ip-disclaimers/. GlobalPlatform shall not be held responsible for identifying any or all such IPR, and takes no position concerning the possible existence or the evidence, validity, or scope of any such IPR.

1.3 References

The tables below list references applicable to this specification. The latest version of each reference applies unless a publication date or version is explicitly stated.

Table 1-1: Normative References

Standard / Specification	Description	Ref
GPD_SPE_007	GlobalPlatform Technology TEE Client API Specification	[TEE Client]
GPD_SPE_009	GlobalPlatform Technology TEE System Architecture	[TEE Arch]
GPD_SPE_010	GlobalPlatform Technology TEE Internal Core API Specification	[TEE Core API]
GPD_SPE_025	GlobalPlatform Technology TEE TA Debug Specification	[TEE TA Debug]
GPD_SPE_121	GlobalPlatform Technology TMF: Symmetric Cryptography Security Layer	[TMF Symmetric]
GPD_SPE_122	GlobalPlatform Technology TMF: Asymmetric Cryptography Security Layer	[TMF Asymmetric]
GPD_SPE_123	GlobalPlatform Technology TMF: Open Trust Protocol (OTrP) Profile	[TMF OTrP]
IETF RFC 2119	Key words for use in RFCs to Indicate Requirement Levels	[RFC 2119]
IETF RFC 4122	A Universally Unique IDentifier (UUID) URN Namespace	[UUID]
ISO/IEC 14977	Information technology – Syntactic meta language – Extended Backus-Naur Form	[Backus-Naur]
ITU-T X.680	Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation	[ASN.1]
ITU-T X.682	Information technology – Abstract Syntax Notation One (ASN.1): Constraint specification	[ASN.1 Constraint]
ITU-T X.690	Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)	[ASN.1 Encoding]

Table 1-2: Informative References

Standard / Specification	Description	Ref
GP_REQ_025	GlobalPlatform Security Task Force Root of Trust Definitions and Requirements	[RoT]
ISO/IEC 19505	Unified Modeling Language	[UML]

1.4 Terminology and Definitions

The following meanings apply to SHALL, SHALL NOT, MUST, MUST NOT, SHOULD, SHOULD NOT, and MAY in this document (refer to [RFC 2119]):

- **SHALL** indicates an absolute requirement, as does **MUST**.
- **SHALL NOT** indicates an absolute prohibition, as does **MUST NOT**.
- **SHOULD** and **SHOULD NOT** indicate recommendations.
- **MAY** indicates an option.

Selected terms used in this document are included in the following table. Additional terms are defined in [TEE Core API].

Table 1-3: Terminology and Definitions

Term	Definition
Actor	A stakeholder performing a specific role in a GlobalPlatform-compliant environment. These stakeholders may take the form of card issuers, application developers, personalization bureaus, etc.
Authority	An Actor that grants permission to perform a specific set of actions. An Authority is represented in the device by a Security Domain.
Authorization Token	In the TEE administration security model, a piece of structured information, emitted by an Authority, that grants some rights to be able to execute an administration operation. Its structure includes a data-integrity mechanism and means to authenticate the issuer. (For more information, see section 5.3.)
Authorizing Security Domain (SD-A)	The Security Domain responsible for validating an administrative operation by verifying an Authorization Token and/or strongly authenticating the remote authority that submitted the administration operation. (For more information, see Chapter 5).
Bootstrap Domain (BD)	A non GlobalPlatform compliant domain capable of instantiating a GlobalPlatform domain. (For more information, see section 4.1.5.)
Client Application (CA)	An application running outside of the Trusted Execution Environment (TEE) making use of the TEE Client API to access facilities provided by Trusted Applications inside the TEE. Contrast *Trusted Application*.
Distinguished Encoding Rules (DER)	A set of rules specified by [ASN.1 Encoding] to encode the value of a type structure defined by [ASN.1].
Extended Backus-Naur Form (EBNF)	A family of metasyntax notations, any of which can be used to express a context-free grammar; defined in ISO/IEC 14977 ([Backus-Naur]).
Panic	An exception that kills a whole TA instance. See [TEE Core API] section 2.3.3 for full definition.
Performing Security Domain (SD-P)	The recipient Security Domain of the operation command; that is, the Security Domain that performs the operation.
Privileged Operation	An operation which requires authorization by an Authority.

Term	Definition
Regular Execution Environment (REE)	An Execution Environment comprising at least one Regular OS and all other components of the device (SoCs, other discrete components, firmware, and software) which execute, host, and support the Regular OS (excluding any Secure Components included in the device). From the viewpoint of a Secure Component, everything in the REE is considered untrusted, though from the Regular OS point of view there may be internal trust structures. (Formerly referred to as a *Rich Execution Environment (REE)*.) Contrast *Trusted Execution Environment*.
Regular OS	An OS executing in a Regular Execution Environment. May be anything from a large OS such as Linux down to a minimal set of statically linked libraries providing services such as a TCP/IP stack. (Formerly referred to as a *Rich OS* or *Device OS*.) Contrast *Trusted OS*.
Root of Trust	A computing engine, code, and possibly data, all co-located on the same platform. It provides security services (as discussed in [RoT]). No ancestor entity is able to provide a trustable attestation (in Digest or other form) for the initial code and data state of the Root of Trust.
Root Security Domain (rSD)	A Security Domain over which other Authorities have very limited control (see section 4.1.3.3).
Security Domain (SD)	An on-device representative of an Authority in the TEE Management Framework security model. Security Domains are responsible for the control of administration operations. SDs are used to perform the provisioning of TEE properties and to manage the life cycle of Trusted Applications and SDs associated with them. (For more information, see section 4.1.)
Security Layer	A layer providing some level of isolation and/or validation of information carried over a transport layer.
Target Security Domain (SD-T)	The Security Domain on which the operation is being performed or the *Target Security Domain* parameter of specific commands.
TEE Factory Reset	The TEE factory reset is a privileged operation that moves the TEE and its assets to a notional “factory” state.
TEE Management Framework (TMF)	A security model for administration of Trusted Execution Environments (TEEs) and for administration and life cycle management of Trusted Applications (TAs) and corresponding Security Domains (SDs).
Trusted Application (TA)	An application running inside the Trusted Execution Environment that provides security related functionality to Client Applications outside of the TEE or to other Trusted Applications inside the TEE. Contrast *Client Application*.

Term	Definition
Trusted Execution Environment (TEE)	An Execution Environment that runs alongside but isolated from an REE. A TEE has security capabilities and meets certain security-related requirements: It protects TEE assets against a set of defined threats which include general software attacks as well as some hardware attacks, and defines rigid safeguards as to data and functions that a program can access. There are multiple technologies that can be used to implement a TEE, and the level of security achieved varies accordingly. Contrast *Regular Execution Environment.*
Trusted OS	An OS executing in a Secure Component. Contrast *Regular OS*.
Universally Unique Identifier (UUID)	An identifier as specified in RFC 4122 [UUID].
Unprivileged operation	An operation which does not require authorization by an Authority.

1.5 Abbreviations and Notations

Selected abbreviations and notations used in this document are included in the following table. Additional abbreviations and notations are defined in [TEE Core API].

Table 1-4: Abbreviations and Notations

Abbreviation / Notation	Meaning
BD	Bootstrap Domain
CA	Client Application
DER	Distinguished Encoding Rules
DLM	Debug Log Message
EBNF	Extended Backus-Naur Form
ECC	Elliptic Curve Cryptography
OTrP	Open Trust Protocol
PMR	Post Mortem Reporting
REE	Regular Execution Environment
RFU	Reserved for Future Use
rSD	Root Security Domain (or root SD)
SD	Security Domain
SD-A	Authorizing Security Domain
SD-P	Performing Security Domain
SD-T	Target Security Domain
TA	Trusted Application
TEE	Trusted Execution Environment

Abbreviation / Notation	Meaning
TLV	Tag, Length, Value
TMF	TEE Management Framework
UUID	Universally Unique Identifier

1.6 Revision History

GlobalPlatform technical documents numbered *n*.0 are major releases. Those numbered *n*.1, *n*.2, etc., are minor releases where changes typically introduce supplementary items that do not impact backward compatibility or interoperability of the specifications. Those numbered *n.n*.1, *n.n*.2, etc., are maintenance releases that incorporate errata and precisions; all non-trivial changes are indicated, often with revision marks.

Table 1-5: Revision History

Date	Version	Description
November 2016	1.0	Public Release as *TEE Management Framework*
May 2019	1.0.1	Public Release as *TEE Management Framework including ASN.1 Profile* Simultaneously, GlobalPlatform released an alternative for implementation of TEE management: *TEE Management Framework: Open Trust Protocol (OTrP) Profile*, GPD_SPE_123 Revisions to this document include: • Identified details specific to the ASN.1 Profile • Updated to reflect changes to [TEE Core API], including: o The use of the property `gpd.tee.internalCore.version` as well as the deprecated property `gpd.tee.apiversion`. o Adjustments to the identifiers of selected cryptographic algorithms. • Clarifications • Adjustments to examples
October 2019	1.0.1.3	Committee Review
January 2020	1.0.1.5	Member Review

Date	Version	Description
April 2020	1.0.1.12	Public Review • Added Update TA and Data for atomic updates (sections 6.2.6 and 8.4.6, Update TA and Data). • Added Fetch Object to permit retrieval of public keys belonging to an SD (sections 6.4.4 and 8.6.4, Fetch Object). • Added Get TA Definition 1 (sections 6.6.5 and 8.8.5) which returns a versioned structure enabling more information to be returned and which returns the version number of the TA as an integer. • Added section 8.3.4, CryptoProcID, making the definition of `CryptoProcID` normative (based on the prior informative section B.2, SD Installation: Examples of Cryptographic Procedures).
TBD	1.1	Public Release

2 General Considerations (Informative)

2.1 Scope

The GlobalPlatform TEE Management Framework (TMF) defines standard methods to administer the TEE from outside of the TEE. Such administration includes data and key provisioning, Security Domain management, Trusted Application management, audit, and overall TEE management.

The framework enables this by defining protocols and interfaces accessed either through the GlobalPlatform TEE Client API ([TEE Client]) or via extensions to the TEE Internal Core API ([TEE Core API]).

Administration operations may be initiated by both on-line and off-line Actors. An off-line Actor may be inside the device itself, such as a component of the Regular Execution Environment (REE), or even inside the TEE. For example, an REE application (via a Client Application (CA)) or a TA might issue a Factory Reset command.

This specification also describes an ASN.1 implementation, referred to as the ASN.1 Profile. Other profiles may be defined in separate specifications.

2.2 Authorities

Various Actors may be involved in the administration of a Trusted Execution Environment and the Trusted Applications – e.g. TEE Implementer, TEE Issuer, TA Providers, or Trusted Service Managers. The responsibilities of each of these Actors and the administration operations they are allowed to perform may vary from one business domain to another.

For example, the issuer of the TEE might be responsible for the administration of the TEE life cycle and also may act as an application provider. In another business context, several partners may decide to delegate the administration responsibilities to a trusted third party or may combine responsibilities to achieve an administration operation.

The security model for TEE administration identifies Authorities, and allows one or multiple Authorities to be represented in the TEE. Each Authority is assigned a set of privileges which precisely defines the set of administration operations it is allowed to perform.

2.3 Nature of this Specification

This specification gives implementers a toolbox with which to fulfill their business needs, but does not force a particular business model on an implementation. As such, a wide variety of systems can be implemented, from those with only one SD managing data for a fixed pool of TAs, to a device with many root SDs, each with a tree of child SDs managing many TAs.

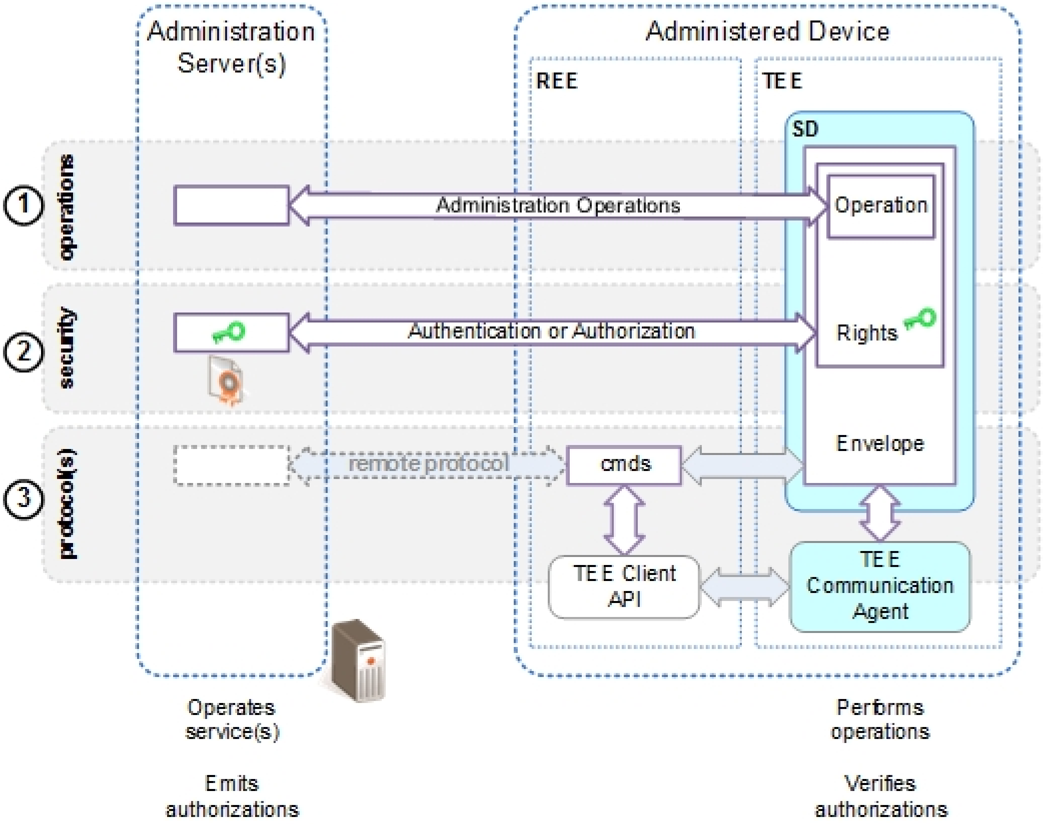
2.4 System Overview

The TMF provides means to securely administrate Trusted Applications and Security Domains in a Trusted Execution Environment. The three layers described in this document pertain to the ASN.1 Profile; other profiles may use part or all of any of these layers.

1. **Administration Operations**
    - Defines the set of supported operations to manage Trusted Applications and Security Domains, the conditions of use, and the detailed behavior of each operation.
2. **Security Model**
    - Defines who the Actors are and how the different business relationships and responsibilities can be mapped on the concept of Security Domains with privileges and associations.
    - Defines the security mechanisms used to authenticate the entities establishing a communication channel, to secure the communication, and to authorize administration operations to be performed by Security Domains.
    - Defines schemes for key and data provisioning and describes the associated key management.
3. **Protocol(s)**
    - Defines the command set (over the TEE Client API [TEE Client] and the optional support of the TEE Internal Core API [TEE Core API]) to be used to perform administration operations.
    - Defines the encoding of administration operations.
    - Recommends the use of suitable protocols to establish a secure session with a Security Domain.

Note: The definition of a remote protocol to transport administration commands from a remote server to the administrated device (i.e. the part outside the TEE) is out of scope of this version of the specification.

Figure 2-1: TEE Management Framework Structure



Chapter 5 describes the principles of Authentication and Authorization required by the security model.

Note: While this specification defines the main characteristics of this model in Chapter 5 and section 8.2 (Security Layer description), GlobalPlatform will publish future specifications relating to specific configurations of this model (e.g. use cases for Authorization Tokens, definitions of Security Layers, etc.).

Chapter 6 describes the administration operations in detail.

Chapter 8 defines the full set of administration commands and their encoding.

2.5 Resources

While in a perfect world a device may have infinite resources, we do not live in a perfect world. Therefore, a GlobalPlatform TMF compliant device may at any time return codes or states indicative of a lack of resources. Thus, for example, while a Security Domain may have the capability to authorize an infinite depth of Security Domain installations, the actual achievable depth may actually be one.

3 General Considerations (Normative)

3.1 Endianness Convention

All the data structures described in this specification are encoded according to big-endian ordering. Whereas it is implicitly the case for any TLV structures introduced by Chapter 7 according to the ITU.X 690 standard ([ASN.1 Encoding]), it must also be the case for the encoding of UUID v5 values defined by section 5.6.

3.2 Cryptographic Keys and Algorithm Usage

Most of the administration operations described in this specification rely on the use of strong cryptographic algorithms. These algorithms are dependent on the TEE implementation (e.g. elliptic curve algorithms may or may not be supported). Nevertheless, their modes of usage (encryption/decryption, digest, signature computation or verification), their algorithm identifier values, and the key strength required by any cryptographic operations SHALL be compliant with the characteristics and definitions of algorithms and key objects specified in [TEE Core API].

Depending on the context of cryptographic operations, this specification requires the usage of particular algorithms with some recommendations for the keys' strength, and suggests additional algorithms that an implementation could optionally support (see Table A-8). As specified in [TEE Core API], a TEE vendor-specific implementation MAY define and use its own algorithms to fulfill the requirements imposed by specific market needs or a certification process.

4 Security Model for Administration

The goals of the security model for administration are:

- to provide means to manage the Trusted Execution Environment (TEE), Security Domains (SD), and Trusted Applications (TA),
- to ensure the security and integrity of these entities,
- to enable data confidentiality,
- to provide a scalable model allowing deployments involving a unique Actor or multiple Actors,
- and to enforce the security policy of each Actor while preserving its assets.

To ensure the security and integrity of these entities, the TMF code implementation on the device is a Trusted OS Component (see [TEE Arch]), or composed from a group of such components. As such it inherits the same security requirements as other Trusted OS Components.

4.1 Security Domains

Security Domains are components of the TEE that SHALL be accessible from the TEE Client API [TEE Client] and that MAY be accessible from the Internal Client API [TEE Core API] (see Chapter 8).

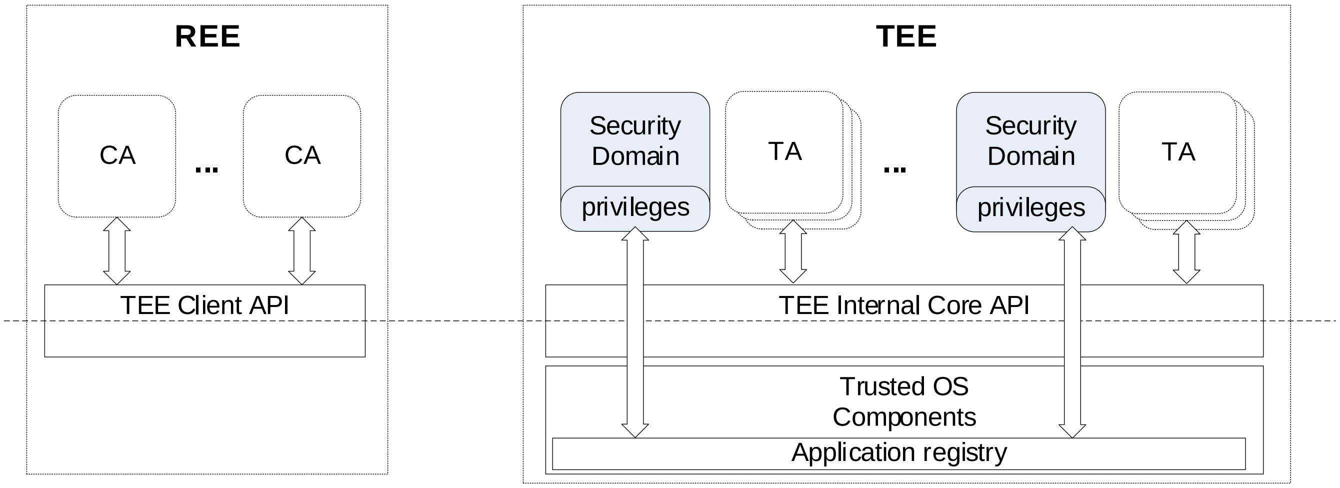
The way the code implementing the behavior of Security Domains is integrated into the TEE (pre-installed, dynamically loaded, etc.) is out of scope of this version of the specification. Nevertheless, instantiation of Security Domains (basically creating a set of privileges and storing a key set) is performed using commands described in this specification.

Security Domains have administration and provisioning responsibilities. Their privileges give them access to internal TEE resources, particularly the application registry. They hold cryptographic keys whose use includes, but is not limited to, securely initiating the execution and authorization of administration operations. The set of administration operations that a Security Domain is capable of depends on its privileges (see section 4.1.3).

A TEE that offers administration capabilities according to this document SHALL contain at least one root Security Domain compliant with this specification. It can be the on-device representative of the TEE issuer, the device manufacturer, the Mobile Network Operator, or whatever Authority has been elected for administrative operations, based on pre-established business considerations between the different stakeholders.

It is permissible for a device to have multiple root Security Domains, thus enabling different Authorities to have independent hierarchies.

Figure 4-1: Architecture Overview



4.1.1 Security Domain Associations

Security Domains and Trusted Applications are organized in hierarchical tree structures (aka “hierarchies”) where it is possible to have multiple independent roots.

Associations between Security Domains and Trusted Applications represent hierarchies where a child is controlled by its direct parent and/or by ancestors in the path starting from the child and going to the root of the tree structure.

The *administration privileges* of each node in this path define the type of control (i.e. the list of administration operations) that the node can perform on its descendants.

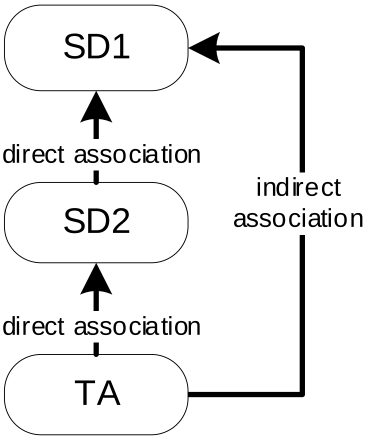
Only Security Domains have administration privileges (see section 4.1.3).

By definition, Trusted Applications are nodes with no privileges. They constitute the leaves of the tree structure and cannot directly perform any administrative operations.

Associations can be *direct* or *indirect*:

- A TA or an SD is *directly associated* with an SD that is its immediate parent.
  - Direct associations are constructed using the *Install TA* (section 6.2.1) and *Install SD* (section 6.3.1) operations that associate a TA or an SD, respectively, with a single parent SD.
  - A Security Domain that is the root of a hierarchy has no immediate parent SD.
- A TA or an SD is *indirectly associated* with an SD that is not its immediate parent but is any other ancestor in the path going from the root of the tree structure to the TA or SD.

Figure 4-2: Example of Direct and Indirect Associations



4.1.2 Retrieving the UUID of a Parent Security Domain

Trusted Applications

A Trusted Application can retrieve the UUID of its direct parent Security Domain by retrieving the `gpd.ta.parentSD` property using the `TEE_PROPSET_CURRENT_TA` pseudo-handle.

```
TEE_GetPropertyAsUUID(TEE_PROPSET_CURRENT_TA, "gpd.ta.parentSD", &value)
```

If the Trusted Application is used by another Trusted Application, herein called the client TA, the TA can retrieve the UUID of the client TA's SD by retrieving the `gpd.client.parentSD` property using the `TEE_PROPSET_CURRENT_CLIENT` pseudo-handle.

```
TEE_GetPropertyAsUUID(TEE_PROPSET_CURRENT_CLIENT, "gpd.client.parentSD", &value)
```

The `gpd.client.parentSD` property is available only if the client is a Trusted Application. That is if .the property `gpd.client.identity` associated with the client TA is equal to `TEE_LOGIN_TRUSTED_APP` – see [TEE Core API] section 4.6. Otherwise `TEE_GetPropertyAsUUID(…)` returns the `TEE_ERROR_ITEM_NOT_FOUND` error code.

Security Domains

By examining the structure returned by the *Get SD Definition* operation (see sections 6.6.2, 8.8.2, and 9.2.2), a remote entity can retrieve the UUID of an SD's parent SD.

4.1.3 Security Domain Privileges

Every Security Domain is configured to permit only those administration operations on the TEE that are appropriate to the role of the Authority that the SD represents. The list of privileges granted to a security domain constrains the operations that may be performed.

Each privilege is defined by a *function* that determines the list of administration operations granted by this privilege (see section 4.1.3.1).

The context of execution of an administration operation in the TEE is determined by:

- The Security Domain (SD-A) that authorizes the operation (see section 5.2)
- The Security Domain (SD-P) that performs the operation, i.e. the SD to which the administration operation command is passed during an administration session with the TEE (see Chapter 6)
- The target node (a Security Domain (SD-T) or a Trusted Application) on which the operation is performed

SD-P SHALL be SD-A itself or SHALL be directly or indirectly associated with SD-A.

The scope of control of SD-A's privileges is defined by the set of rules (as specified in sections 4.1.3.2 and 4.1.3.3) that SHALL correspond to the administration operations applying to any target node of the TEE.

4.1.3.1 Privilege Functions and Associated Operations

Each Security Domain privilege function defines a list of privileged administration operations.

Table 4-1 lists the privilege functions, provides the privilege name value of each, and discusses the associated administration operations. Table 4-2 lists specific operations associated with each privilege.

Table 4-1: Privilege Functions

Privilege Function	Privilege Name Value (*)	Privileged Administration Operations
TEE Management	`gpd.privilege.` `teeManagement`	A Security Domain with this privilege is allowed to: • Manage the Trusted Execution Environment life cycle (*Lock TEE* and *Unlock TEE* operations – see sections 6.5.1 and 6.5.2). • Modify TEE properties (*Store TEE Property* operation – see section 6.5.3). • Invoke a TEE factory reset (*Factory Reset* operation – see section 6.5.4).
TA Management	`gpd.privilege.` `taManagement`	A Security Domain with this privilege is allowed to: • Install new Trusted Applications (*Install TA* operation – see section 6.2.1). • Uninstall or update existing Trusted Applications (*Uninstall TA* and *Update TA* operations – see sections 6.2.2 and 6.2.3). • Manage the usage of Trusted Applications (*Lock TA* and *Unlock TA* operations – see sections 6.2.4 and 6.2.5).
SD Management	`gpd.privilege.` `sdManagement`	A Security Domain with this privilege is allowed to: • Install new Security Domains (*Install SD* operation – see section 6.3.1) that are not root Security Domains (see root SD in section 4.1.3.3). • Uninstall existing Security Domains (*Uninstall SD* operation – see section 6.3.2) that are not root Security Domains (see root SD in section 4.1.3.3). • Block or unblock the operations of any Security Domain (*Block SD* and *Unblock SD* operations – see sections 6.3.3 and 6.3.4). • Restrict or unrestrict the usage of Security Domain keys/data (*Restrict SD* and *Unrestrict SD* operations – see sections 6.3.5 and 6.3.6).
rSD Management	`gpd.privilege.` `rsdManagement`	A security Domain with this privilege is allowed to: • Install new root Security Domains (*Install SD* operation – section 6.3.1; root SD – section 4.1.3.3). • Uninstall a root Security Domain (*Uninstall SD* operation – section 6.3.2; root SD – section 4.1.3.3).

Privilege Function	**Privilege Name Value (*)**	**Privileged Administration Operations**
TA Personalization	`gpd.privilege.taPersonalization`	A Security Domain with this privilege is allowed to: • Personalize keys and data of Trusted Applications (*Store Data* and *Delete Data* operations – see sections 6.4.1 and 6.4.2). • Manage the usage of Trusted Applications (*Lock TA* and *Unlock TA* operations – see sections 6.2.4 and 6.2.5). • List the personalized keys and data of Trusted Applications (*List Objects* operation – see section 6.4.3).
SD Personalization	`gpd.privilege.sdPersonalization`	A Security Domain with this privilege is allowed to: • Personalize keys and data of Security Domains (*Store Data* and *Delete Data* operations – see sections 6.4.1 and 6.4.2). • Restrict or unrestrict the usage of Security Domain keys/data (*Restrict SD* and *Unrestrict SD* operations – see sections 6.3.5 and 6.3.6). • List the personalized keys and data of Security Domains (*List Objects* operation – see section 6.4.3).

(*) Privilege names are encoded as a byte value as defined by the `privilegeID` value in Table 8-18.

Table 4-2: List of the Privileged Operations Associated with the SD Privileges

	Management				Personalization	
	TEE	**TA**	**SD**	**rSD**	**TA**	**SD**
Lock TEE	✓	-	-	-	-	-
Unlock TEE	✓	-	-	-	-	-
Store TEE Property	✓	-	-	-	-	-
Factory Reset	✓	-	-	-	-	-
Install SD	-	-	✓	✓	-	-
Uninstall SD	-	-	✓	✓	-	-
Block SD	-	-	✓	-	-	-
Unblock SD	-	-	✓	-	-	-
Restrict SD	-	-	✓	-	-	✓
Unrestrict SD	-	-	✓	-	-	✓
Install TA	-	✓	-	-	-	-
Uninstall TA	-	✓	-	-	-	-
Update TA	-	✓	-	-	-	-
Lock TA	-	✓	-	-	✓	-
Unlock TA	-	✓	-	-	✓	-
Store Data	-	-	-	-	✓	✓
Delete Data	-	-	-	-	✓	✓
List Objects	-	-	-	-	✓	✓

All operations associated with a privilege SHALL be supported by a compliant implementation offering this privilege.

The details of all these operations are described in Chapter 6, Administration Operations. The commands that trigger these operations are described in Chapter 8, Administration Commands Encoding.

4.1.3.2 Scope of Control of SD-A Privileges

In the context of execution of administration operations, the rules of control of SD-A's privileges over any SD or TA apply as follows.

Table 4-3: Scope of Control of SD-A Privileges

If SD-A has this privilege ...	... then operations are permitted that:
`gpd.privilege.taManagement`	• Install any new TA directly associated with a specified target SD (SD-T) where SD-T is SD-A or any SD directly or indirectly associated with SD-A. • Uninstall, update, lock, and unlock any TA directly or indirectly associated with SD-A.
`gpd.privilege.taPersonalization`	• Store, delete, or list objects in the personalization data storage (see section 5.5) of any TA directly or indirectly associated with SD-A. • Lock and unlock any TA directly or indirectly associated with SD-A.
`gpd.privilege.sdManagement`	• Install any new SD directly associated with a specified target SD (SD-T) where SD-T is SD-A or any SD directly or indirectly associated with SD-A. • Uninstall, restrict, and unrestrict SD-A itself and any SD directly or indirectly associated with SD-A. • Block and unblock any SD directly or indirectly associated with SD-A.
`gpd.privilege.rsdManagement`	• Install any new rSD directly associated with a specified target SD (SD-T) where SD-T is SD-A or any SD directly or indirectly associated with SD-A. • Uninstall SD-A itself when SD-A is an rSD. • Uninstall any rSD directly or indirectly associated with SD-A.
`gpd.privilege.sdPersonalization`	• Store, delete, or list objects in the private storage (see section 5.5) of both SD-A and any SD directly or indirectly associated with SD-A. • Restrict and unrestrict SD-A itself and any SD directly or indirectly associated with SD-A.

> ***Warning:*** These rules do not take precedence over the rules limiting the control of SD-A over root Security Domains as defined in section 4.1.3.3. For example, SD-A having the `gpd.privilege.taManagement` privilege SHALL NOT permit operations that install a new TA directly associated with SD-T when SD-T is an rSD or if there is an rSD in the path of associated SDs from SD-A to SD-T.

4.1.3.3 Root Security Domains

A root Security Domain (rSD) is an SD that is internally flagged by the TEE as having the `gpd.sd.isRootSD` property.

- An SD SHALL be considered a root SD when the SD is successfully installed using the Install SD command (see section 8.5.1.1) where the `Privileges` parameter has the `isRootSD` field set to TRUE as defined in section 8.3.3.10. (This operation requires SD-A to have the `gpd.privilege.rsdManagement` privilege.)
- An SD SHALL be considered a root SD when it is installed with no parent using any method outside the scope of TMF (for example, in factory).

If an rSD has no parent, then only the rSD itself can perform administrative operations on the rSD.

If SD-A is a direct parent or any ancestor of an rSD, SD-A SHALL have strictly limited control over the rSD, as follows.

Table 4-4: Scope of Control of SD-A Privileges Regarding rSD

SD-A Privilege	Limitation
`gpd.privilege.taPersonalization`	• SD-A SHALL NOT authorize any operation enabled by this privilege on any TA directly or indirectly associated with the rSD.
`gpd.privilege.sdPersonalization`	• SD-A SHALL NOT authorize any operation enabled by this privilege on the rSD or on any of its direct or indirect Security Domains.
`gpd.privilege.taManagement`	• SD-A SHALL NOT authorize any operation enabled by this privilege on any TA directly or indirectly associated with the rSD.
`gpd.privilege.sdManagement`	• SD-A SHALL NOT authorize any operation enabled by this privilege on any Security Domain directly or indirectly associated with the rSD. ***Exception:*** SD-A MAY unblock an rSD installed with the Blocked life cycle state (see sections 4.4.1 and 4.4.4). • The rSD itself SHOULD NOT be affected by any uninstall, block, or restrict operations normally enabled by the privilege.
`gpd.privilege.teeManagement`	• The rSD MAY be affected by operations enabled by the privilege.
`gpd.privilege.rsdManagement`	• SD-A SHALL NOT authorize any operation enabled by this privilege on any Security Domain directly or indirectly associated with the rSD.

4.1.4 Trustworthiness of SDs and TAs

When a root SD is installed, it is intrinsically trusted as a part of the TEE. That root SD can be determined by validating the response to the TEE and SD audit commands (e.g. submitting the command through a Security Layer and verifying certificates and protocol information returned by the command – see sections 9.1.1, 9.1.5, and 9.2.2).

When an SD is installed by another SD, its trustworthiness is based on that of its parent, any other ancestor SDs, and the TEE itself. The trustworthiness of these elements can be determined by validating the response to the TEE and SD audit commands (see sections 9.1.1, 9.1.5, and 9.2.2); i.e. we trust an SD to manage things correctly because we trust its owner, the people who can change it, and the people who allowed it to be installed.

When a TA is installed by an SD, the TA's trustworthiness is based on that of its parent, any other ancestor SDs to that parent, and the TEE itself. The trustworthiness of these elements can be determined by validating the response to the TEE, SD, and TA audit commands (see sections 9.1.1, 9.1.5, 9.2.2, and 9.3.2); i.e. we trust a TA to do things correctly because we trust its owner and the people who can change it.

4.1.5 (Informative) Installation of Roots of SD Hierarchies: The Bootstrap Domain

In some cases, GlobalPlatform compliant TEE Security Domains may not be present when a device is first received by an end user in the field; i.e. the device may have the capability to be compliant with the GlobalPlatform TMF but initially may not have any of that compliant functionality exposed.

This version of the specification does not include any direct requirements regarding how the first TEE Security Domains are installed, either in the factory or in the field. However, to clarify the main principles of field installation, this version references the Bootstrap Domain (BD), a conceptual domain that may install its first set of GlobalPlatform compliant TEE SDs in the field.

This and other GlobalPlatform specifications do include some indirect requirements to ensure that a BD does not infringe the expected management and security boundaries.

The presence of the first GlobalPlatform TMF compliant Security Domains distinguishes a device's state in the TEE life cycle flow (see section 4.2).

The Bootstrap Domain

The Bootstrap Domain is not required to support any GlobalPlatform communication protocols or API interfaces, but may understand a subset of GlobalPlatform commands (e.g. just Install SD and Factory Reset).

A TMF-capable device does not need a Bootstrap Domain if it has at least one factory installed GlobalPlatform compliant rSD.

If a domain is GlobalPlatform compliant, then it cannot be considered a Bootstrap Domain.

A Bootstrap Domain may be internally complicated and multilayered, but the TMF regards it as a single management entity. Due to the presumed singular nature of the architecture, the Bootstrap Domain is considered to have no parents; that is, only its controlling Authority can change the BD's rights.

Trust and the Bootstrap Domain

As part of the TEE, the Bootstrap Domain, like any other factory installed domain, has the same level of intrinsic trust as the TEE. Therefore, the security guarantee of the BD's operations is the same as the security guarantee of the TEE in general.

Having said this, while the BD's functionality is proprietary, the BD's capabilities to manipulate GlobalPlatform Security Domains, Trusted Applications, and their storage systems can be considered to have the same potential set of privilege functionality restrictions as a GlobalPlatform Security Domain.

The BD's capabilities are not directly discoverable using this specification because, being proprietary, the BD does not support such operations.

4.2 Trusted Execution Environment Life Cycle

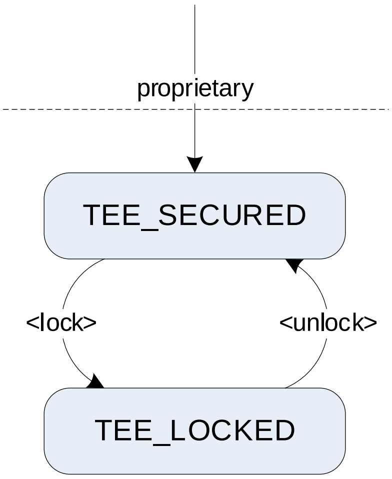
The figure below represents the life cycle of the Trusted Execution Environment.

A TEE might include several states prior to its issuance; those are considered implementation-specific and are out of scope of this specification.

Without any GlobalPlatform TMF compliant Security Domain (e.g. a Security Domain as defined in section 4.1), the device is considered to be in a proprietary management state.

Once the first GlobalPlatform Security Domain is present, the device is considered manageable through the GlobalPlatform TMF and can then be placed in either the `TEE_SECURED` or `TEE_LOCKED` life cycle state.

Figure 4-3: Trusted Execution Environment Life Cycle



4.2.1 TEE_SECURED Life Cycle State

The `TEE_SECURED` life cycle state indicates that the GlobalPlatform TEE has been configured with at least one Security Domain compliant with this specification. This Security Domain has been created and personalized with keys and data in order to perform administration operations. Supplementary Security Domains may be created and personalized.

The means and command set used to perform these operations or to switch to the `TEE_SECURED` life cycle state from any state prior to the GlobalPlatform TMF enablement are out of scope of this specification and remain implementation-specific.

In the `TEE_SECURED` life cycle state, an accessible and authorized Security Domain (SD-A) can perform any privileged administration operations according to:

- Its list of privileges
- Its current life cycle state as described by section 4.4
    - Some states keep the Security Domain inaccessible from any Client Applications or limit the Security Domain usage.

To modify the behavior of the TEE in this state, an accessible and authorized Security Domain (SD-A) having the `gpd.privilege.teeManagement` privilege may perform the following privileged administration operations:

Lock TEE

The *Lock TEE* operation switches the TEE to the `TEE_LOCKED` life cycle state, with no effect on the life cycle state of any Trusted Applications or Security Domains of the TEE.

Store TEE Property

The *Store TEE Property* operation stores new TEE properties or updates existing ones.

Warning: As storing or updating TEE properties may fundamentally change the behavior of the TEE, this operation SHOULD be performed while the TEE is in the `TEE_LOCKED` life cycle state.

Factory Reset

The *Factory Reset* operation restores the TEE and its trusted and secure storage to the last valid install package. All non-install package Trusted Applications or Security Domains with their associated trusted and secure stores are uninstalled and removed. The install package description is specified by the modifiable `gpd.tee.tmf.resetpreserved.entities` property defined in Table 8-57.

4.2.2 TEE_LOCKED Life Cycle State

The `TEE_LOCKED` life cycle state disables new accesses from Client Applications to:

- Any Trusted Application, regardless of its life cycle state (see section 4.3)
- Any accessible Security Domains with no `gpd.privilege.teeManagement` privilege

In the `TEE_LOCKED` life cycle state, any attempt by a Client Application to open a new session with a Trusted Application or with a Security Domain that does not have the `gpd.privilege.teeManagement` privilege is rejected with the `TEE_ERROR_ACCESS_DENIED` error code.

A TEE MAY provide a proprietary mechanism to exclude the `TEE_LOCKED` state from preventing execution of, and access to, a specific list of TAs.

Once the TEE is locked, ***only*** an accessible and authorized Security Domain (SD-A) having the `gpd.privilege.teeManagement` privilege is allowed to perform the following privileged administration operations:

Unlock TEE

The *Unlock TEE* operation switches the TEE to the `TEE_SECURED` life cycle state, with no effect on the life cycle state of any Trusted Applications or Security Domains of the TEE.

Store TEE Property

The *Store TEE Property* operation creates new TEE properties or updates existing ones, to modify the behavior of the TEE.

Factory Reset

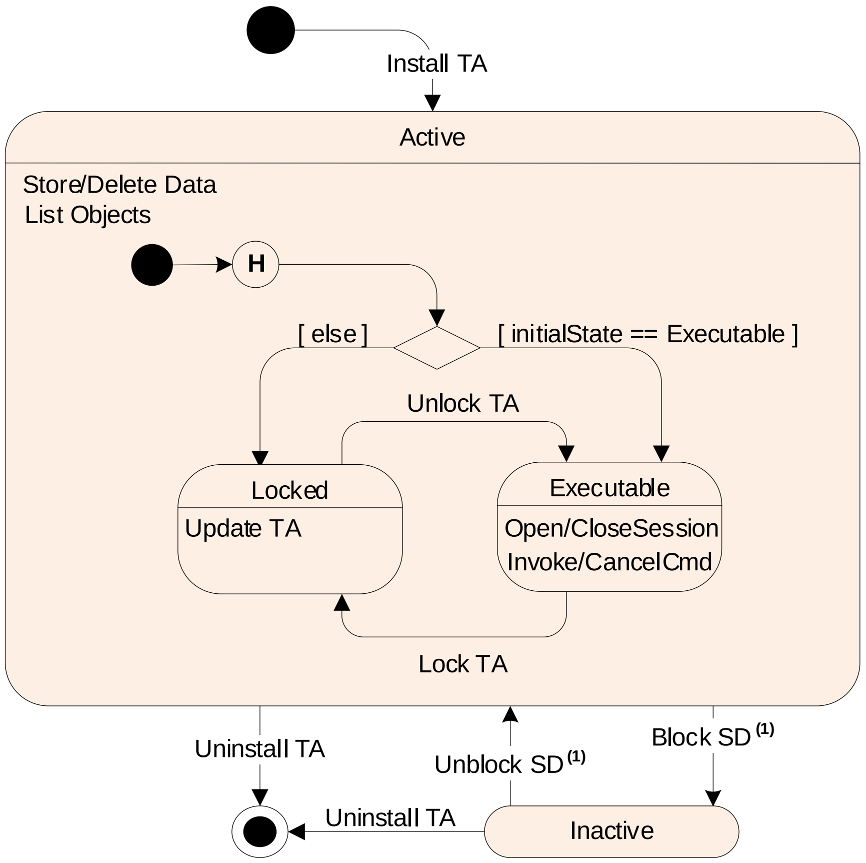
The *Factory Reset* operation restores the TEE and its trusted and secure storage to the last valid install package. All non-install package Trusted Applications or Security Domains with their associated trusted and secure stores are uninstalled and removed. The install package description is specified by the modifiable `gpd.tee.tmf.resetpreserved.entities` property defined by Table 8-57.

4.3 Trusted Application Life Cycle

4.3.1 General State Diagram

The following state diagram (based on the UML standard [UML]) represents the life cycle of a Trusted Application and lists the privileged administration operations that trigger the transition from one state to another.

Figure 4-4: Trusted Application Life Cycle



These states are persistent states, meaning that they are not affected by a power-off. The transitions from one state to another are atomic and always triggered by administration operations performed either through SD-P on the TA or (for transitions marked with $^{(1)}$) through SD-P on the TA's direct parent SD.

The states of a Trusted Application can be grouped into two categories:

Table 4-5: TA Life Cycle State Categories

Category	Description	State(s)
Active	In Executable State, the TA can be used by Client Applications. In Locked State, the TA is temporarily suspended to perform some maintenance operations	Executable Locked
Inactive	The TA is directly controlled by an SD that has been blocked (see section 4.4.4) and can neither be used nor administrated until its SD is unblocked.	Inactive

When a Trusted Application is installed or updated, its initial state can be chosen and can be either Executable (available for immediate use by Client Applications) or Locked (e.g. additional personalization required, or delayed usage). This is depicted in Figure 4-4 above by the guard expression *[initialState == Executable]* and the *[else]* alternative.

The sub-sections below describe the life cycle states in detail.

4.3.2 Executable Life Cycle State

The Executable life cycle state indicates that the TA is fully operational and ready to handle sessions opened by REE Client Applications or by Trusted Applications of the TEE.

Sessions opened with the TA will result in the creation of a new instance of the TA or the reuse of the single instance (refer to [TEE Core API] for possible execution models for a TA).

In this state, an authorized Security Domain (SD-A) controlling this TA and having the required privileges may perform the following privileged administration operations:

Store Data and Delete Data

Personalization data can be updated in this state using the *Store Data* operation. However, since running instances may access the data, the update should be done in a way that maintains the consistency of the data. This could be achieved, for example, by performing the atomic update of a single object or by using a specific design of the TA. For maintenance operations that may temporarily make personalization data inconsistent, the Locked life cycle state must be used.

List Objects

This operation retrieves the list of the objects currently stored by the *Store Data* operation.

Uninstall TA

This operation automatically and immediately closes all sessions opened with instances of this TA (see Chapter 11) and deletes all data and metadata associated with this application (see section 6.2.2, item #4).

Lock TA

This operation switches the TA to the Locked life cycle state.

4.3.3 Locked Life Cycle State

The Locked life cycle state is used to prevent REE Client Applications and other Trusted Applications from using this TA. This is typically used to ensure that the application is not in use while a maintenance operation, such as the update of code or personalization data, is being performed.

Transition into this state automatically and immediately closes all sessions opened with instances of this TA (see Chapter 11). Any attempt to open a session with a TA in the Locked life cycle state SHALL be rejected with the `TEE_ERROR_ACCESS_DENIED` error code. Thus, application instances will only be able to access consistent data, after the maintenance operation ends.

In this state, an authorized Security Domain (SD-A) controlling this TA and having the required privileges may perform the following privileged administration operations:

Store Data and Delete Data

TA personalization data can be updated in this state using the *Store Data* or *Delete Data* operations.

List Objects

This operation retrieves the list of the TA's objects currently stored by the *Store Data* operation.

Update TA

TA code can be updated only in this state. This ensures that multiple versions of a TA will never be running at the same time. This operation will modify the life cycle state of the updated TA only if the *Executable* state value is passed as the *Initial State* parameter value of this operation (see section 6.2.3).

Uninstall TA

This operation deletes all data and metadata associated with the TA (see section 6.2.2, item #4).

Unlock TA

This operation switches the TA to the Executable life cycle state.

4.3.4 Inactive Life Cycle State

A TA automatically switches to the Inactive life cycle state when its parent Security Domain is blocked (see the Blocked state of a Security Domain in section 4.4.4). This state change automatically and immediately closes all sessions opened with instances of this TA (see Chapter 11).

Any attempt to open a session with a TA in Inactive state will be rejected with the `TEE_ERROR_ACCESS_DENIED` error code.

If the parent Security Domain is unblocked, the TA switches back to the state it had before entering the Inactive state (represented in Figure 4-4 by the shallow history pseudostate H).

In this state, an authorized Security Domain (SD-A) controlling this TA and having the required privileges may perform the following privileged administration operation:

Uninstall TA

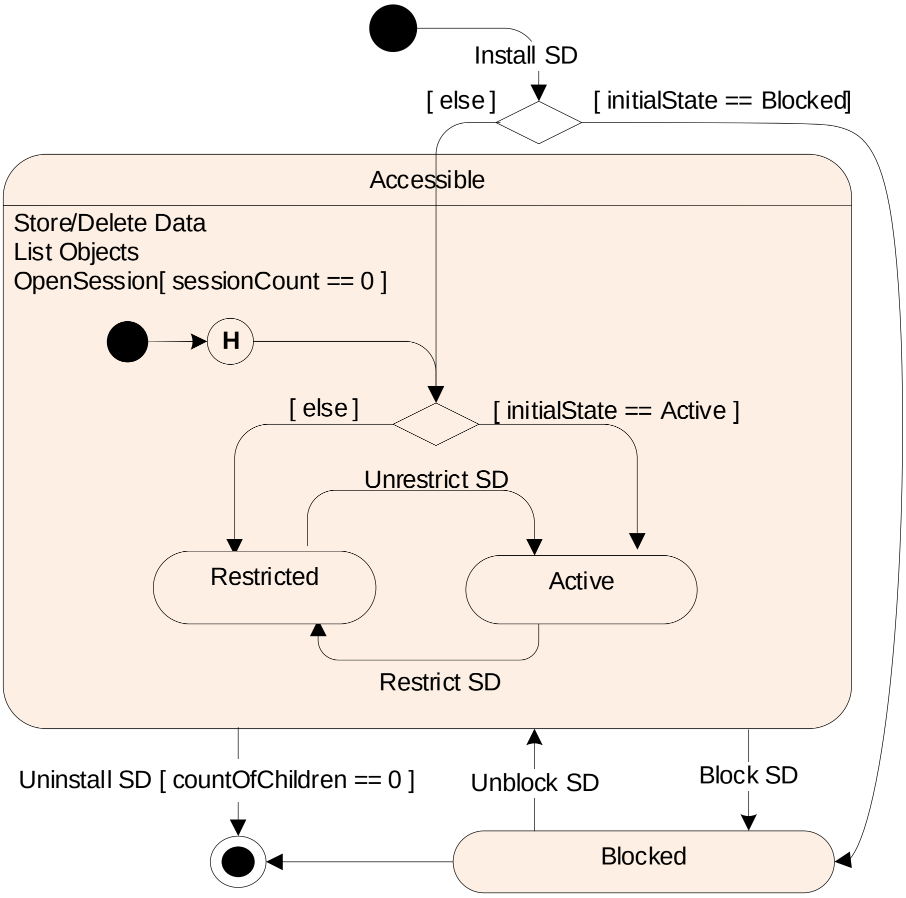
This operation deletes all data and metadata associated with the TA (see section 6.2.2, item #4).

4.4 Security Domain Life Cycle

4.4.1 General State Diagram

The following state diagram (based on [UML]) represents the life cycle of a Security Domain and lists the privileged administration operations that trigger the transition from one state to another.

Figure 4-5: Security Domain Life Cycle



These states are persistent states, meaning that they are not affected by a power-off. The transitions from one state to another are atomic and always triggered by administration operations performed on the Security Domain.

The states of a Security Domain can be grouped into two categories:

Table 4-6: SD Life Cycle State Categories

Category	Description	State(s)
Accessible	The SD is fully operational. It can be used by Client Applications (Active state) or is temporarily suspended to perform some maintenance operations (Restricted state).	Active Restricted
Blocked	The SD has been blocked and can neither be used nor administrated until it is unblocked.	Blocked

When a Security Domain is installed, its initial state can be chosen and can be the Blocked state (e.g. delayed activation) or either of the *Accessible* states: Active (available for immediate use) or Restricted (e.g. additional personalization required). This is depicted in Figure 4-5 above by the guard expressions *[initialState == Blocked]* and *[initialState == Active]* with the corresponding *[else]* alternatives.

The following sub-sections describe the life cycle states in detail.

4.4.2 Active Life Cycle State

The Active life cycle state indicates that the SD is fully operational and able to handle sessions opened by REE Client Applications or by Trusted Applications of other TEEs. It can also be used to verify Authorization Tokens or to perform services for the applications under its control.

In this version of the specification, there SHALL be at most one administration session opened at a time with any SD in the whole TEE. This is depicted in Figure 4-5 above by the guard expression *[sessionCount ==* `0`*]*.

In this state, an authorized Security Domain (SD-A) controlling this SD and having the required privileges may perform the following privileged administration operations:

Store Data and Delete Data

SD keys can be updated in this state using the *Store Data* or *Delete Data* operations but must maintain consistency of the data since applications may access the keys during the update. For maintenance operations that may temporarily make key data inconsistent, the Restricted life cycle state must be used.

List Objects

This operation retrieves the list of the SD’s objects currently stored by the *Store Data* operation.

Restrict SD

This operation switches the SD to the Restricted life cycle state (see section 4.4.3).

Block SD

This operation switches the SD to the Blocked life cycle state (see section 4.4.4).

Uninstall SD

This operation deletes all data and metadata associated with this SD (see section 6.3.2, item #4).

Uninstallation can occur only if the SD has no remaining child SD or TA. This is depicted in Figure 4-5 by the guard expression *[countOfChildren ==* `0`*]*.

4.4.3 Restricted Life Cycle State

The Restricted life cycle state is used to suspend the internal services offered by an SD in order to perform maintenance operations. In particular, when an SD is in this state, any attempt by another SD to use it as the Authorizing SD to verify the signature of an Authorization Token (see sections 5.2 and 5.3) will fail with the `TEE_ERROR_ACCESS_DENIED` error code. However, it is still possible to open a session with the SD to perform maintenance operations.

In this state, an authorized Security Domain (SD-A) controlling this SD and having the required privileges may perform the following privileged administration operations:

Store Data and Delete Data

SD keys and data can be updated in this state with the guarantee that the SD will not be used until it is switched back to the Active state.

List Objects

This operation retrieves the list of the SD's objects currently stored by the *Store Data* operation.

Unrestrict SD

This operation switches the SD to the Active state (see section 4.4.2).

Block SD

This operation switches the SD to the Blocked state (see section 4.4.4).

Uninstall SD

This operation deletes all data and metadata associated with this SD (see section 6.3.2, item #4).

Uninstallation can occur only if the SD has no remaining child SD or TA. This is depicted in Figure 4-5 by the guard expression *[countOfChildren ==* `0`*]*.

4.4.4 Blocked Life Cycle State

The Blocked life cycle state prevents any use of an SD as well as all the TAs directly associated with it. It is not possible to open a session with the SD or with any of its direct TAs, nor is it possible to use the services offered by the SD (e.g. to verify an Authorization Token associated with an operation or to access its keys to establish a secure channel with a Secure Element). In particular, an operation requiring Authorization Token verification from an SD-A that is in the Blocked state SHALL fail with the `TEE_ERROR_ACCESS_DENIED` error code.

The transition to the Blocked state is performed by another Authority whose SD has control over the one being blocked. The state change automatically triggers each TA whose parent is this SD to switch to the Inactive state. All sessions to affected TAs are closed immediately (see Chapter 11).

Blocking an SD has no effect on child SDs.

Any attempt to open a session with an SD in the Blocked state will be rejected with the `TEE_ERROR_ACCESS_DENIED` error code.

In this state, an authorized Security Domain controlling this SD and having the required privileges may perform the following privileged administration operations:

Unblock SD

This operation switches the SD back to the state it had before entering the Blocked state (depicted in Figure 4-5 above by the shallow history pseudostate H).

In the special case where the SD is installed in the Blocked state (depicted in Figure 4-5 above by the guard expression *[initialState == Blocked]*), this operation switches the SD to the Active state.

This unblock operation can be performed only by a parent SD controlling a blocked child.

Uninstall SD

This operation deletes all data and metadata associated with this SD (see section 6.3.2, item #4).

Uninstallation can occur only if the SD has no remaining child SD or TA. This is depicted in Figure 4-5 by the guard expression *[countOfChildren == `0`]*.

4.5 TEE Audit Information

When installing applications on a device that has both an insecure and a secure side, it is necessary to determine information about the capabilities of the TEE into which the secure part will be installed.

This information allows an unprivileged application installed in the REE to determine whether its secure component can run and to determine which Authority it should contact, either to request authorization to install its trusted component or to request that the Authority perform the secure installation. This is especially important for applications that can continue to operate in a degraded manner when their trusted component is not installed.

This information is provided in the form of TLV data which is associated with, and maintained along with, the TEE/Trusted OS.

This specification provides a full description of unprivileged operations allowing Client Applications to access TEE, Security Domain, and Trusted Application characteristics.

Information returned by the audit functions is in plain text. It SHOULD therefore be accessed over a channel that prevents third parties from modifying the response.

The audit commands (see section 8.8) corresponding to these unprivileged operations (see section 6.6) are the following:

- Get TEE Definition (section 6.6.1)
  - Provides TEE information in TLV format (as defined in section 9.1)
- Get SD Definition (section 6.6.2)
  - Provides Security Domain information in TLV format (as defined in section 9.2)
- Get List of Trusted Applications (section 6.6.3)
  - Provides the UUIDs of Trusted Applications of a Security Domain
- Get Trusted Application Definition (section 6.6.4)
  - Provides Trusted Application information in TLV format (as defined in section 9.3)

Any audit commands can be submitted to SD-P, where SD-P can be:

- Any Security Domain that is accessible; i.e.:
  - The SD is not blocked (Blocked life cycle state) and, if the TEE is locked, the SD has the `gpd.privilege.teeManagement` privilege.
- The audit SD defined by this specification (hereafter called the TMF audit SD):
  - Its reserved UUID is defined in section A.3.
  - It is capable of performing only unprivileged audit operations. Any attempt to use it to perform any other administration command will be rejected with the `TEE_ERROR_DENIED_ACCESS` error code.
  - It is implementation-defined whether the TMF audit SD is made available to Trusted Applications executing within the TEE.

5 Authentication and Authorization

Before any privileged operation defined by this specification is performed on the TEE, authorization to perform the operation SHALL be verified.

The following options are defined to authorize administration operations:

- **Implicit authorization** – An operation is implicitly authorized as soon as a sufficiently secure communication channel session (see section 5.1) is opened to pass the corresponding administration command to a destination Security Domain (SD-P) of the TEE. The existence of such a secure opened session is considered valid proof of verification of such an authorization.
- **Explicit authorization** – Performing administration operations on the TEE may require the collaboration of several remote entities having different privileges (i.e. owning different Security Domains) to administer the TEE. In such a configuration, an Authorization Token (as discussed in section 5.3) can be passed with administration command avoiding, for example, opening multiple secure channels to implicitly authorize the operations. This specification defines how an Authorization Token is formatted, how it is passed with an administrative command to a destination Security Domain (SD-P) of the TEE, and how it is verified during this operation. It is out of scope of this specification to define how it is provided and distributed by the remote entities.
- **Combination authorization** – Implicit and explicit authorization may be combined by sending an Authorization Token over a secure channel.

5.1 Authentication and Secure Communication

To begin an administration session, the remote entity on one side and the Security Domain application on the other side SHOULD perform mutual authentication and establish a secure communication channel.

In addition, when performing personalization operations including sensitive data, the integrity and confidentiality of the exchanges are required.

Mutual authentication, integrity, confidentiality, and anti-replay measures are fully supported by using a Security Layer. An implementation can support several protocols to fulfill different security constraints, different business needs, or local rules.

This specification does not define any specific Security Layer but rather provides a generic structure to support the use of a Security Layer (see section 8.2). Each Security Domain that uses a Security Layer needs to agree with its remote entity on the Security Layer to be used.

GlobalPlatform has specified Security Layers – TMF: Asymmetric Cryptography Security Layer ([TMF Asymmetric]), TMF: Symmetric Cryptography Security Layer ([TMF Symmetric]) – and may define additional ones for use in future releases of this specification.

5.2 Authorization of Administration Operations

Before an administration operation is performed on a Trusted Application or on another Security Domain, the authorization for the operation SHALL be verified. The SD that receives the command corresponding to the administration operation is not necessarily the SD that authorizes the operation. To support this distinction, two concepts are introduced:

- The *Authorizing Security Domain* (denoted SD-A) owns the credential required to verify the authorization.
- The *Performing Security Domain* (denoted SD-P) is the recipient SD of the operation command.

The authorization can be achieved in two distinct ways:

- Explicit authorization using Authorization Tokens
- Implicit authorization using a secure channel

5.2.1 Explicit Authorization Using Authorization Tokens

Explicit authorization can be used when there is no means or desire to establish a direct communication channel between the Authority that signs the authorization and its corresponding on-device Security Domain SD-A in the TEE. The off-device Authority computes and signs the Authorization Token for the operation, and this token is then delivered to the TEE through some other channel. Use cases for explicit authentication include, for example, broadcast channels and other one-way channels. It is also convenient when SD-A is an ancestor of SD-P. In this case the off-device Authority for SD-P can, through other channels, obtain the authorization from the off-device Authority for SD-A, and by its own means distribute the Authorization Token to SD-P.

If the TEE does not support explicit authorization, then any operation including an Authorization Token SHALL be rejected with the `TEE_ERROR_ACCESS_DENIED` error code.

If the TEE supports explicit authorization, then when an Authorization Token is received, SD-A is retrieved as described in section 5.3.3.

5.2.2 Implicit Authorization Using a Secure Channel

In implicit authorization, the off-device Authority is indirectly authorized by a Secure Channel. The trust model here is that the secure channel already includes authentication of the communicating parties, so administrative commands sent over the secure channel are implicitly authorized.

Implicit authorization implies that SD-A is equal to SD-P.

5.2.3 Secure Channel with Authorization Tokens

An Authorization Token may be sent over a secure channel. This is considered an explicit authorization and, if the TEE supports explicit authorization, the SD-P SHALL perform the steps described in section 5.3.3.

If the TEE does not support explicit authorization, then any operation including an Authorization Token SHALL be rejected with the `TEE_ERROR_ACCESS_DENIED` error code.

5.3 Authorization Tokens

Explicit authorization relies on the use of Authorization Tokens. An Authorization Token is computed and signed by an Authority, and represents its authorization to perform an operation under given conditions. An Authorization Token contains the following information:

- The UUID of the Security Domain identifying the emitter of this authorization
- A set of *authorization constraints* which defines a set of conditions that need to be verified (see sections 5.3.2 and 5.3.3) before performing the administration operation authorized by the token
- A *signature* computed with a key owned by the Authority and covering the command to be authorized together with the authorization constraints

The Authorization Token must be verified by the Security Domain of the Authority that has emitted this token: the Authorizing Security Domain (SD-A). The necessary key material to verify the token signature would have been stored either at SD-A installation time or later using the *Store Data* operation.

The Security Domain performing an operation (SD-P) must search SD-A as follows:

- If SD-P has the UUID specified in the Authorization Token and has the privilege to perform this operation, then it will be SD-A (i.e. SD-A = SD-P).
- Else, if an SD that is an ancestor of SD-P has the UUID specified in the Authorization Token and has the privilege to authorize the operation, then it will be SD-A.
- Otherwise (i.e. if no Authorizing Security Domain can be found in the TEE) the operation must be rejected with the `TEE_ERROR_ACCESS_DENIED` error code.

When found, SD-A must first verify the authorization signature, then check that all the constraints are satisfied (see detailed procedure in section 5.3.3). If the verifications are successful, the Security Domain that initially received the operation request (SD-P) can proceed with the requested operation. If the token verification procedure fails, the operation is rejected with the `TEE_ERROR_ACCESS_DENIED` error code.

5.3.1 Authorization Token Structure

Each Authorization Token must contain the following components:

- **Authorization format identifier**

  The format identifier distinguishes different versions of the format of an authorization. This allows changing or extending the structure in future releases of the specification.

- **UUID of the Authorizing Security Domain (SD-A)**

  The UUID unambiguously identifies the Security Domain able to verify the token.

- **Key identifier**

  The key identifier unambiguously identifies the key to use to verify the authorization.

- **Algorithm identifier**

  The algorithm identifier is used to select the algorithm to verify the authorization. Section A.10 defines the mandatory algorithms that an ASN.1 TMF compliant implementation SHALL support to verify a token signature (as well as other possible optional algorithms).

- **Constraints**

  The authorization embeds a set of constraints that must be satisfied to perform the operation. These constraints are used by the emitter of an authorization to restrict its scope, as defined in section 5.3.2.

Before being associated with an administrative command, the Authorization Token SHALL be signed with a signature computed on the entire token's fields as specified in Chapter 10, using the specified key and algorithm.

5.3.2 Authorization Constraints

Three categories of constraints can be used to reduce the scope of an Authorization Token. The first defines constraints on the operation and parameters it authorizes. The second defines which device can execute the operation. The last defines in which specific context the operation is allowed.

Constraint on the Operation

This optional constraint binds the Authorization Token to a single operation.

- **Operation parameters constraints** – This constraint specifies the digest value of a chosen set (which could be empty) of the command identifier and parameters as described by the `ConstraintParamsDigest` type defined in section 10.1.2.

Constraint on the Targeted Device

This optional constraint restricts the operation to a subset of devices. If this constraint is not present, the Authorization Token is valid for any device.

- Operation limited to a **specific model**: The Authorization Token contains a UUID representing the model that must match the `gpd.tee.modelID` property.
- Operation limited to a **unique device**: The Authorization Token contains a UUID representing a unique device that must match the `gpd.tee.deviceID` property.

Constraint on the Execution Context

This optional constraint restricts an administration operation to being executed only when the device is in a specific state (called execution context). Since the execution context evolves, an Authorization Token with such a constraint may not be valid for the entire lifetime of the TEE.

- **Version constraint:** Applicable for all TA operations except *Install TA*; the operation will be authorized only if the version of the application already installed (indicated by its `gpd.ta.version.number` property) is in the range described in the Authorization Token. This could be useful, for example, to upgrade an application and ensure that the operation is not used to revert to an older version.

5.3.3 Authorization Token Verification Procedure

The Authorization Token is verified by the TEE implementation as described below. If a format error is detected while verifying the Authorization Token, the operation must be rejected with the `TEE_ERROR_ACCESS_DENIED` error code.

1. Verify first that the Authorization Token payload and its signature are emitted by a known entity; otherwise reject the operation with the `TEE_ERROR_ACCESS_DENIED` error code.
   - Extract the UUID of SD-A.
   - Verify that SD-A is an ancestor of SD-P and SD-A has the required privilege for the operation.
   - Verify that SD-A is not in the Restricted or Blocked life cycle state.
   - Extract the key identifier specified in the Authorization Token and look for the corresponding key in the SD-A secure storage. If the specified key is not found, reject the operation.
   - Extract the algorithm identifier specified in the Authorization Token. If the algorithm identifier is not found or does not match the key type, reject the operation.
   - Verify the Authorization Token signature using the algorithm and the key specified in the token. If the signature is invalid or the algorithm is unknown or not supported, reject the operation.
2. Verify that the constraints listed in the Authorization Token are satisfied as defined below; otherwise reject the operation with the `TEE_ERROR_ACCESS_DENIED` error code. If the list contains duplicated constraints, then reject the operation with the `TEE_ERROR_BAD_FORMAT` error code.
   - Check that the operation parameters are compliant with the constraints on the operation as follows:

     Calculate a digest value over the concatenation of the operation parameter values as described in section 10.1.2, and check that this value is equal to the digest value specified in the `ConstraintParamsDigest` constraint field of the Authorization Token.
   - If the Authorization Token embeds constraints on the targeted device, check them as follows:
     - If the Authorization Token is emitted for a specific model of devices, check that the `gpd.tee.modelID` property is equal to the one defined in the Authorization Token.
     - If the Authorization Token is emitted for a unique device, check that the `gpd.tee.deviceID` property is equal to the one defined in the Authorization Token.
   - If the Authorization Token embeds a constraint on the execution context, check it as follows:
     - If the Authorization Token is valid for a specific version of a TA, check that the TA already exists and has a version in the defined range:

       ```
       ConstraintMinVersion <= gpd.ta.version.number <= ConstraintMaxVersion
       ```

       The `gpd.ta.version.number` property value is compared as a 32-bit unsigned integer (obtained using `TEE_GetPropertyAsU32` function defined by [TEE Core API] section 4.4.3). Comparison as an unsigned integer enables TA vendors and Authorization emitters to agree upon flexible numbering schemes such as [major.minor]; however, no such scheme is presumed or enforced.

       The `gpd.ta.version` property value (see [TEE Core API] section 4.5) is ignored for the purpose of this constraint. The encoding, if any, of the `gpd.ta.version` property value is undefined, as is its relationship to the `gpd.ta.version.number` property, if any.

5.4 Key Management

5.4.1 Root of Trust Instantiation

To enable the authentication and/or authorization of the device in the field, a Root of Trust (as discussed in GlobalPlatform Root of Trust Definitions and Requirements [RoT]) SHALL guarantee the integrity of the Bootstrap Domain and/or root SDs installed in factory.

The key(s) of the Root of Trust MAY be tied to the device identifier. The keys used to create further SDs SHALL be vouched for, directly or indirectly, by the keys of the Root of Trust.

The method of instantiation of this Root of Trust is out of scope of this document and is TEE or device vendor specific.

5.4.2 Security Domain Keys

The Security Domain keys are regular persistent key objects as defined in [TEE Core API].

Each Authority is in charge of managing the keys of its Security Domain and can implement its own policy. It can, among other things:

- Define a scheme to uniquely identify its keys within its Security Domain. The identifiers of the keys are managed by the key management system associated with this SD.
- Decide the number of keys to use. One Authority may decide to have a unique key to verify the authorization while another may want to have different keys for different groups of operations.
- Decide which key – and by extension, which key length – must be used for an operation.
- Decide when a key must be created or updated.

Adding a key consists of creating a new object in the Security Domain personalization storage (i.e. creating an object with the structure defined above and with a new identifier). This is done using the *Store Data* operation (see section 6.4.1).

Updating a key consists of updating its key material related attributes. This is done using a *Store Data* operation with the same Object Identifier, Type, and Size, but with different attribute values.

The *Install Security Domain* operation permits the initialization of cryptographic material, and the exchange of cryptographic material between the remote Authority and the Security Domain to be created. The `CryptographicData` parameter of the Install SD command (see section 8.5.1 for more details) defines the optional and mandatory request/response values involved in this operation (key data ID, remote Authority input keys for authentication or key encryption, cryptographic material output value, etc.).

Authenticity, integrity, and confidentiality of this exchange are guaranteed by using the cryptographic material of the Security Domain performing this operation (for example, its own key encryption key or the Secure Channel established during this operation).

To bootstrap a Security Domain, the provisioning of its initial keys is needed. See section B.3 for an example describing a method to perform such provisioning.

5.4.3 Using Keys in Administration Operations

Keys are typically used by administration operations to perform the following actions:

- Establish a secure messaging channel between the remote entity and its associated SD (section 5.1).
- Verify the authenticity of an Authorization Token (section 5.3.3).
- Encrypt and decrypt the data-flow.

The Authorization Token structure contains the Key Identifier and Algorithm Identifier to be used to verify it (see section 5.3.1).

5.5 Data Storage

A Security Domain holds secure objects (keys and data) used to perform the administration operations it is in charge of. The required storage for such objects must be private and consequently cannot be accessed by unauthorized Security Domains or Trusted Applications or by any other entities.

A Trusted Application may hold secure objects (keys and data) that are supplied by an owning remote entity before the first invocation of the TA by any Client Applications or, during the TA's life cycle, for renewal purposes. For example, these personalization data are required to parameterize DRM or One-Time-Password Trusted Applications with specific diversified key material. The required personalization data storage SHALL only be opened read only by the Trusted Application. The authorized SD SHALL be capable to asynchronously populate the personalization data storage. For all these reasons, this specification defines a persistent `TEE_STORAGE_PERSO` storage area for Trusted Applications into which these objects can be stored using the Store Data command (described in section 6.4.1).

This storage area is defined by the following identifier:

Table 5-1: Personalization Storage Identifier

`TEE_STORAGE_PERSO`	`0x00000002`

The `TEE_STORAGE_PERSO` storage area and the private SD storage SHALL guarantee the persistency, confidentiality, integrity, and anti-replay (if supported by TEE implementation) of objects that are stored in it.

Moreover, the access rights and sharing permissions (as defined in [TEE Core API] Table 5-3) of a secure object stored in the persistent `TEE_STORAGE_PERSO` storage area SHALL satisfy the following conditions:

For a TA to open any objects in `TEE_STORAGE_PERSO`:

- The access control flags value SHALL be set at least with the `TEE_DATA_FLAG_ACCESS_READ` value (in order to be read by the TA) and the `TEE_DATA_FLAG_SHARE_WRITE` value (to allow an authorized SD to store data without conflict).
- The sharing permission `TEE_DATA_FLAG_SHARE_READ` flag value MAY be added to provide a shared access.

Any attempt by a Trusted Application to open objects located in the `TEE_STORAGE_PERSO` storage area with any other flags SHALL cause a Panic.

Any attempt by a Trusted Application to create or restrict usage of objects located in the `TEE_STORAGE_PERSO` storage area (using `TEE_CreatePersistentObject`, `TEE_RestrictObjectUsage`, or `TEE_RestrictObjectUsage1`) SHALL cause a Panic. (See [TEE Core API] Chapter 5.)

5.6 Secure UUID Generation, Proofing, and Verification

Security Domains and Trusted Applications are identified by their Universal Unique ID (UUID). However, as any authorized Security Domain can install Security Domains or Trusted Applications, there can be no central registry of UUIDs. Consequently, a malicious person could create an SD or TA with the same UUID as an existing legitimate TA and hence impersonate them.

The UUID specification specifies five mechanisms for creating UUIDs. The version numbers are given by the bits 4 through 7 of the *time_hi_and_version* field of the UUID's time stamp as specified in [UUID]. If the UUID is version 5 (SHA-1 digest), the TEE will check that the UUID is the SHA-1 hash of a public key and will require proof of possession of the corresponding private key in the form of a signature on all Install SD, Install TA, or Update TA commands. No such check is performed for UUIDs from Version 1 through Version 4.

In order for a malicious person to impersonate the UUID, he would need to generate an appropriate key pair whose public key is a pre-image of the SHA-1 digest. Assuming a reasonable signature mechanism is used, this should be infeasible.

5.6.1 Generation of UUID Version 5

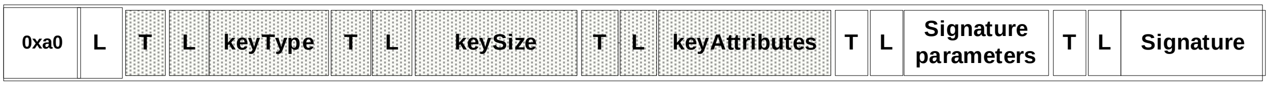
The generation of Trusted Application and Security Domain UUID v5 SHALL be performed according to the following steps:

1. Generate a key pair.

   This version of the specification uses the `keyType`, `keySize`, and `keyAttributes` fields of the UUIDV5Params type (as defined in section 8.3.3.7) to encode the key structure format of the generated public key.

   The TLV format to encode the generated public key is depicted in the grey dotted boxes of the following figure:

Figure 5-1: UUIDV5Params Type Encoding



   The TEE SHALL support the following `keyType` values:

   o `TEE_TYPE_RSA_PUBLIC_KEY`

   o `TEE_TYPE_DSA_PUBLIC_KEY`

   o `TEE_TYPE_ECDSA_PUBLIC_KEY` (if ECC is supported by the TEE implementation)

   The public key SHALL at least include the mandatory attributes listed in Table 5-2 (depending on the `keyType` value).

Table 5-2: List of Mandatory Attributes of the Generated Public Key

Key Types	Mandatory Key Attributes(1)
`TEE_TYPE_RSA_PUBLIC_KEY`	`TEE_ATTR_RSA_MODULUS` `TEE_ATTR_RSA_PUBLIC_EXPONENT`

`TEE_TYPE_DSA_PUBLIC_KEY`	`TEE_ATTR_DSA_PRIME`
	`TEE_ATTR_DSA_SUBPRIME`
	`TEE_ATTR_DSA_BASE`
	`TEE_ATTR_DSA_PUBLIC_VALUE`

`TEE_TYPE_ECDSA_PUBLIC_KEY`	`TEE_ATTR_ECC_PUBLIC_VALUE_X`
	`TEE_ATTR_ECC_PUBLIC_VALUE_Y`
	`TEE_ATTR_ECC_CURVE`(2)

(1) For each key type, the mandatory key attributes encoded in the UUIDV5Params structure value SHALL occur in the same order as the attributes listed in the right column.

When checking the UUID proof of possession (see section 5.6.3), any misplaced, irrelevant, or duplicate attribute SHALL be considered as a bad input and rejected by the parser of the structure with the `TEE_ERROR_BAD_FORMAT` error code.

(2) The ECC curve attribute value is encoded using an `Attribute` type value where the attribute identifier equals `TEE_ATTR_ECC_CURVE` (see [TEE Core API]) and the `value` field encoded with `a` equals the chosen ECC curve value and `b` equals zero (see `Attribute` type encoding in section 8.3.3.1).

2. Calculate a 20-byte hash value using SHA-1 digest algorithm.
   - Name space ID values are 16-byte UUIDs following the endianness convention defined in Chapter 3.
   - Select the appropriate name space ID according to the kind of TEE entity for which the UUID v5 must be generated:

Table 5-3: Name Space ID Value per TEE Entity

TEE Entity	Name Space ID Value
Security Domain	`0xdc03921eb10052dcb4d75fb862734e21`
Trusted Application	`0xd89a41fa1dfd5e1e8593037d0f4c76e4`

   - Concatenate the sequence of bytes of the name space ID with the sequence of bytes of the public key value (i.e. the three grey dotted TLV structures depicted in Figure 5-1).
   - Calculate the hash value of the resulting concatenation using SHA-1 digest algorithm.

3. Transform the resulting 20-byte hash value into a 16-byte version 5 UUID as follows. UUIDs are defined here in big-endian byte order. Refer to [UUID] for field definitions and encodings.
   - Set octets `0` through 3 of the *time_low* field to octets `0` through 3 of the hash.
   - Set octets `0` and 1 of the *time_mid* field to octets 4 and 5 of the hash.
   - Set octets `0` and 1 of the *time_hi_and_version* field to octets 6 and 7 of the hash.
   - Set the four most significant bits (bits 12 through 15) of the *time_hi_and_version* field to 0101.
   - Set the *clock_seq_hi_and_reserved* field to octet 8 of the hash.
   - Set the two most significant bits (bits 6 and 7) of the *clock_seq_hi_and_reserved* field to 01.
   - Set the *clock_seq_low* field to octet 9 of the hash.
   - Set octets `0` through 5 of the *node* field to octets 10 through 15 of the hash.

5.6.2 Proof of Possession

A signature (using the generated private key as defined in step 1 of section 5.6.1) is calculated and concatenated to both the signature information (i.e. the signature algorithm identifier) and the generated public key type and attributes. It constitutes the `UUIDVerificationParams` type parameter value of the Install SD, Install TA, or Update TA commands. Normative section A.10 lists the possible signature algorithms (according to the key types listed in Table 5-2).

Install TA and Update TA Commands

Based on the signature algorithm identifier (see its detailed encoding in [TEE Core API]) a signature is calculated over the sequence of bytes resulting from the concatenation of the tag-length-value octets of the *TA UUID* and the *Application File* parameters of the Install TA or Update TA commands.

Install SD Command

The signature is calculated over the sequence of bytes resulting from the concatenation of the tag-length-value octets of the *SD UUID* and the `CryptographicData` parameters of the Install SD command.

5.6.3 Checking the Proof

During the processing of the Install SD, Install TA, or Update TA command, the TEE verifies a version 5 UUID as follows:

- Extract from the `UUIDVerificationParams` parameter of the command, the UUID v5 parameters consisting of the public key type, its length and attributes values as well as the values of the signature and the algorithm used to calculate this signature. They describe the public key and signature of the UUID owner. Reject the operation with the `TEE_ERROR_BAD_FORMAT` error code if the public key attributes are not encoded according to Table 5-2.
- Calculate the “Verified UUID” from the public key type, its length and attributes values as described in steps 2 and 3 of section 5.6.1. Then check the resulting value against the TA or SD UUID passed as parameter of the command.
- If correct, verify the signature according to the type of command (see section 5.6.2) and the signature algorithm value.

6 Administration Operations

6.1 Introduction

A TEE administration session is opened when a Client Application has successfully called the open session function (defined by [TEE Client] and [TEE Core API]) by passing a Security Domain UUID value as the *destination* UUID parameter value of the function.

In this version of the specification, there SHALL be one and only one administration session opened at a time with any Security Domain in the whole TEE. Any attempt to open another TEE administration session SHALL fail and return the `TEE_ERROR_ACCESS_DENIED` error code.

When the Client Application invokes an administration command during this session, the destination Security Domain (SD-P) is said to *perform the corresponding operation*.

Once the TEE administration session is successfully opened with SD-P, any operation is performed in an execution context which is dependent on the life cycle states of both the TEE, the SD-A that authorizes the operation, the target SD or target TA on which the operation is performed and, in the case of the *Install/Update TA* and *Install SD* operations, the associated parent SD.

The following table defines the error codes that SHALL be immediately returned by an operation when wrong life cycle states are detected while performing the operation or setting its execution context.

Table 6-1: Return Error Codes of Operations According to Life Cycle States

Operations	Life Cycle States and Error Codes		Remarks
Opening an administration session with SD-P	SD-P Blocked	TEE Locked (1)	(1) Applicable only when SD-P has no `gpd.privilege.teeManagement` privilege
	`TEE_ERROR_ACCESS_DENIED`		
Verifying the operation authorization using SD-A (Security Layer and/or token)	SD-A Restricted or Blocked		An administration session SHALL have been successfully opened with SD-P
	`TEE_ERROR_ACCESS_DENIED`		
Lock TA	TA Inactive	TEE Locked	For all these operations, an administration session SHALL have been successfully opened with SD-P and the authorization successfully verified.
	`TEE_ERROR_BAD_STATE`	`TEE_ERROR_ACCESS_DENIED`	
Unlock TA	TA Inactive	TEE Locked	
	`TEE_ERROR_BAD_STATE`	`TEE_ERROR_ACCESS_DENIED`	
Store Data, Delete Data, or List Objects of TA	TA Inactive	TEE Locked	
	`TEE_ERROR_BAD_STATE`	`TEE_ERROR_ACCESS_DENIED`	

Operations	Life Cycle States and Error Codes		Remarks
Install TA or Update TA where SD-T is the parent SD	SD-T Blocked or SD-T Restricted [2]	TEE Locked	[2] Applicable only if an SD-T's key is required to decrypt the TA's *Application File* (see section 6.2.1)
	`TEE_ERROR_BAD_STATE`	`TEE_ERROR_ACCESS_DENIED`	
Restrict SD, Unrestrict SD	SD Blocked	TEE Locked	
	`TEE_ERROR_BAD_STATE`	`TEE_ERROR_ACCESS_DENIED`	
Store Data, Delete Data, or List Objects of SD	SD Blocked	TEE Locked	
	`TEE_ERROR_BAD_STATE`	`TEE_ERROR_ACCESS_DENIED`	
Install SD associated with SD-T, the parent SD	SD-T Blocked or SD-T Restricted [3]	TEE Locked	[3] Applicable only if SD-T's credentials are required to handle optional SD's `CryptographicData` during installation (see section 6.3.1)
	`TEE_ERROR_BAD_STATE`	`TEE_ERROR_ACCESS_DENIED`	
Lock TEE	TEE Locked		
	`TEE_ERROR_BAD_STATE`		
Unlock TEE	TEE Secured		
	`TEE_ERROR_BAD_STATE`		

Moreover, if opening an administration session fails due to a wrong life cycle state, then any command passed as a parameter of the open session call SHALL NOT be performed.

Backward Compatibility

In version 1.0, several places in this table contradicted the textual description. In particular the table required commands that change the state of a TA or SD to return an error if the target was already in the requested state – whereas the text stated that the command should immediately return `TEE_SUCCESS`. The table has been corrected to match the text.

6.1.1 Unprivileged Audit Operations

Any unprivileged audit operations can be submitted to SD-P as soon as a TEE administration session is successfully opened with it.

Moreover, whatever the current life cycle state of the TEE, any unprivileged audit operations can always be submitted to and performed by the TMF audit SD as mentioned in sections 4.5 and 6.6.

6.1.2 Authorization of Operations

For privileged operations, if an Authorization Token is present with the operation command then the authorization is verified according to the procedure defined in section 5.3.3; otherwise the privileged operation SHALL be performed only if SD-P has the privilege required by the operation (see Table 4-2) and a Secure Channel session (see section 5.1) is currently open with it (i.e. SD-A is the SD-P itself, and the operation is implicitly authorized).

For unprivileged audit operations, no authorization is required but if an Authorization Token is present with the operation command, then the authorization is verified according to the procedure defined in section 5.3.3.

Note: In such a case, SD-A is not required to have any specific privilege.

If an Authorization Token is present and its verification failed, then reject the operation with the error codes defined by the procedure in section 5.3.3; otherwise for any other reasons of authorization failure, reject the operation with the `TEE_ERROR_ACCESS_DENIED` error code.

6.1.3 Operation Return Codes

When an operation is successfully performed, it SHALL return the `TEE_SUCCESS` return code value.

This specification defines the `TEE_ERROR_LIMIT_EXCEEDED` error code to be returned when an operation would take the TEE beyond its implementation limits.

If an implementation does not have enough resources to perform any operation, it SHALL return the `TEE_ERROR_OUT_OF_MEMORY` error code.

If an attempt to unwrap the parameters of an administration command fails, then an implementation SHALL return the `TEE_ERROR_BAD_PARAMETERS` or `TEE_ERROR_BAD_FORMAT` error code, as applicable.

All other error codes returned by any subsequent operations are defined by [TEE Core API].

6.1.4 Handling Variable Length Return Values

When a TEE administration session has been opened using the standard TEE Client API [TEE Client], any operation that returns output data as a result of an administration command SHALL use the mechanism defined by this standard to handle variable length return values, in particular:

- If the output does not fit in the output buffer, then the `TEE_ERROR_SHORT_BUFFER` error code is returned as the return code of the command and the `TEEC_ERROR_SHORT_BUFFER` code is returned as the status code of the envelope command (see details in section 8.1).
- The size indicator of the output buffer parameter of the envelope command is populated with the size that would be required for a subsequent call to succeed. This may be an overestimate but must always be sufficient for a subsequent call to succeed.

When an implementation of this specification supports a TEE administration session opened by a client Trusted Application using the TEE Internal Core API ([TEE Core API]), then any operation that returns output data as a result of an administration command SHALL use the mechanism defined by this standard to handle variable length return values, in particular:

- If the output does not fit in the output buffer, then the implementation SHALL update the output buffer parameter size indicator with the required number of bytes and then return the `TEE_ERROR_SHORT_BUFFER` error code as the result of the command invocation by the internal TA (see details in section 8.1).

6.1.5 Atomicity of Operations

All operation commands SHALL appear atomic to Actors using the GlobalPlatform TMF protocols. A TEE may adopt a variety of strategies internally, including performing garbage collection and applying other required operations in a delayed manner following a TMF operation command. Some TMF operation commands MAY lock out GlobalPlatform TA or SD functionality until the TEE can complete processing of the requested TMF operation.

6.1.6 Operations Description

The next sections describe the *minimum* set of actions that a TEE implementation compliant with this specification SHALL perform during the processing of administration operations.

A logical order is proposed to describe this set of actions through different steps. Nevertheless, a compliant implementation MAY change this order, MAY mandate that an optional behavior is mandatory, and/or MAY add new implementation-defined actions, provided that these changes respect the characteristics defined and requested when performing an operation (e.g. specific UUID verifications, atomicity, and roll-back, when applicable).

6.2 Trusted Application Privileged Operations

6.2.1 Install Trusted Application

Install TA is a privileged operation that downloads a new Trusted Application in the TEE. The installation consists of copying the required parts of the Application File in the persistent storage controlled by the Target Security Domain and creating the corresponding metadata to make the TA ready for execution. It also sets the initial Trusted Application life cycle state and associates the TA with a parent Security Domain.

The Security Domain in charge of this operation (SD-P) performs the following actions:

1. Unwrap the operation parameters:
    - The *Trusted Application UUID* of the TA to install
    - The *Target Security Domain* with which the installed TA must be associated
    - The *Initial State* of the application
    - The *Application File* including both the TA binary code and the TA properties
    - The *Encryption* parameter, indicating that the *Application File* is encrypted, and including the following attributes:
        - The *Key Identifier* of the encryption key
        - The *Key Algorithm* of the encryption key
        - Optional extra *parameters* associated with this *Key Algorithm* (e.g. an Initial Vector value for symmetric algorithms)
    - The *UUID verification* parameter, including the following attributes:
        - The *Public key type and value attributes* to verify the possible version 5 UUID identifying the installed TA
        - The *Signature algorithm and value* proving the possession of the version 5 UUID
    - (Optional) The *Authorization Token* (explicit authorization only)
2. Identify the Authorizing Security Domain (SD-A) as described in section 6.1.2.
3. Verify the operation parameters and, in case of failure, reject the operation with the indicated error code; if no specific code is mentioned, use `TEE_ERROR_ACCESS_DENIED`:
    - If the *Target Security Domain* (denoted SD-T) does not exist, then reject the operation with the `TEE_ERROR_ITEM_NOT_FOUND` error code.
    - If SD-T is in the Blocked life cycle state, then reject the operation with the `TEE_ERROR_BAD_STATE` error code.
    - If the *Application File* is encrypted, use the SD-T's key (defined by the *Key Identifier*, *Key Algorithm*, and extra *parameters* attributes) to decrypt it. If SD-T is in the Restricted life cycle state or the key object described by the *Key Identifier* is corrupted or not found, reject the operation with the appropriate error code: `TEE_ERROR_BAD_STATE`, `TEE_ERROR_CORRUPT_OBJECT`, `TEE_ERROR_ITEM_NOT_FOUND`, or `TEE_ERROR_STORAGE_NOT_AVAILABLE`.
    - Verify the TA binary code – details are implementation dependent.

- o Verify the TA properties whose values and types are defined in [TEE Core API] or other GlobalPlatform specifications:
  - ▪ Check that each `gpd.ta.*` entry identifies a property defined in [TEE Core API].
  - ▪ Check that each `gpd.ta.*` entry has a value compatible with its appropriate type.
  - ▪ Check that no property is defined in the `gpd.*` namespace other than in `gpd.ta.*`.
- o If the *Trusted Application UUID* parameter is a version 5 UUID, check that the *UUID verification* parameter is present, then verify the *Trusted Application UUID* value according to the procedure defined in section 5.6.3.

4. Verify that the pre-conditions to install the new TA are satisfied; otherwise reject the operation with the `TEE_ERROR_ACCESS_DENIED` error code:
   - o Check that the *Trusted Application UUID* value does not correspond to an existing TA/SD in the TEE.
   - o Check that the TA to be installed is in the range of SD-A's scope of control for the `gpd.privilege.taManagement` privilege by applying the rules defined in sections 4.1.3.2 and 4.1.3.3.
5. Atomically perform the Trusted Application installation:
   - o If the operation would take the TEE beyond its implementation limits, reject the operation with the `TEE_ERROR_LIMIT_EXCEEDED` error code.
   - o Store the necessary components of the *Application File* in the persistent storage.

     If there is not enough memory in the persistent storage, reject the operation with the `TEE_ERROR_STORAGE_NO_SPACE` error code.

     If the persistent storage is currently not accessible, reject the operation with the `TEE_ERROR_STORAGE_NOT_AVAILABLE` error code.
   - o Perform the registration of the TA with the trusted OS by recording the metadata related to the TA binary code and appending the newly installed TA to the list of applications already directly controlled by SD-T.

     If the TA refers to an unknown API, reject the operation with the `TEE_ERROR_ACCESS_DENIED` error code.

     If there is not enough memory to perform the registration, reject the operation with the `TEE_ERROR_OUT_OF_MEMORY` error code.
   - o Finally, commit the operation by setting the TA life cycle state according to the *Initial State* parameter value.

     If the installation fails, reject the operation and perform all the necessary cleanup.

6.2.2 Uninstall Trusted Application

Uninstall TA is a privileged operation that removes a Trusted Application from the list of available applications making it impossible to open new sessions with it. It also performs the necessary memory cleanup and removes the application code, the application data, and all the associated metadata.

The Security Domain in charge of this operation (SD-P) performs the following actions:

1. Unwrap the operation parameters:
    - The *Trusted Application UUID* of the TA to uninstall
    - (Optional) The *Authorization Token* (explicit authorization only)
2. Identify SD-A as described in section 6.1.2.

    If the verifications fail, reject the operation with the `TEE_ERROR_ACCESS_DENIED` error code.
3. Verify that the pre-conditions to uninstall the Trusted Application are satisfied; otherwise reject the operation with the `TEE_ERROR_ACCESS_DENIED` error code (only when no specific code is mentioned):
    - Check that the *Trusted Application UUID* corresponds to an existing TA on the TEE; otherwise reject the operation with the `TEE_ERROR_ITEM_NOT_FOUND` error code.
    - Check that the TA is in the range of the SD-A's scope of control for the `gpd.privilege.taManagement` privilege by applying the rules defined in sections 4.1.3.2 and 4.1.3.3.
4. Atomically uninstall the Trusted Application.
    - If the TA life cycle state is Executable, then internally flag that the TA is going to shut down, then shut down all currently open sessions with this TA as specified in Chapter 11.
    - Remove the TA from the list of available TAs associated with its parent SD to avoid concurrent access while deleting associated data.
    - Delete all data of the TA that was created in its private and personalization storage since its installation.
    - Delete all metadata associated with the TA registration stored during its installation.

6.2.3 Update Trusted Application

Update TA is a privileged operation that downloads a new version of a Trusted Application in the persistent storage of the TEE while keeping the data of the previous version.

The Security Domain in charge of this operation (SD-P) performs the following actions:

1. Unwrap the operation parameters:
   - The *Trusted Application UUID* of the TA to update
   - The *New State* of the application
   - The *Application File* including the binary code and the Trusted Application properties
   - The *Encryption* parameter, indicating that the Application File is encrypted, and including the following attributes:
     - The *Key Identifier* of the encryption key
     - The *Key Algorithm* of the encryption key
     - Optional extra *parameters* associated with this *Key Algorithm* (e.g. an Initial Vector value for symmetric algorithms)
   - The *UUID verification* parameter, including the following attributes:
     - The *Public key type and value attributes* to verify the possible version 5 UUID assigned to the updated TA
     - The *Signature algorithm and value* proving the possession of the version 5 UUID
   - (Optional) The *Authorization Token* (explicit authorization only)
2. Identify SD-A as described in section 6.1.2.
3. Verify the content of the *Application File* and, in case of failure, reject the operation with the error code indicated; if no specific code is mentioned, use `TEE_ERROR_ACCESS_DENIED`:
   - Check that the *Trusted Application UUID* parameter corresponds to an existing TA; otherwise reject the operation with the `TEE_ERROR_ITEM_NOT_FOUND` error code.
   - If the *Application File* is encrypted:
     - If the direct parent SD of the updated TA is in the Blocked life cycle state or the Restricted life cycle state, reject the operation with the `TEE_ERROR_BAD_STATE` error code.
     - Decrypt the *Application File* using the direct parent SD's key defined by the *Key Identifier*, *Key Algorithm*, and optional extra *parameters* attributes. If the key object described by the *Key Identifier* is corrupted, reject the operation with the `TEE_ERROR_CORRUPT_OBJECT` error code; if it is not found, reject the operation with the `TEE_ERROR_ITEM_NOT_FOUND` error code.
   - Verify the binary code – details are implementation dependent.
   - Verify the TA properties whose values and types are defined in [TEE Core API]:
     - Check that each `gpd.ta.*` entry is a known property name.
     - Check that each `gpd.ta.*` entry has a value compatible with its appropriate type.
     - Check that no properties are defined in the `gpd.*` namespace other than in `gpd.ta.*`.
   - If the *Trusted Application UUID* parameter is a version 5 UUID, check that the *UUID verification* parameter is present, then verify the *Trusted Application UUID* value according to the procedure described in section 5.6.3.

4. Verify that the pre-conditions to update the Trusted Application are satisfied; otherwise reject the operation with the `TEE_ERROR_ACCESS_DENIED` error code (only when no specific code is mentioned):
   - Check that this TA is in the range of the SD-A's scope of control for the `gpd.privilege.taManagement` privilege by applying the rules defined in sections 4.1.3.2 and 4.1.3.3.
   - Check that this TA is in the Locked life cycle state, otherwise reject the operation with the `TEE_ERROR_BAD_STATE` error code.
5. Atomically perform the Trusted Application update:
   - If the operation would take the TEE beyond its implementation limits, reject the operation with `TEE_ERROR_LIMIT_EXCEEDED`.
   - Store the necessary components of the Application File in the persistent storage.

     If there is not enough memory in the persistent storage, reject the operation with the `TEE_ERROR_STORAGE_NO_SPACE` error code.

     If the persistent storage is currently not accessible, reject the operation with the `TEE_ERROR_STORAGE_NOT_AVAILABLE` error code.
   - Perform the registration of the TA with the trusted OS by modifying the metadata of the updated TA with a reference to the new TA binary code.

     If the TA refers to an unknown API, reject the operation with the `TEE_ERROR_ACCESS_DENIED` error code.

     If there is not enough memory to perform the registration, reject the operation with the `TEE_ERROR_OUT_OF_MEMORY` error code.
   - Finally, commit the operation by updating the TA life cycle state according to the *New state* parameter value.

6.2.4 Lock TA

Lock TA is a privileged operation that updates the life cycle state of a Trusted Application (see section 4.3).

The Security Domain in charge of this operation (SD-P) performs the following actions:

1. Unwrap the operation parameters and extract the following parameters:
   - The *Trusted Application UUID* to lock
   - (Optional) The *Authorization Token* (explicit authorization only)
2. Identify SD-A as described in section 6.1.2.
3. Verify that the pre-conditions to lock the Trusted Application are satisfied; otherwise reject the operation with the `TEE_ERROR_ACCESS_DENIED` error code (only when no specific code is mentioned):
   - Check that the *Trusted Application UUID* corresponds to an existing TA; otherwise reject the operation with the `TEE_ERROR_ITEM_NOT_FOUND` error code.
   - Check that the TA is in the range of the SD-A's scope of control for `gpd.privilege.taManagement` or `gpd.privilege.taPersonalization` privilege – by applying the rules defined in sections 4.1.3.2 and 4.1.3.3.
   - If the TA is already in the Locked life cycle state, then return immediately with the `TEE_SUCCESS` return code.
   - Check that the TA is in the Executable life cycle state; otherwise reject the operation with the `TEE_ERROR_BAD_STATE` error code.
4. Atomically modify the Trusted Application life cycle state:
   - Flag internally that the TA is going to shut down, then shut down all currently open sessions with this TA as specified in Chapter 11.
   - Move the TA to the Locked life cycle state.

6.2.5 Unlock TA

Unlock TA is a privileged operation that updates the life cycle state of a Trusted Application (see section 4.3).

The Security Domain in charge of this operation (SD-P) performs the following actions:

1. Unwrap the operation parameters and extract the following parameters:
   - The *Trusted Application UUID* to unlock
   - (Optional) The *Authorization Token* (explicit authorization only)
2. Identify SD-A as described in section 6.1.2.
3. Verify that the pre-conditions to unlock the Trusted Application are satisfied; otherwise reject the operation with the `TEE_ERROR_ACCESS_DENIED` error code (only when no specific code is mentioned):
   - Check that the *Trusted Application UUID* corresponds to an existing TA; otherwise reject the operation with the `TEE_ERROR_ITEM_NOT_FOUND` error code.
   - Check that the TA is in the range of the SD-A's scope of control for the `gpd.privilege.taManagement` or the `gpd.privilege.taPersonalization` privilege – by applying the rules defined in sections 4.1.3.2 and 4.1.3.3.
   - If the TA is already in the Executable life cycle state, then return immediately with the `TEE_SUCCESS` return code.
   - Check that the TA is in the Locked life cycle state; otherwise reject the operation with the `TEE_ERROR_BAD_STATE` error code.

   If the verifications fail, reject the operation with the `TEE_ERROR_ACCESS_DENIED` error code.
4. Atomically modify the Trusted Application life cycle state.
   - Move the TA to the Executable life cycle state.

6.2.6 Update TA and Data

New in version 1.1

Update TA and Data is a privileged operation that atomically downloads a new version of a Trusted Application in the persistent storage of the TEE and updates its associated data.

The Security Domain in charge of this operation (SD-P) performs the following actions:

1. Unwrap the operation parameters:
   - The *Trusted Application UUID* of the TA to update
   - The *New State* of the application
   - The *Application File* including the binary code and the Trusted Application properties
   - The *Encryption* parameter, indicating that the Application File is encrypted, and including the following attributes:
     - The *Key Identifier* of the encryption key
     - The *Key Algorithm* of the encryption key
     - Optional extra *parameters* associated with this *Key Algorithm* (e.g. an Initial Vector value for symmetric algorithms)
   - The *UUID verification* parameter, including the following attributes:
     - The *Public key type and value attributes* to verify the possible version 5 UUID assigned to the updated TA
     - The *Signature algorithm and value* proving the possession of the version 5 UUID
   - (Optional) The *Authorization Token* (explicit authorization only)
2. Identify SD-A as described in section 6.1.2.
3. Verify the content of the *Application File* and, in case of failure, reject the operation with the error code indicated; if no specific code is mentioned, use `TEE_ERROR_ACCESS_DENIED`:
   - Check that the *Trusted Application UUID* parameter corresponds to an existing TA; otherwise reject the operation with the `TEE_ERROR_ITEM_NOT_FOUND` error code.
   - If the *Application File* is encrypted:
     - If the direct parent SD of the updated TA is in the Blocked life cycle state or the Restricted life cycle state, reject the operation with the `TEE_ERROR_BAD_STATE` error code.
     - Decrypt the *Application File* using the direct parent SD's key defined by the *Key Identifier*, *Key Algorithm*, and optional extra *parameters* attributes. If the key object described by the *Key Identifier* is corrupted, reject the operation with the `TEE_ERROR_CORRUPT_OBJECT` error code; if it is not found, reject the operation with the `TEE_ERROR_ITEM_NOT_FOUND` error code.
   - Verify the binary code – details are implementation dependent.
   - Verify the TA properties whose values and types are defined in [TEE Core API]:
     - Check that each `gpd.ta.*` entry is a known property name.
     - Check that each `gpd.ta.*` entry has a value compatible with its appropriate type.
     - Check that no properties are defined in the `gpd.*` namespace other than in `gpd.ta.*`.

- If the *Trusted Application UUID* parameter is a version 5 UUID, check that the *UUID verification* parameter is present, then verify the *Trusted Application UUID* value according to the procedure described in section 5.6.3.
- The *Object* to be stored in the personalization storage of the TA or SD. This object consists of:
  - An *object identifier*
  - An *object type*
  - An *access attribute* made of a combination of access control and sharing permissions flags
  - A *list of attributes* defining the attributes values of the object when referring to a key or key-pair object as defined in [TEE Core API]
  - A possible data stream associated with the object as defined in [TEE Core API]
  - The possible metadata associated with the object when referring to a key or key-pair object as defined in [TEE Core API]

4. Verify that the pre-conditions to update the Trusted Application are satisfied; otherwise reject the operation with the `TEE_ERROR_ACCESS_DENIED` error code (only when no specific code is mentioned):
   - Check that this TA is in the range of the SD-A's scope of control for the `gpd.privilege.taManagement` privilege by applying the rules defined in sections 4.1.3.2 and 4.1.3.3.
   - Check that this TA is in the Locked life cycle state, otherwise reject the operation with the `TEE_ERROR_BAD_STATE` error code.
5. Atomically perform the Trusted Application and Data update:
   - If the operation would take the TEE beyond its implementation limits, reject the operation with `TEE_ERROR_LIMIT_EXCEEDED`.
   - Store the necessary components of the Application File in the persistent storage.

     If there is not enough memory in the persistent storage, reject the operation with the `TEE_ERROR_STORAGE_NO_SPACE` error code.

     If the persistent storage is currently not accessible, reject the operation with the `TEE_ERROR_STORAGE_NOT_AVAILABLE` error code.
   - Perform the registration of the TA with the trusted OS by modifying the metadata of the updated TA with a reference to the new TA binary code.

     If the TA refers to an unknown API, reject the operation with the `TEE_ERROR_ACCESS_DENIED` error code.

     If there is not enough memory to perform the registration, reject the operation with the `TEE_ERROR_OUT_OF_MEMORY` error code.
   - If the *Decryption* parameter is not null, then extract the necessary information to decipher the ciphered text encoding the *Object* parameter.
     - The key identifier of the *Decryption* parameter refers to a key object owned by the direct parent SD in case of a TA, or owned by SD-P in case of an SD; if decryption fails then reject the operation with the `TEE_ERROR_ACCESS_DENIED` error code.
   - Verify that the *Object* parameter value is consistent; otherwise reject the operation with either the `TEE_ERROR_BAD_FORMAT` error code or, if the TEE implementation does not support the type or length of an attribute, with the `TEE_ERROR_NOT_SUPPORTED` error code:

  - Check that the *access attribute* of the *Object* parameter is valid according to the values defined by [TEE Core API] Table 5-2 and the constraints defined by section 5.5 of this specification.
  - Check that the *object type* is a value as defined in [TEE Core API] Table 6-13.
  - If the list of attributes is not empty, then:
    - For each attribute:
      - Check that its *identifier* and the format of its value conform to [TEE Core API] Tables 6-15, 6-16, 6-17, and 6-18.
      - Check that no mandatory attribute is missing for the specified *object type*.
- Determine the kind of operation to be performed on the `TEE_STORAGE_PERSO` storage of the TA:
  - If an object with the same identifier as specified by the *Object* parameter already exists in this storage:
    - If this object has the same type, then this operation will attempt to replace it.
    - Otherwise reject the operation with the `TEE_ERROR_ACCESS_DENIED` error code.
  - Otherwise this operation will attempt to create a new permanent object.

6. Atomically, according to the operation to be performed:
   - If the operation would take the TEE beyond its implementation limits, reject the operation with the `TEE_ERROR_LIMIT_EXCEEDED` error code.

     Depending on the kind of operation to be performed (see last bullet of step 3), create or replace the permanent object in the `TEE_STORAGE_PERSO` storage space of the TA or in the private storage of the SD. If the storage is unreachable, reject the operation with the `TEE_ERROR_STORAGE_NOT_AVAILABLE` error code; if it is corrupted, reject the operation with the `TEE_ERROR_CORRUPT_OBJECT` error code
7. Finally, commit the operation by updating the TA life cycle state according to the *New state* parameter value.

6.3 Security Domain Privileged Operations

6.3.1 Install Security Domain

Install SD is a privileged operation that creates a new Security Domain on the TEE.

The Security Domain in charge of this operation (SD-P) performs the following actions:

1. Unwrap the operation parameters and extract the following parameters:
    - The *Security Domain UUID* that identifies the SD to install
    - The *Target Security Domain UUID* that identifies the SD (denoted SD-T) which this newly created SD will be directly associated with
    - The *Initial state* of the newly created SD
    - The *Privileges* assigned to this SD
    - The *Authority* that identifies the remote entity managing this SD
        - A name and an optional URL
    - The *Cryptographic data* that describes the possible key material provided by the remote entity server that has to be installed in the SD
    - The *UUID verification* parameter, including the following attributes:
        - The *Public key type and value attributes* to verify the possible UUID v5 assigned to the installed SD
        - The *Signature algorithm and value* proving the possession of the UUID v5
    - (Optional) The *Authorization Token* (explicit authorization only)
2. Identify SD-A as described in section 6.1.2.
3. Verify that the pre-conditions to install the new SD are satisfied; otherwise reject the operation with the `TEE_ERROR_ACCESS_DENIED` error code (only when no specific code is mentioned):
    - Check that a UUID with the same value as the *Security Domain UUID* parameter value does not already exist in the TEE.
    - If the *Security Domain UUID* is a version 5 UUID, check that the *UUID verification* parameter is present, then verify the *Security Domain UUID* value according to the procedure described in section 5.6.3.
    - Check that the *target Security Domain UUID* corresponds to an existing SD; otherwise reject the operation with the `TEE_ERROR_ITEM_NOT_FOUND` error code.
    - Check that the *Privileges* parameter does not contain duplicate privilege values (i.e. with the same `privilegeID` value as defined in section 8.3.3.10); otherwise reject the operation with the `TEE_ERROR_BAD_FORMAT` error code.
    - If the property field `isRootSD` of the *Privileges* parameter is set to TRUE (see section 8.3.3.10) then check that the newly created SD is in the range of the SD-A's scope of control for the `gpd.privilege.rsdManagement` privilege; otherwise perform the check for the `gpd.privilege.sdManagement` privilege – in both cases, by applying the rules defined in sections 4.1.3.2 and 4.1.3.3.

4. Atomically perform the Security Domain installation:
   - If the operation would take the TEE beyond its implementation limits, reject the operation with `TEE_ERROR_LIMIT_EXCEEDED`.
   - If the `CryptographicData` parameter has a *non-null* value, then perform the appropriate implementation-defined actions according to the `cryptoProcID` value of the `CryptographicData` parameter as described in section 8.3.3.8. If the implementation does not support such a parameter value, then reject the operation with the `TEE_ERROR_NOT_SUPPORTED` error code.
     - If the private storage of the newly created SD is used when performing these actions:
       - If the private storage is not accessible, reject the operation with the `TEE_ERROR_STORAGE_NOT_AVAILABLE` error code.
       - If it does not have enough space, reject the operation with the `TEE_ERROR_STORAGE_NO_SPACE` error code.
     - If output values result from these actions, then either apply the mechanism to handle variable length return values or generate and write the content of the output data to the output buffer and assign the number of written bytes to the “size of content” indicator.
   - If the *Authority* parameter is not null, store it as an object in the SD’s private storage, and identify it using the “SD Authority information” object identifier (see section A.9).
   - Add the newly created SD with its privileges (and their scopes) to the list of SDs directly associated with SD-T. If the *Privileges* parameter value indicates that the newly created SD is a root SD (i.e. its `isRootSD` field value is set to TRUE) then set the `gpd.sd.isRootSD` property of the SD to TRUE. If there is not enough memory to register the newly created SD, reject the operation with the `TEE_ERROR_OUT_OF_MEMORY` error code.
   - Commit the operation by setting the newly created SD’s life cycle state according to the *Initial state* parameter value.

6.3.2 Uninstall Security Domain

Uninstall SD is a privileged operation that deletes an installed Security Domain and performs the necessary memory cleanup.

If a recursive *Uninstall SD* operation is interrupted, any remaining SD SHALL have a parent SD.

The Security Domain in charge of this operation (SD-P) performs the following actions:

1. Unwrap the operation parameters and extract the following parameters:
    - The *Security Domain UUID* that identifies the SD to uninstall (denoted SD-T)
    - The *Recursive* flag indicating a recursive removal of all sub-domains of SD-T (under the conditions specified below – see item #3)
    - (Optional) The *Authorization Token* (explicit authorization only)
2. Identify SD-A as described in section 6.1.2.
3. Verify that the pre-conditions to uninstall SD-T are satisfied; otherwise reject the operation with the `TEE_ERROR_ACCESS_DENIED` error code (only when no specific code is mentioned):
    - Check that the *Security Domain UUID* corresponds to an existing SD; otherwise reject the operation with the `TEE_ERROR_ITEM_NOT_FOUND` error code.
    - If the *Recursive* flag value is set to TRUE, then verify that:
        - SD-T has the `gpd.sd.isRootSD` property set to TRUE (it is an rSD).
        - SD-T and any of its directly/indirectly associated SD has no child TA.
    - If the *Recursive* flag value is set to FALSE, then check that SD-T has neither child TA nor child SD.
    - If SD-T has its `gpd.sd.isRootSD` property set to TRUE, then check that SD-T is in the range of the SD-A's scope of control for the `gpd.privilege.rsdManagement` privilege; otherwise perform the check for the `gpd.privilege.sdManagement` privilege – in both cases, by applying the rules defined in sections 4.1.3.2 and 4.1.3.3.
4. Atomically uninstall SD-T:
    - Remove SD-T from the list of SDs directly associated with its parent SD to avoid concurrent access while deleting associated data.
    - If the *Recursive* flag value is set to TRUE, then remove all existing SDs from the list of SDs directly or indirectly associated with SD-T and delete all data of their private storage created since their installation.
    - Delete all data created in SD-T's private storage since its installation.
    - If there is an opened Secure Channel with SD-P and if the operation consists of uninstalling SD-P (e.g. when SD-T = SD-P, or when SD-P is in the list of the recursively uninstalled SDs) then this Secure Channel is automatically closed by the TEE at the end of this operation.

6.3.3 Block SD

Block SD is a privileged operation that updates the life cycle state of a Security Domain (see section 4.4).

The Security Domain in charge of this operation (SD-P) performs the following actions:

1. Unwrap the operation parameters and extract the following parameters:
   - The *Security Domain UUID* that identifies the SD to block
   - (Optional) The *Authorization Token* (explicit authorization only)
2. Identify SD-A as described in section 6.1.2.
3. Verify that the pre-conditions to block the Security Domain are satisfied; otherwise reject the operation with the `TEE_ERROR_ACCESS_DENIED` error code (only when no specific code is mentioned):
   - Check that the *Security Domain UUID* parameter corresponds to an existing SD (denoted SD-T) on the TEE; otherwise reject the operation with the `TEE_ERROR_ITEM_NOT_FOUND` error code.
   - Check that SD-T is different from SD-P.
   - Check that SD-T is in the range of the SD-A's scope of control for the `gpd.privilege.sdManagement` privilege – by applying the rules defined in sections 4.1.3.2 and 4.1.3.3.
   - If SD-T is already in the Blocked life cycle state, then return immediately with the `TEE_SUCCESS` return code.
4. Atomically perform these operations:
   - Inactivate all TAs directly associated with SD-T:
     - If any direct child TA has currently running sessions, these must be shut down according to the procedure specified in Chapter 11.
     - Record the current life cycle state of each direct child TA – the TA must be restored to this state when SD-T is unblocked – and assign it the *Inactive* life cycle state.
   - Record the current life cycle state of SD-T – the SD must be restored to this state when it is unblocked.
   - Move SD-T to the Blocked life cycle state.

Backward Compatibility

Version 1.0 described actions to take if `lockflag`, incorrectly referred to as the *Inactive Flag*, was set.

Descriptions of the state transition in this document state that all Trusted Applications in the target Security Domain will be locked when the Security Domain enters the Blocked State. This behavior is consistent with the flag always being set.

In version 1.1 this flag is marked as RFU and implementations are not required to check its value.

6.3.4 Unblock SD

Unblock SD is a privileged operation that updates the life cycle state of a Security Domain (see section 4.4).

The Security Domain in charge of this operation (SD-P) performs the following actions:

1. Unwrap the operation parameters and extract the following parameters:
   - The *Security Domain UUID* that identifies the SD to unblock
   - (Optional) The *Authorization Token* (explicit authorization only)
2. Identify SD-A as described in section 6.1.2.
3. Verify that the pre-conditions to unblock the Security Domain (denoted SD-T) are satisfied; otherwise reject the operation with the `TEE_ERROR_ACCESS_DENIED` error code (only when no specific code is mentioned):
   - Check that the *Security Domain UUID* parameter corresponds to an existing SD (denoted SD-T) on the TEE; otherwise reject the operation with the `TEE_ERROR_ITEM_NOT_FOUND` error code.
   - Check that SD-T is different from SD-P.
   - Check that SD-T is in the range of the SD-A's scope of control for the `gpd.privilege.sdManagement` privilege – by applying the rules defined in sections 4.1.3.2 and 4.1.3.3.
   - Check that SD-T is in the Blocked life cycle state; otherwise return immediately with the `TEE_SUCCESS` return code.
4. Atomically perform these operations:
   - If a direct child TA of SD-T is in the *Inactive* state, restore its life cycle state with the state memorized when SD-T was blocked.
   - Restore SD-T's life cycle state with the state memorized when it was blocked. If there is no memorized state (i.e. the SD was blocked when installed), move SD-T to the *Active* life cycle state.

6.3.5 Restrict SD

Restrict SD is a privileged operation that updates the life cycle state of a Security Domain (see section 4.4).

The Security Domain in charge of this operation (SD-P) performs the following actions:

1. Unwrap the operation parameters and extract the following parameters:
    - The *Security Domain UUID* that identifies the SD to restrict.
    - (Optional) The *Authorization Token* (explicit authorization only)
2. Identify SD-A as described in section 6.1.2.
3. Verify that the pre-conditions to restrict the Security Domain (denoted SD-T) are satisfied; otherwise reject the operation with the `TEE_ERROR_ACCESS_DENIED` error code (only when no specific code is mentioned):
    - Check that the *Security Domain UUID* corresponds to an existing SD; otherwise reject the operation with the `TEE_ERROR_ITEM_NOT_FOUND` error code.
    - Check that SD-T is in the range of the SD-A's scope of control for the `gpd.privilege.sdManagement` or the `gpd.privilege.sdPersonalization` privilege – by applying the rules defined in sections 4.1.3.2 and 4.1.3.3.
    - If SD-T is already in the *Restricted* life cycle state, then return immediately with the `TEE_SUCCESS` return code.
    - Check that this SD is in the *Active* life cycle state; otherwise reject the operation with the `TEE_ERROR_BAD_STATE` error code.
4. Atomically modify SD-T's life cycle state:
    - Move SD-T to the *Restricted* life cycle state.

6.3.6 Unrestrict SD

Unrestrict SD is a privileged operation that updates the life cycle state of a Security Domain (see section 4.4).

The Security Domain in charge of this operation (SD-P) performs the following actions:

1. Unwrap the operation parameters and extract the following parameters:
    - The *Security Domain UUID* that identifies the SD to unrestrict
    - (Optional) The *Authorization Token* (explicit authorization only)
2. Identify SD-A as described in section 6.1.2.
3. Verify that the pre-conditions to unrestrict the Security Domain (denoted SD-T) are satisfied; otherwise reject the operation with the `TEE_ERROR_ACCESS_DENIED` error code (only when no specific code is mentioned):
    - Check that the *Security Domain UUID* corresponds to an existing SD; otherwise reject the operation with the `TEE_ERROR_ITEM_NOT_FOUND` error code.
    - Check that SD-T is in the range of the SD-A's scope of control for the `gpd.privilege.sdManagement` or the `gpd.privilege.sdPersonalization` privilege – by applying the rules defined in sections 4.1.3.2 and 4.1.3.3.
    - If SD-T is already in the *Active* life cycle state, then return immediately with the `TEE_SUCCESS` return code.
    - Check that SD-T is in the *Restricted* life cycle state; otherwise reject the operation with the `TEE_ERROR_BAD_STATE` error code.
4. Atomically modify SD-T's life cycle state:
    - Move SD-T to the *Accessible* life cycle state.

6.4 Privileged Operations Common to TA and SD

6.4.1 Store Data

Store Data is a privileged operation used to personalize either a Trusted Application or a Security Domain. It creates a persistent object with attributes and/or data stream content in the `TEE_STORAGE_PERSO` storage of the TA or the private storage of the SD.

> The Store Data command takes a `StoredDataObject` structure as an input. This may be used to encode key generation parameters for an asymmetric key pair. The Store Data command must accept such requests, although it does not return the public key. The caller must use Fetch Object command (introduced in version 1.1) to retrieve the public key.

The Security Domain in charge of this operation (SD-P) performs the following actions:

1. Unwrap the operation parameters and extract the following parameters:
   - The *Application UUID* that identifies either the TA or the SD to personalize
   - The *Decryption parameter* when the *Object* parameter is passed encrypted
   - The *Object* to be stored in the personalization storage of the TA or SD. This object consists of:
     - An *object identifier*
     - An *object type*
     - An *access attribute* made of a combination of access control and sharing permissions flags
     - A *list of attributes* defining the attributes values of the object when referring to a key or key-pair object as defined in [TEE Core API]
     - A possible data stream associated with the object as defined in [TEE Core API]
     - The possible metadata associated with the object when referring to a key or key-pair object as defined in [TEE Core API]
   - (Optional) The *Authorization Token* (explicit authorization only)

2. Identify SD-A as described in section 6.1.2.

> The existence in the TEE of the UUID value of the *Application UUID* parameter SHOULD first be checked to determine the SD-A's privilege required by this operation; if this UUID does not correspond to any TA or SD in the TEE, reject the operation with the `TEE_ERROR_ITEM_NOT_FOUND` error code.

3. Verify that the pre-conditions to store persistent data are satisfied; otherwise reject the operation with the `TEE_ERROR_ACCESS_DENIED` error code (only when no specific code is mentioned below):
   - If the *Application UUID* parameter corresponds to an existing TA, verify the conditions:
     - The TA SHALL be in the range of the SD-A's scope of control for the `gpd.privilege.taManagement` or the `gpd.privilege.taPersonalization` privilege – by applying the rules defined in sections 4.1.3.2 and 4.1.3.3.
     - The TA SHALL NOT be in the *Inactive* life cycle state; otherwise reject the operation with the `TEE_ERROR_BAD_STATE` error code.

- Otherwise the *Application UUID* parameter SHALL correspond to an existing SD; then verify the conditions:
    - The SD SHALL be in the range of the SD-A's scope of control for the `gpd.privilege.sdManagement` or the `gpd.privilege.sdPersonalization` privilege – by applying the rules defined in sections 4.1.3.2 and 4.1.3.3.
    - The SD SHALL NOT be in the Blocked life cycle state; otherwise reject the operation with the `TEE_ERROR_BAD_STATE` error code.
- If the *Decryption* parameter is not null, then extract the necessary information to decipher the ciphered text encoding the *Object* parameter.
    - The key identifier of the *Decryption* parameter refers to a key object owned by the direct parent SD in case of a TA, or owned by SD-P in case of an SD; if decryption fails then reject the operation with the `TEE_ERROR_ACCESS_DENIED` error code.
- Verify that the *Object* parameter value is consistent; otherwise reject the operation with either the `TEE_ERROR_BAD_FORMAT` error code or, if the TEE implementation does not support the type or length of an attribute, with the `TEE_ERROR_NOT_SUPPORTED` error code:
    - Check that the *access attribute* of the *Object* parameter is valid according to the values defined by [TEE Core API] Table 5-2 and the constraints defined by section 5.5 of this specification.
    - Check that the *object type* is a value as defined in [TEE Core API] Table 6-13.
    - If the list of attributes is not empty, then:
        - For each attribute:
            - Check that its *identifier* and the format of its value conform to [TEE Core API] Tables 6-15, 6-16, 6-17, and 6-18.
            - Check that no mandatory attribute is missing for the specified *object type*.
- Determine the kind of operation to be performed on the `TEE_STORAGE_PERSO` storage of the TA or the private storage of the SD:
    - If an object with the same identifier as specified by the *Object* parameter already exists in this storage:
        - If this object has the same type, then this operation will attempt to replace it.
        - Otherwise reject the operation with the `TEE_ERROR_ACCESS_DENIED` error code.
    - Otherwise this operation will attempt to create a new permanent object.

4. Atomically, according to the operation to be performed:
    - If the operation would take the TEE beyond its implementation limits, reject the operation with the `TEE_ERROR_LIMIT_EXCEEDED` error code.
    - Depending on the kind of operation to be performed (see last bullet of step 3), create or replace the permanent object in the `TEE_STORAGE_PERSO` storage space of the TA or in the private storage of the SD. If the storage is unreachable, reject the operation with the `TEE_ERROR_STORAGE_NOT_AVAILABLE` error code; if it is corrupted, reject the operation with the `TEE_ERROR_CORRUPT_OBJECT` error code.

6.4.2 Delete Data

Delete Data is a privileged operation used to remove an object previously stored (using the Store Data command) in the `TEE_STORAGE_PERSO` storage of a Trusted Application or the private storage of a Security Domain. Attributes, metadata, and/or data stream content of the object are removed during this operation.

The Security Domain in charge of this operation (SD-P) performs the following actions:

1. Unwrap the operation parameters and extract the following parameters:
   - The *Application UUID* that identifies either the TA or the SD to personalize
   - The *Object Identifier* that identifies the object to remove from the personalization storage of the TA or SD
   - (Optional) The *Authorization Token* (explicit authorization only)

2. Identify SD-A as described in section 6.1.2.

> The existence in the TEE of the UUID value of the *Application UUID* parameter SHOULD first be checked to determine the SD-A's privilege required by this operation; if this UUID does not correspond to any TA or SD in the TEE, reject the operation with the `TEE_ERROR_ITEM_NOT_FOUND` error code.

3. Verify that the pre-conditions to store persistent data are satisfied; otherwise reject the operation with the `TEE_ERROR_ACCESS_DENIED` error code (only when no specific code is mentioned):
   - If the *Application UUID* parameter corresponds to an existing TA, verify the conditions:
     - The TA SHALL be in the range of the SD-A's scope of control for the `gpd.privilege.taManagement` or the `gpd.privilege.taPersonalization` privilege – by applying the rules defined in sections 4.1.3.2 and 4.1.3.3.
     - The TA SHALL NOT be in the *Inactive* life cycle state; otherwise reject the operation with the `TEE_ERROR_BAD_STATE` error code.
   - Otherwise the *Application UUID* parameter SHALL correspond to an existing SD; then verify the conditions:
     - The SD SHALL be in the range of the SD-A's scope of control for the `gpd.privilege.sdManagement` or the `gpd.privilege.sdPersonalization` privilege – by applying the rules defined in sections 4.1.3.2 and 4.1.3.3.
     - The SD SHALL NOT be in Blocked life cycle state; otherwise reject the operation with the `TEE_ERROR_BAD_STATE` error code.
     - Verify that the *Object Identifier* parameter corresponds to an existing object in the `TEE_STORAGE_PERSO` storage of the TA or the private storage of the SD; otherwise reject the operation with the `TEE_ERROR_ITEM_NOT_FOUND` error code.

4. Atomically, according to the operation to be performed:
   - Remove the permanent object (possibly the attributes, metadata, and data stream) from the `TEE_STORAGE_PERSO` storage space of the TA or the private storage of the SD. If the storage is unreachable then reject the operation with the `TEE_ERROR_STORAGE_NOT_AVAILABLE` error code.

6.4.3 List Objects

List Objects is a privileged operation used to get the list of objects of a Trusted Application or a Security Domain that are currently stored in the `TEE_STORAGE_PERSO` storage space. A list of the object identifiers is returned.

The Security Domain in charge of this operation (SD-P) performs the following actions:

1. Unwrap the operation parameters and extract the following parameters:
   - The *Application UUID* that identifies either the TA or the SD to retrieve the objects for
   - (Optional) The *Authorization Token* (explicit authorization only)

2. Identify SD-A as described in section 6.1.2.

The existence in the TEE of the UUID of the *Application UUID* parameter SHOULD first be checked to determine the requested privilege of SD-A; if the corresponding TA or SD does not exist in the TEE, reject the operation with the `TEE_ERROR_ITEM_NOT_FOUND` error code.

3. Verify that the pre-conditions to retrieve the list of objects are satisfied; otherwise reject the operation with the `TEE_ERROR_ACCESS_DENIED` error code (only when no specific code is mentioned):
   - If the *Application UUID* parameter corresponds to an existing TA, verify the conditions:
     - The TA SHALL be in the range of the SD-A's scope of control for the `gpd.privilege.taManagement` or the `gpd.privilege.taPersonalization` privilege – by applying the rules defined in sections 4.1.3.2 and 4.1.3.3.
     - The TA SHALL NOT be in the *Inactive* life cycle state; otherwise reject the operation with the `TEE_ERROR_BAD_STATE` error code.
   - Otherwise the *Application UUID* parameter SHALL correspond to an existing SD; then verify the conditions:
     - The SD SHALL be in the range of the SD-A's scope of control for the `gpd.privilege.sdManagement` or the `gpd.privilege.sdPersonalization` privilege – by applying the rules defined in sections 4.1.3.2 and 4.1.3.3.
     - The SD SHALL NOT be in Blocked life cycle state; otherwise reject the operation with the `TEE_ERROR_BAD_STATE` error code.

4. Atomically, return the list of object identifiers.
   - If the operation would take the TEE beyond its implementation limits, reject the operation with the `TEE_ERROR_LIMIT_EXCEEDED` error code.
   - If the `TEE_STORAGE_PERSO` storage of the TA or the private storage of the SD is unreachable, reject the operation with the `TEE_ERROR_STORAGE_NOT_AVAILABLE` error code; if it is corrupted, reject the operation with the `TEE_ERROR_CORRUPT_OBJECT` error code.
   - Determine the required length of the list of object identifiers to be returned – apply the mechanism to handle variable length return values, if any.
   - Generate and write the content of the list of object identifiers to the buffer provided and return the number of bytes written in the "size of content" indicator.

6.4.4 Fetch Object

New in version 1.1

Fetch Object is a privileged operation that returns a public object that is currently stored in the private storage area of a Security Domain.

The Security Domain in charge of this operation (SD-P) performs the following actions:

1. Unwrap the operation parameters and extract the following parameters:
    - The *Security Domain UUID* that identifies the SD that owns the key.
    - The *object identifier* of the object to return (key 1)
    - The *object identifier* of the object used to sign or MAC the returned data (key 2)
    - The algorithm identifier of the mechanism to use.
    - (Optional) The *Authorization Token* (explicit authorization only)
2. Identify SD-A as described in section 6.1.2.
3. Verify that the pre-conditions to return the data object are satisfied; otherwise, reject the operation with the `TEE_ERROR_ACCESS_DENIED` error code (only when no specific code is mentioned):
    - Check that the *Security Domain UUID* corresponds to an existing SD; otherwise reject the operation with the `TEE_ERROR_ITEM_NOT_FOUND` error code.
4. Identify object to be returned (key 1)
    - If the `objectidentifier` refers to a public key, return all the key data.
    - If the `objectidentifier` refers to an asymmetric key pair, return only the public key.
    - If the `objectidentifier` refers to any other object, reject the operation with the `TEE_ERROR_ACCESS_DENIED` error code.
5. Identify the key to be used to sign the returned value (key 2). If the null identifier is specified, do not add a signature.
    - If the `objectidentifier` refers to a private key or key pair with usage flag `TEE_USAGE_SIGN`, sign the structure using the private key and the specified signing operation.
    - If the `objectidentifier` refers to a symmetric key with the usage flag `TEE_USAGE_MAC`, create a Message Authentication Code using the specified algorithm.
    - If the key does not have an appropriate usage flag, return the error `TEE_ERROR_ACCESS_DENIED`.
6. If output values result from these actions, then either apply the mechanism to handle variable length return values or generate and write the content of the output data to the output buffer and assign the number of written bytes to the “size of content” indicator.

6.5 Privileged Operations on TEE

6.5.1 Lock TEE

Lock TEE is a privileged operation that updates the life cycle state of the TEE preventing from opening new sessions with:

- Any Trusted Applications
- Or any Security Domains with no `gpd.privilege.teeManagement` privilege

The Security Domain in charge of this operation (SD-P) performs the following actions:

1. Extract the optional *Authorization Token* (explicit authorization only).
2. Identify SD-A as described in section 6.1.2.
3. Verify that the pre-conditions to lock the TEE are satisfied:
    - Check that the TEE is in the `TEE_SECURED` life cycle state.

    If the verifications fail, reject the operation with the `TEE_ERROR_ACCESS_BAD_STATE` error code.
4. Atomically Lock the TEE:
    - Internally flag that the TEE is locking, then shut down all open sessions with any TA in the TEE as specified in Chapter 11.
    - Move the TEE to the `TEE_LOCKED` life cycle state.

6.5.2 Unlock TEE

Unlock TEE is a privileged operation that updates the life cycle state of the TEE, once again allowing the opening of new sessions with any Trusted Application or Security Domains.

The Security Domain in charge of this operation (SD-P) performs the following actions:

1. Extract the optional *Authorization Token* (explicit authorization only).
2. Identify SD-A as described in section 6.1.2.
3. Verify that the pre-conditions to unlock the TEE are satisfied:
    - Check that the TEE is in the `TEE_LOCKED` life cycle state.

    If the verifications fail, reject the operation with the `TEE_ERROR_BAD_STATE` error code.
4. Atomically move the TEE life cycle state to the `TEE_SECURED` life cycle state.

6.5.3 Store TEE Property

Store TEE Property is a privileged operation used to personalize the TEE itself. Its primary use case is in support of the TEE TA Debug specification (see [TEE TA Debug]) and *Factory Reset* operation (see section 6.5.4).

The Security Domain in charge of this operation (SD-P) performs the following actions:

1. Unwrap the operation parameters and extract the following parameters:
   - The *Property* to be stored. A property is defined by the triplet (name, type, value) as defined by the Property type in section 8.3.3.9.
   - (Optional) The *Authorization Token* (explicit authorization only)
2. Identify SD-A as described in section 6.1.2.
3. Verify that the pre-conditions to store persistent data are satisfied; otherwise reject the operation with the `TEE_ERROR_ACCESS_DENIED` error code (only when no specific code is mentioned):
   - The TEE SHALL restrict the ability of a client to update non-modifiable properties or to create nonstandard `gpd.tee.*` properties. Nevertheless, it is allowed to create new properties outside the domain name `gpd.tee.*`.
   - For any modifiable `gpd.tee.*` property, check that its type and value are consistent with the type of value as specified in the related GlobalPlatform standard API specification that defines this property (e.g. [TEE Core API], [TEE TA Debug], this specification, etc.).
   - Reject any malformatted property value with the `TEE_ERROR_BAD_FORMAT` error code.
   - Reject any oversized property value with the `TEE_ERROR_EXCESS_DATA` error code or the `TEE_ERROR_LIMIT_EXCEEDED` error code if the operation would take the TEE beyond its implementation limits.
4. Atomically create and initialize (or update) the property in the `TEE_PROPSET_TEE_IMPLEMENTATION` property set.

6.5.4 Factory Reset

Factory Reset is a privileged operation that moves the TEE to a notional “factory” state.

The state itself is identified by Security Domains being listed by UUID in a table held in the property `gpd.tee.tmf.resetpreserved.entities`, or being Trusted Applications directly controlled by one of those listed Security Domains.

The binary format of the list held in property `gpd.tee.tmf.resetpreserved.entities` is described in section 8.7.4. The list may be manipulated by invoking the Store TEE Property command (see section 8.7.3). A TEE MAY choose to defer validation of the property value until *Factory Reset* is invoked. The *Factory Reset* operation does not affect the value held in `gpd.tee.tmf.resetpreserved.entities`.

If *Factory Reset* is interrupted, all TAs and SDs in the TEE SHALL be left without missing controlling SDs. (In practice, an Actor invoking *Factory Reset* may wish to check the operation’s return status, and retry *Factory Reset* if the return code is unexpected or not received.)

The Security Domain in charge of this operation (SD-P) performs the following actions:

1. Extract the optional *Authorization Token* (explicit authorization only).
2. Identify SD-A as described in section 6.1.2.
3. Verify that the pre-conditions to factory reset the TEE are satisfied; otherwise reject the operation with the `TEE_ERROR_ACCESS_DENIED` error code (only when no specific code is mentioned):
    - Check that SD-P is itself marked as to be preserved across an invocation of *Factory Reset*.
4. Modify the TEE state according to the following requirements:
    - All SDs not listed in `gpd.tee.tmf.resetpreserved.entities`, and all TAs not directly associated with any of the SDs listed in `entities` SHALL become uninstalled, according to the procedure detailed in step 4 of the *Uninstall TA* and *Uninstall SD* operations (see sections 6.2.2 and 6.3.2 respectively).
    - All SDs listed in `gpd.tee.tmf.resetpreserved.entities` are retained unmodified.
    - For each SD listed in `gpd.tee.tmf.resetpreserved.entities`, the system SHALL act as if all the SDs in the path from the root of the hierarchy to this listed SD were implicitly listed (and therefore SHALL be retained unmodified).
    - If the operation is interruptible, the modification SHALL be ordered such that no SD or TA can be left unassociated with an ancestor SD.
    - All TAs which are directly associated with an SD listed in `gpd.tee.tmf.resetpreserved.entities` SHALL be reset according to the following requirements:
        - All active TEE Client or TEE Internal sessions are terminated. If the administration session used to perform the *Factory Reset* operation is terminated, then the factory reset SHALL continue.
        - All data (if any) in the `TEE_STORAGE_PERSO` storage space is retained unmodified.
        - All data (if any) in the `TEE_STORAGE_PRIVATE` storage space is removed atomically.

    Warning: Future TEE specifications may add new storage IDs that are not mentioned in this document. Consult those specifications to determine how the new storage IDs react to factory reset.

6.6 Unprivileged Audit Operations

The subsequent audit operations are unprivileged operations; i.e. any Actor, whether authenticated or not, may invoke the command that performs the audit operation to retrieve the expected information.

These audit commands can be submitted to:

- The TMF audit SD (as mentioned in section 4.5) regardless of the TEE life cycle state
    - This audit SD is identified on the TEE by a reserved UUID value defined in section A.3.
- Any accessible SD (i.e. not in the Blocked life cycle state) provided that the TEE is in the `TEE_SECURED` life cycle state
- Any accessible SD (i.e. not in the Blocked life cycle state) with the `gpd.privilege.teeManagement` privilege when the TEE is in the `TEE_LOCKED` life cycle state

If an audit operation command is submitted with an Authorization Token, the procedure described in section 6.1.2 for unprivileged operations SHALL apply before performing any operation-specific action listed in the following sub-sections.

Operation Return Codes

When the TEE cannot read the internal information to be returned by the command performing one of any subsequent audit operations, the `TEE_ERROR_INTERNAL` error code is returned.

When the response demands more space than the TEE is able to provide in a single response, then the `TEE_ERROR_LIMIT_EXCEEDED` error code is returned.

When the response exceeds the output buffer's capacity, then the mechanism described in section 6.1.4 SHALL be applied.

6.6.1 Get TEE Definition

The *Get TEE Definition* operation returns information about the device and the trusted operating system running the TEE, as well as the supported optional GlobalPlatform standard APIs and optionally some TEE properties values (`gpd.tee.*`).

The following actions are performed:

1. Determine the required length of the TEE characteristics data (encoding defined by section 9.1.6) to be returned – apply the mechanism to handle variable length return values, if any.
2. Generate and write the content of the TEE characteristics data into the output buffer and return the number of bytes written in the "size of content" indicator.

6.6.2 Get SD Definition

The *Get SD Definition* operation returns the SD information about its current life cycle state, its remote entity, and its parent SD and direct sub-domains identifiers.

The following actions are performed:

1. Unwrap the operation parameters:
   - The *Security Domain UUID* that identifies the SD to retrieve the definition for
2. Verify that the specified UUID refers to an existing Security Domain, returning the `TEE_ERROR_ITEM_NOT_FOUND` error code if it does not.
3. Determine the required length of the SD characteristics data (encoding defined by section 9.2.2) to be returned – apply the mechanism to handle variable length return values, if any.
4. Generate and write the content of the SD characteristics data to the buffer provided and return the number of bytes written in the “size of content” indicator.

6.6.3 Get List of Trusted Applications

The *Get List of Trusted Applications* operation returns the list of UUIDs of all the TAs directly and (optionally) indirectly associated with an SD.

The following actions are performed:

1. Unwrap the operation parameters and extract the following parameter:
   - The *Security Domain UUID* that identifies the SD that the list of TA is retrieved from.
2. Verify that the *Security Domain UUID* parameter refers to an existing Security Domain, returning the `TEE_ERROR_ITEM_NOT_FOUND` error code if it does not.
3. Determine the number of TAs that exist and the length of the required buffer to return the list of UUIDs (encoding defined by section 8.8.3.2) – apply the mechanism to handle variable length return values, if any.
4. Generate and write the list of Trusted Applications UUIDs to the buffer provided and return the number of bytes written in the “size of content” indicator.

6.6.4 Get TA Definition

The *Get TA Definition* operation returns the TA information about its current life cycle state, its version number, and its parent SD identifier.

The following actions are performed:

1. Unwrap the operation parameters and extract the following parameter:
    - The *Trusted Application UUID* that identifies the TA to retrieve the definition for
2. Verify that the *Trusted Application UUID* parameter refers to an existing TA, otherwise return the `TEE_ERROR_ITEM_NOT_FOUND` error code.
3. Determine the required length of the Trusted Application characteristics data (encoding defined by section 9.3.2) to be returned – apply the mechanism to handle variable length return values, if any.
4. Generate and write the content of the TA characteristics data to the buffer provided and return the number of bytes written in the “size of content” indicator.

6.6.5 Get TA Definition 1

New in version 1.1

The *Get TA Definition 1* operation returns the TA information about its current life cycle state, its version number, and its parent SD identifier.

The following actions are performed:

1. Unwrap the operation parameters and extract the following parameters:
    - The *Trusted Application UUID* that identifies the TA to retrieve the definition for
    - The version of the structure
2. Verify that the *Trusted Application UUID* parameter refers to an existing TA, otherwise return the `TEE_ERROR_ITEM_NOT_FOUND` error code.
3. Determine the required length of the Trusted Application characteristics data (encoding defined by section 9.3.3) to be returned – apply the mechanism to handle variable length return values, if any.
4. Generate and write the content of the TA characteristics data to the buffer provided and return the number of bytes written in the “size of content” indicator.

7 TLV Encoding Rules and Grammar

The encoding of administration messages (including the commands and their parameters) defined by this specification is based on the ITU-T X.680 ([ASN.1]) and X.690 ([ASN.1 Encoding]) series of specifications and reuses some of the defined types and rules. By using a small subset of the ASN.1/DER language to describe all the structures defined in this document, the risk of security threats inherent to the usage of a "weak" context-free language (mainly due to the non-deterministic 'Length' fields) is negligible.

The following types, including their identifier octets (tags), definitions, and encodings, have been reused from the above-mentioned specifications.

Table 7-1: Types Reused from ITU-T X.680 Standard

Types	Tag Value	Formal Definition According to [ASN.1]
`BOOLEAN`	`0x01`	A simple type with two distinguished values (true and false).
`INTEGER`	`0x02`	A simple type with distinguished values which are positive and negative whole numbers, including zero.
`OCTET STRING`	`0x04`	A simple type whose distinguished values are an ordered sequence of zero, one, or more octets, each octet being an ordered sequence of eight bits.
`NULL`	`0x05`	A simple type consisting of a single value, also called null.
`PrintableString`	`0x13`	A character string type defining 'printable' characters (Table 10 of [ASN.1]).
`UTF8String`	`0x0c`	A character string type defining UTF-8 characters (sub-clause 41.16 in [ASN.1]) defined by ISO/IEC 10646 Annex D.

`SEQUENCE`
`SEQUENCE OF`
	`0x30`	A type defined by referencing a fixed, ordered list of types (some of which may be declared to be optional); the value of a sequence type is an ordered list of values, one from each component type. *This ordered list of values follows the order of declaration of types in the SEQUENCE.*

`CHOICE`	The tag value of one of the types list	A type defined by referencing a list of distinct types. Example: `CHOICE {paramtype BOOLEAN, defaulttype NULL}`, the tag value is either `0x01` or `0x05`.

To describe these messages and commands, the grammar from [ASN.1] and the DER encoding rules from [ASN.1 Encoding] have been adopted in parts as well. This means in particular that:

- The 'definite form' of the 'Length Octets' encoding is used as described in section 7.6.
- The BOOLEAN 'TRUE' value is the one-byte value `0xFF`.
- The endianness of any encoded TLV value is 'big-endian' (as stated in Chapter 3).

7.1 Future Type Extensions

For readability, this specification does not use parameterized types or the complex syntax of the 'open' types as permitted by [ASN.1].

More specifically, when an 'open' type (formerly described by the 'ANY' or 'ANY DEFINED BY' syntax, then replaced in [ASN.1] by 'information class' objects syntax) is necessary to represent any type, this specification uses an OCTET STRING type to represent it.

The value octets of this OCTET STRING type SHALL contain the TLV structure (DER encoded) of the type instantiating this 'open' type. The notation adopted from [ASN.1 Constraint], i.e. OCTET STRING (CONTAINING <Type>), SHALL be used to formalize this constraint on such an OCTET STRING value.

The general encoding structure is given by TLV structure of the form:

Table 7-2: Structure of TLV Encoding

Identifier Octets (Tag)	Length Octets (Length)	Value Octets (Value)

7.2 Identifier Octets

Identifier octets define the tag value of the defined type. The tag value consists of one or multiple octets, where a single octet adapts the following definition from the ITU-T X.690 standard ([ASN.1 Encoding]).

Figure 7-1: Tag with One Identifier Octet (Low Tag Number)

Bits	8	7	6	5	4	3	2	1
	Class		P/C	Tag Number in Range [0 – 30]				

Class (bits 8-7), P (Primitive, bit 6=0) / C (Constructed, bit 6=1), and Tag Number (bits 5-1) are adopted from [ASN.1 Encoding].

Whereas a Primitive type is a generic type such as BOOLEAN, INTEGER, PrintableString, etc., a type is Constructed when the Value Octets are made of a series of TLV encodings (e.g. a SEQUENCE type tag).

The first two bits encode the tag class that define the *Universal* (00b), *Application* (01b), *Context-specific* (10b), and *Private* (11b) types according to [ASN.1].

'Implicit' tagging has been used for all context-specific class tags according to [ASN.1] when ambiguities occur in the type definition representing such messages and commands, and have to be resolved (for example, when two optional elements of the same type are consecutive in a sequence of elements). Some context-specific class tag numbers are reserved for this specification (see section 8.2) – when necessary, vendor-specific extensions SHOULD use tag numbers outside the reserved ranges.

7.3 Tag Values Encoded with One Identifier Octet (Low Tag Number)

Most of the types defined by this specification are Application class tags (denoted [APPLICATION <tag-number>]) that are encoded using only one identifier octet as illustrated in Figure 7-1. One identifier octet allows to encode tag numbers in the range [0 – 30] according to [ASN.1 Encoding] (the value 31 – bit 1 to bit 5 all set to 1b – defines a special "marker" that cannot be used).

Tag numbers in range [0 – 30] are reserved for this specification to tag any Primitive or Constructed types of Application and Private class. Vendor implementations SHALL NOT use this range of tag numbers to define their own type extensions.

Any tag value encoded with one identifier octet that does not adopt the standard [ASN.1 Encoding] or that is vendor-specific MAY be ignored by the parser of any compliant implementation of this specification.

The usage and range of tag values encoded with one identifier octet that are allowed by this specification are summarized in the following table:

Table 7-3: Usage and Range of Tag Values Encoded with One Identifier Octet

Usage	**Range of Tag Values (According to [ASN.1 Encoding])**
Tag values defined by this specification or reserved for future use	0x40 – 0x5e (Application class, Primitive type) 0x60 – 0x7e (Application class, Constructed type) 0xc0 – 0xde (Private class, Primitive type) 0xe0 – 0xfe (Private class, Constructed type)
Tag values for proprietary extensions (vendor specific)	N/A (SHOULD use 'Private' class tags of either Primitive or Constructed types encoded with a number of identifier octets greater than one)

7.4 Tag Values Encoded with Two Identifier Octets (High Tag Number)

Figure 7-2: Tag with Two Identifier Octets (High Tag Number)

Bits	8	7	6	5	4	3	2	1	8	7	6	5	4	3	2	1
	Class		P/C	1	1	1	1	1	0	Tag Number in Range [31 – 127]						
	First Octet								**Second Octet**							

All administration commands are defined by types tagged with two identifier octets (see Table 8-7) and encoded according to [ASN.1 Encoding]:

- The first byte value is `0x7f` (for Application class of Constructed types, and all remaining bits set to 1).
- The most significant bit of the second octet is set to `0`, indicating that it is the last byte of the tag value, and the remaining bits encoding a tag number in range [31 – 127].

Tag numbers in range [31 – 127] are reserved for this specification to tag any Primitive or Constructed types of Application class.

A specific vendor implementation could use this range of tag numbers to define new Primitive or Constructed types of 'Private' class only. Any tag value encoded with two identifier octets that does not adopt the standard [ASN.1 Encoding] or that is vendor-specific MAY be ignored by the parser of any compliant implementation of this specification.

The usage and range of tag values with two identifier octets that are allowed by this specification are summarized in the following table.

Table 7-4: Usage and Range of Tag Values Encoded with Two Identifier Octets

Usage	Range of Tag Values (According to [ASN.1 Encoding])	

Tag values defined by this specification or reserved for future use	`[0x7f1f – 0x7fff]`
	`[0x5f1f – 0x5fff]`
		(Application class, Constructed type)
		(Application class, Primitive type)

Tag values for proprietary extensions	`[0xff1f – 0xffff]`
	`[0xdf1f – 0xdfff]`
		(Private class, Constructed type)
		(Private class, Primitive type)

7.5 Tag Values Encoded with More than Two Identifier Octets (High Tag Number)

This specification does not use tag values encoded with more than two identifier octets (for tag numbers equal or greater than 128).

Vendor-specific tags (extensions) may adopt, when necessary, such encodings to avoid any collisions with values already defined or reserved for a future use by this specification. These tags SHOULD be ignored by the parser of any implementation strictly compliant with this specification.

In any case, when performing a cryptographic operation (e.g. a hash function to verify a signature) any data structure encoded with tags that the parser could ignore SHALL NOT be excluded from the input data of the operation.

7.6 Length Octets

Length octets are limited to the 'definite form' only (see [ASN.1 Encoding]) and encoded using either:

- The 'short form', to specify length values up to 127. The most significant bit (bit 8) is set to `0` and the remaining bits encode the actual length value.

Figure 7-3: Length Octets – 'Short Form' Encoding

Bits	8	7	6	5	4	3	2	1
	`0`	Length						

- Or the 'long form', consisting of:
  - o An initial octet where the most significant bit (bit 8) is set to 1 and the remaining bits encode the number of subsequent octets as an unsigned binary integer (with bit 7 as the most significant bit)
  - o The subsequent octets together encode the number of Value Octets as an unsigned binary integer using all available bits of these octets

Figure 7-4: Length Octets – 'Long Form' Encoding

Bits	8	7	6	5	4	3	2	1
	1	N octets						
	First octet of length							
	….							
	Last octet (N) of length							

7.7 Value Octets

Value octets consist of zero, one, or more octets as specified in [ASN.1 Encoding].

8 Administration Commands Encoding

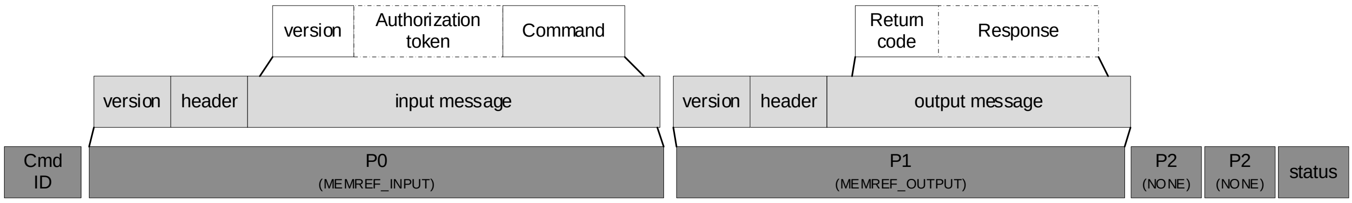
The structure of commands can be split into three different layers:

□ Operation Layer containing authorized operations

░ Security Layer providing authentication, integrity, and confidentiality

■ Transport Layer over TEE Client API protocol

Figure 8-1: Protocol Layers



Transport Layer (section 8.1)

- The Client API protocol ([TEE Client]) SHALL be supported by an implementation of this specification.
- The TEE Internal Core API ([TEE Core API]) MAY also be supported by an implementation of this specification to invoke administration commands from a Trusted Application.
- New protocols could be defined in future releases of this specification.

Security Layer (section 8.2)

- Describe the encoding of the security container that bundles an input (or output) message embedding a command request (or response) payload. A Security Layer implementation may have to perform particular cryptographic, splitting, or reformatting operations to extract or output the command payloads.
- Use the formal Extended Backus-Naur Form (EBNF, ISO/IEC 14977) for type descriptions and TLV format encoding as defined in Chapter 7 for the security container description.
- Provide the generic structures that any future GlobalPlatform TEE specifications can extend to define new mechanisms securing the transport of administration commands.

Operation Layer (section 8.3)

- Describe the encoding of administration commands and their responses.
- Use EBNF for type descriptions and TLV format encoding as defined in Chapter 7 for the command operations and Authorization Tokens.
    - Authorization Tokens are optional (see section 5.2.1). They are part of the applicative layer because they must be evaluated and verified in the applicative context.

8.1 Transport Layer

The Transport Layer uses a single envelope command with two parameters:

- The first parameter is an input buffer containing the security container payload embedding the entire command request payload.
- The second parameter is an output buffer containing the security container payload embedding the command response payload.

Figure 8-2: Single Envelope Command

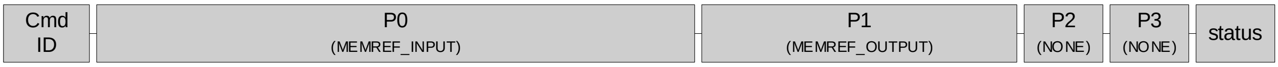


Table 8-1: Envelope Command Encoding

Parameters	Value	Description
Command ID	`0x00C20000`	Envelope used to transport command messages.
Parameter #0	`TEEC_MEMREF_*_INPUT`	Request message including the command payload.
Parameter #1	`TEEC_MEMREF_*_OUTPUT`	Response message including the command response.
Parameter #2	`TEEC_NONE`	Not used
Parameter #3	`TEEC_NONE`	Not used
Status	–	Execution status of the envelope command.

General Considerations

Each administrative command transported over the protocol defined by [TEE Client] (and, when supported by an implementation, by [TEE Core API]) is executed on the TEE in a specific context of an administrative session as defined in section 6.1.

Well-identified by such a context and given that it is not possible to execute multiple concurrent commands within a session, a command response is always synchronously returned as the result of a Client Application’s command request.

TEE Management Framework Origin Code

This specification defines a specific origin code constant value extending the set of values defined in the mandatory Client API protocol [TEE Client] (and, when supported by an implementation, the TEE Internal Core API protocol [TEE Core API]).

The `TEEC_ORIGIN_TRUSTED_SD` (or `TEE_ORIGIN_TRUSTED_SD`) constant value defined by section A.7 SHALL be set as the return code origin of the envelope command response, regardless of whether the destination of the administration command is an SD, or possibly the TMF audit SD (see section 4.5) for unprivileged audit commands.

Reserved Command IDs

When `TEEC_InvokeCommand` is called to send TMF messages to a Security Domain, the following [TEE Client] Command IDs are reserved.

Table 8-1b: Reserved Command IDs

Range	Description
`0x00000000 – 0x00C1FFFF`	Reserved for GlobalPlatform use
`0x00C20000 – 0x00C2FFFF`	Reserved for TMF ASN.1 Profile
`0x00C30000`	JSON OTrP messages
`0x00C30001 – 0x00C3FFFF`	Reserved for TMF OTrP Profile
`0x00C40000 – 0x3FFFFFFE`	Reserved for GlobalPlatform use
`0x3FFFFFFF`	Defined Error value
`0x40000000 – 0xFFFFFFFF`	Implementation defined

8.1.1 Using the Mandatory TEE Client API

Table 8-2: Envelope Command Return Codes Using the TEE Client API Protocol

Return Code	Origin Code	Description
`TEEC_SUCCESS`	`TEEC_ORIGIN_TRUSTED_SD`	The envelope command has successfully been executed and the output buffer embeds the administration command response.
`TEEC_ERROR_SHORT_BUFFER`	`TEEC_ORIGIN_TRUSTED_SD`	The supplied buffer is too short for the generated output.
`TEEC_ERROR_BAD_FORMAT`	`TEEC_ORIGIN_TRUSTED_SD`	Input data of invalid format.
`TEEC_ERROR_GENERIC`	`TEEC_ORIGIN_TRUSTED_SD`	Something failed during the command execution. More information is available in the response message (in P1).
`TEEC_ERROR_BAD_PARAMETERS`	`TEEC_ORIGIN_TRUSTED_SD`	Invalid envelope command parameter.
`TEEC_ERROR_OUT_OF_MEMORY`	`TEEC_ORIGIN_TRUSTED_SD`	Something failed during the command execution, but there were no system resources available to create the response message. Hence there will not be any response data (in P2).
`TEEC_ERROR_TARGET_DEAD`	`TEEC_ORIGIN_TEE`	A Panic occurred in the underlying TEE commands invoked to implement the TMF command.

The TEE Client API protocol (defined in [TEE Client]) requires that a Client Application opens a session with the entity performing the administrative command.

The UUID specified as the *destination* parameter of the `TEEC_OpenSession` function SHALL be the UUID of the Security Domain performing the administrative operation (SD-P).

The envelope command is then sent to this SD by using the `TEEC_InvokeCommand` function. The possible response status codes (defined by Table 8-2) are returned as the `TEEC_RESULT` value of this function (the `returnOrigin` parameter value SHALL be set to the corresponding value as indicated in Table 8-2).

The client interface may produce other standard errors, as defined in Table 4-2 in section 4.4.2 of the TEE Client API v1.0 ([TEE Client]) with errata 2.0 applied or later, and these should be handled appropriately.

8.1.2 Using the Internal Client API of the TEE Internal Core API (Optional)

Table 8-3: Envelope Command Return Codes Using the Internal Client API Protocol

Return Code	Origin Code	Description
`TEE_SUCCESS`	`TEE_ORIGIN_TRUSTED_SD`	The envelope command has successfully been executed and output buffer embeds the administration command response.
`TEE_ERROR_SHORT_BUFFER`	`TEE_ORIGIN_TRUSTED_SD`	The supplied buffer is too short for the generated output.
`TEE_ERROR_BAD_FORMAT`	`TEE_ORIGIN_TRUSTED_SD`	Input data of invalid format.
`TEE_ERROR_BAD_PARAMETERS`	`TEE_ORIGIN_TRUSTED_SD`	Invalid envelope command parameter.
`TEE_ERROR_OUT_OF_MEMORY`	`TEE_ORIGIN_TRUSTED_SD`	Not enough resources are available to perform the envelope command
`TEE_ERROR_TARGET_DEAD`	`TEEC_ORIGIN_TEE`	A Panic occurred in the underlying TEE commands invoked to implement the TMF command.

When supported by an implementation of this specification, the Internal Client API protocol (defined in [TEE Core API] section 4.9) can be used to require that a client Trusted Application opens a session with the entity performing the administrative command.

The UUID specified as the *destination* parameter of the `TEE_OpenTASession` function SHALL be the UUID of the Security Domain performing the administrative operation (SD-P).

The envelope command is then sent to this SD by using the `TEE_InvokeTACommand` function. The possible response status codes (defined by Table 8-3) are returned as the `TEE_RESULT` value of this function (the `returnOrigin` parameter value SHALL be set to the corresponding value as indicated in Table 8-3).

The client interface may produce other standard errors, as defined in Table 3-3 in section 3.3.2 of [TEE Core API] v1.2 or later, and these should be handled appropriately.

8.2 Security Layer

The Security Layer encapsulates the entire operation layer and provides:

- Authentication
- Integrity and authenticity of administrative operations
- Confidentiality when required

To be compliant with this specification, an implementation SHALL embed any command request and response payloads of the operation layer in a security container consisting of:

- A version number identifying a version of this specification
- A secure content made of:
    - A header that is identified by a *mandatory* type value and *possible* data used to process the secure payload content, e.g. a Security Layer identifier, any other protocol identifier (e.g. TLS), a proof of authenticity/integrity of the secure payload, etc.
    - An input or output message embedding the administration command request/response

Security Container

The Security container is defined by the `SecurityContainer` type based on the grammar and encoding rules defined in Chapter 7.

```
SecurityContainer ::=  [APPLICATION 23] SEQUENCE {
          version           INTEGER,
          content           ContainerContent
}
```

Where:

- **`version`** – The version of this specification identified by the `gpd.tee.tmf.version` property (see Table A-4), or any prior version.
- **`content`** – The security container's content defined by the following type:

```
ContainerContent ::= SEQUENCE {
     type        ContainerType,
     headerOCTET STRING OPTIONAL, -- an 'open' type as mentioned in section 7.1
     payload     CHOICE {
                 anyData            [0] OCTET STRING,
                 cmdReqPayload      CmdReqPayload,
                 cmdRespPayload     CmdRespPayload
     }
}
```

Where:

- **`type`** – The mandatory container type that determines the header and payload content values.

  The integer type representing the possible values of the container type is described as follows:

  `ContainerType ::= INTEGER (1..255)` -- *any container type values are in range [1..255] (see Table 8-5)*

Table 8-5 defines a generic container type value defined by this specification:

`admin-generic-cont-type ContainerType ::= 1` -- *the generic ContainerType value equals 1*

So, the `ContainerType` can be constrained to the following type (containing only one value):

`GenericContainerType ::= ContainerType (admin-generic-cont-type)`

Other values can be defined similarly for future or vendor-specific usage.

- **`header`** – The **optional** header information determined by the previous `ContainerType` value. Table 8-5 summarizes the possible header values that define either a generic header defined by this specification (see details below) or any header that future releases of this specification or a proprietary implementation could define to transport any kind of information related to a specific secure protocol.
- **`payload`** – The payload that embeds an administration command or the response to a command. The possible encodings are:
  - An OCTET STRING that represents:
    - Any input data to be processed (deciphered, decompressed, formatted…) to extract the command request payload
    - Any output data built (ciphered, compressed, formatted…) from the returned command response payload
  - A command request payload
  - A command response payload

The general TLV encoding of the `SecurityContainer` is defined as follows.

Table 8-4: SecurityContainer TLV Encoding

Tag	Length	Value Octets					Presence
0x77	L	SecurityContainer value					M
		Tag	**Length**	**Value Octets**			
		0x02	L	gpd.tee.tmf.version (the current value of this property or a value identifying a prior version)			M
		0x30	L	content			M
				Tag	**Length**	**Value Octets**	
				0x02	L	type (see Table 8-5)	M
				0x04	L	header (see Table 8-5)	O
				0x80 *or* 0x60 *or* 0x61	L	anyData *or* cmdReqPayload *or* cmdRespPayload	C

Table 8-5 defines possible container type values and associated headers for `SecurityContainer` TLV encoding.

Table 8-5: Container Content Type and Header Values

Container Type Name	Container Type Value	Header Type	Header Value Octets
`ADMIN_GENERIC_CONT_TYPE`	`0x01`	n/a (header SHALL be absent)	None
Reserved for future use	`[0x02 – 0x7F]`	RFU	The TLV structure (DER encoded) of the RFU Header type
Proprietary extensions (vendor-specific)	`[0x80 – 0xFF]`	proprietary	The TLV structure (DER encoded) of the proprietary Header type

Security Container's Content

Container Type and Header Value

The container type value indicates the kind of security protocol that is applied to the security container's content.

This release of the specification only defines a generic container content identified by the `ADMIN_GENERIC_CONT_TYPE` type value that specifies a container content without any header and embedding a whole administration command/response payload outside any pre-established secure channel.

This is typically the case where the payload transports:

- An unprivileged audit command
  - The payload of the request/response can be in clear.
- A privileged command accompanied with an Authorization Token in a context where no connection with a remote entity is possible
  - The payload of the command request/response may be encrypted in an implementation-defined manner – usage of pre-shared keys…

Formally, this generic container content can be represented by subtyping the `ContainerContent` type as follows:

```
GenericContainerContent ::= ContainerContent
     ( WITH COMPONENTS {
                    type  ( GenericContainerType ), -- type value of type GenericContainerType
                    header ABSENT,
                    payload
     })
```

Where the `ContainerType` value is constrained by the `ADMIN_GENERIC_CONT_TYPE` value, the optional header SHALL be absent and the payload SHALL be present with any value represented by one of the CHOICE types.

Other vendor-specific container type and associated header values may be defined and implemented to support, for example, the exchange of encrypted payloads according to a specific Security Layer. In such a case, the header value may be constrained to contain some protocol information.

Payload Value

The payload value embeds a command request payload sent to or a command response payload received from the TEE entity (i.e. a Security Domain or possibly the TMF audit SD for unprivileged audit commands). It can be encrypted and/or formatted in a way that is only determined by the value of the container type.

When encrypted, the whole decrypted payload SHALL always match a data of type:

- `CmdReqPayload` for a command request (encoding details defined by section 8.3.1).
- `CmdRespPayload` for a command response (encoding details defined by section 8.3.2).

An ASN.1 TMF implementation can support multiple Security Layer protocols, each of them being typically defined by specific extensions of the secure container types and contents. The Audit TEE command can be used to retrieve the list of UUID values identifying these protocols (see the `TrustedOS` structure in section 9.1.5). The generic protocol defined above has its own UUID defined in section A.3.

8.3 Operation Layer

This section defines the GlobalPlatform ASN.1 TMF operation types, their purpose, and their encoding using the grammar notation and encoding rules described in Chapter 7.

Encoding of Privileged Administration Commands

The encoding of any subsequent privileged commands requires that all their parameter values SHALL be present and ordered as defined by the SEQUENCE Constructed type used to describe each command. When a parameter value has to be omitted, it is encoded with the ‘place-holder’ null value (this is depicted by the usage of the CHOICE type in the parameter type definitions).

Thus, each command’s parameter value has a fixed position in the sequence of the parameter values.

This allows a remote Authority that emits an Authorization Token for this command to encode a possible parameter’s constraint value (see section 10.1.2) by walking through the TLV structure of the command without any knowledge of the parameter tag values.

8.3.1 Command Request Payload Encoding

The type `CmdReqPayload` is a Constructed type which describes a request payload containing an administrative command.

```
CmdReqPayload ::= [APPLICATION 0] SEQUENCE {
    version     INTEGER,
    token       AuthorizationToken OPTIONAL,
    command     CHOICE {
                     < choice of any possible command types defined in this specification from
                     section 8.3.4 to section 8.8. This CHOICE type is extensible >,
                     …        -- this extension marker indicates that types representing new
                     future defined commands are supported
                }
}
```

With:

- **`version`** – The version of this specification identified by the `gpd.tee.tmf.version` property (see Table A-4), or any prior version.
  - If supported by the implementation, a command payload request of a previous version of this specification (if any) may be embedded in a security container (see section 8.2) as defined by this version of the specification.
- **`token`** – The **optional** Authorization Token.
  - The `AuthorizationToken` definition and encoding is specified in Chapter 10.
  - When present, the token is part of the command processing as described in section 6.1.6.
- **`command`** – One of the command type values as specified from section 8.3.4 to section 8.8.

The general TLV encoding is defined as follows.

Table 8-6: Command Request Payload TLV Encoding

<table>
<tr><th>Tag</th><th>Length</th><th colspan="3">Value Octets</th><th>Presence</th></tr>
<tr><td rowspan="5">0x60</td><td rowspan="5">L</td><td colspan="3">CmdReqPayload value</td><td>M</td></tr>
<tr><th>Tag</th><th>Length</th><th>Value Octets</th><th></th></tr>
<tr><td>0x02</td><td>L</td><td>gpd.tee.tmf.version
(the current value of this property or a value identifying a prior version)</td><td>M</td></tr>
<tr><td>0x76</td><td>L</td><td>token</td><td>O</td></tr>
<tr><td><CommandTag></td><td>L</td><td>command</td><td>M</td></tr>
</table>

Each `<CommandTag>` value represents a tag encoded with two identifier octets that identifies the operation to be performed.

These tag encodings are defined by the following table:

2173 **Table 8-7: Command Tags Definition**

CommandTag Value	Command Name
0x7f41	Install TA
0x7f42	Uninstall TA
0x7f43	Update TA
0x7f44	Lock TA
0x7f45	Unlock TA
0x7f46	Update TA and Data (**version 1.1 and later**)
[0x7f47 – 0x7f49]	***RFU***
0x7f4a	Install SD
0x7f4b	Uninstall SD
0x7f4c	***RFU***
0x7f4d	Block SD
0x7f4e	Unblock SD
0x7f4f	Restrict SD
0x7f50	Unrestrict SD
[0x7f51 – 0x7f54]	***RFU***
0x7f55	Store Data
0x7f56	Delete Data
0x7f57	List Objects
0x7f58	Fetch Object (**version 1.1 and later**)
0x7f59	***RFU***
0x7f5a	Lock TEE
0x7f5b	Unlock TEE
0x7f5c	Store TEE Property
0x7f5d	Factory Reset
[0x7f5e – 0x7f60]	***RFU***
0x7f61	Get TEE Definition
0x7f62	Get Security Domain Definition
0x7f63	Get List of TAs
0x7f64	Get TA Definition
0x7f65	Get TA Definition 1 (**version 1.1 and later**)
[0x7f66 – 0x7f7f]	***RFU***
See Chapter 7	**Proprietary extensions** (vendor specific)

2174

8.3.2 Command Response Payload Encoding

The type `CmdRespPayload` is a Constructed type which describes a response payload of an administrative command.

```
CmdRespPayload ::= [APPLICATION 1] SEQUENCE {
    returnCode   INTEGER,
    response     CHOICE {
                       < choice of any possible command response types defined in this
                       specification from section 8.3.4 to section 8.8. This CHOICE type is
                       extensible >,
                       …      -- this extension marker indicates that types representing new future
                       defined command responses are supported
                 }            OPTIONAL
}
```

With:

- **`returnCode`** – The return code of the command
- **`response`** – One of the command response type values as specified from section 8.3.4 to section 8.8.
    - It SHALL be resent only if the command returns any data.

The general TLV encoding is defined as follows.

Table 8-8: Response Message TLV Encoding

Tag	Length	Value Octets			Presence
`0x61`	L	`CmdRespPayload` value			M
		Tag	**Length**	**Value Octets**	
		`0x02`	L	`returnCode`	M
		`<ResponseTag>` (command-specific)	L	`response`	O

8.3.3 Definition and Encoding of Common Data Types

The following data types are commonly referred to in command and response operations encoding.

8.3.3.1 Attribute Type

The `Attribute` type is a Constructed type that encodes an attribute of an object as defined in [TEE Core API] section 5.3.1.

```
Attribute ::= [APPLICATION 2] SEQUENCE {
     attributID   INTEGER,
     content      CHOICE {
                        reference   OCTET STRING,
                        value       SEQUENCE {
                                          a     INTEGER,
                                          b     INTEGER
                                    }
                  }
     }
```

With:

- **`attributID`** – The attribute identifier value as defined in [TEE Core API] section 6.11
- **`content`** – The attribute value matching the `TEE_Attribute` type definition in [TEE Core API] section 5.3.1. If bit [29] of the attribute identifier is set to `0`, content is a reference value (a buffer of octets); if it is set to 1, content is a value represented by two integers. The value SHALL be represented in a transportable format (e.g. bignums).

The TLV encoding is defined as follows.

Table 8-9: Attribute TLV Encoding

<table>
<tr><th>Tag</th><th>Length</th><th colspan="5">Value Octets</th><th>Presence</th></tr>
<tr><td rowspan="7">0x62</td><td rowspan="7">L</td><td colspan="5">Attribute value</td><td>M</td></tr>
<tr><td><b>Tag</b></td><td><b>Length</b></td><td colspan="3"><b>Value Octets</b></td><td></td></tr>
<tr><td>0x02</td><td>L</td><td colspan="3">attributID</td><td>M</td></tr>
<tr><td rowspan="4">0x04
<i>or</i>
0x30</td><td rowspan="4">L</td><td colspan="3">reference
<i>or</i>
value (a sequence value of)</td><td>M</td></tr>
<tr><td><b>Tag</b></td><td><b>Length</b></td><td><b>Value Octets</b></td><td></td></tr>
<tr><td>0x02</td><td>L</td><td>a</td><td>M (1)</td></tr>
<tr><td>0x02</td><td>L</td><td>b</td><td>M (1)</td></tr>
</table>

(1) Mandatory fields only when the `content` value describes a `value` (rather than a `reference`).

8.3.3.2 UUID Type

The `UUID` type is a Primitive type that encodes a UUID as a 16-octet raw value in a TLV structure.

```
UUID ::= [APPLICATION 3] OCTET STRING
```

The TLV encoding is defined as follows.

Table 8-10: UUID TLV Encoding

Tag	Length	Value Octets	Presence
0x43	0x10	UUID value	M

8.3.3.3 ObjectId Type

The `ObjectId` type is a Primitive type that encodes an object identifier as defined by the TEE Internal Core API ([TEE Core API]) as a value of size 0 to 64 octets in a TLV structure (the notation SIZE(0..64) is adopted from [ASN.1 Encoding]).

```
ObjectId ::= [APPLICATION 4] OCTET STRING (SIZE(0..64))
```

The TLV encoding is defined as follows.

Table 8-11: ObjectId TLV Encoding

Tag	Length	Value Octets	Presence
0x44	L (in range [0..64])	Object ID value	M

8.3.3.4 CryptoOperationParameters Type

The `CryptoOperationParameters` type is a Constructed type that encodes a structure describing the algorithm (an identifier) and possible input parameters used for the calculation of a cryptogram (for encryption, decryption, signature, verification…).The description of parameters maps to the necessary parameters of the cryptographic operations as defined in [TEE Core API].

```
CryptoOperationParameters ::= [APPLICATION 5] SEQUENCE {
    algorithmID        INTEGER,
    operationMode      INTEGER,
    algoParams   CHOICE {
                 iv          OCTET STRING,
                 attrValue   Attribute,
                 aeValue     SEQUENCE {
                       nonce             OCTET STRING,
                       tag         [0]   OCTET STRING      OPTIONAL,
                       tagLen      [1]   INTEGER           OPTIONAL,
                       aad         [2]   OCTET STRING      OPTIONAL,
                       aadLen      [3]   INTEGER           OPTIONAL,
                       payloadLen  [4]   INTEGER           OPTIONAL
                       }
                 }     OPTIONAL
}
```

With:

- **`algorithmID`** – The algorithm identifier as defined in [TEE Core API] Table 6-11. A list of mandatory and optional algorithms depending on the operation context is provided in section A.10.
- **`operationMode`** – The operation mode as defined in [TEE Core API] Table 6-3 (e.g. `TEE_MODE_SIGN`, `TEE_MODE_ENCRYPT`, …)
- **`algoParams`** – Optional extra parameters that can be required by the cryptographic operation using the `algorithmID` field value. Depending on the mode of operation and the algorithm identifier, the possible parameters are described here by a CHOICE type. Refer to the Cipher, MAC, Authenticated Encryption, or Asymmetric algorithm operations defined and explained in [TEE Core API] Chapter 6 (Cryptographic Operations API).

The TLV encoding is defined as follows.

Table 8-12: CryptoOperationParameters TLV Encoding

Tag	Length	Value Octets					Presence
0x65	L	CryptoOperationParameters value					M
		Tag	**Length**	**Value Octets**			
		0x02	0x04	algorithmID			M
		0x02	L	operationMode			M
		0x04 *or* 0x62 *or* 0x30	L	iv *or* attrValue *or* aeValue (a sequence value of)			O
				Tag	**Length**	**Value Octets**	
				0x04	L	nonce	M (1)
				0x80	L	tag	O (2)
				0x81	L	tagLen	O
				0x82	L	aad	O
				0x83	L	aadLen	O
				0x84	L	payloadLen	O

(1) Mandatory field only when algoParams is present and represents Authentication Encryption parameters.

(2) Some fields are Optional because they may depend on the algorithmID (AES CCM or AES GCM) and the mode of operation (encryption/decryption).

8.3.3.5 KeyRefParameters Type

The `KeyRefParameters` type is a Constructed type that encodes a structure describing a key reference and the algorithm parameters used for the calculation of a cryptogram.

```
KeyRefParameters ::= [APPLICATION 6] SEQUENCE {
      keyID              ObjectId,
      keyID2             ObjectId    OPTIONAL,
                         -- for algorithms that require two keys (AES-XTS)
      cryptoParams       CryptoOperationParameters
}
```

With:

- **`keyID`** – The identifier of the key used with the algorithm.
- **`keyID2`** – The identifier of an optional second key used with the algorithm (e.g. AES XTS).
- **`cryptoParams`** – The algorithm parameters to be used when performing a cryptographic operation with the referenced key, as defined in section 8.3.3.4.

The TLV encoding is defined as follows.

Table 8-13: KeyRefParameters TLV Encoding

Tag	Length	Value Octets			Presence
`0x66`	L	`KeyRefParameters value`			M
		Tag	**Length**	**Value Octets**	
		`0x44`	L	`keyID`	M
		`0x44`	L	`keyID2`	O
		`0x65`	L	`cryptoParams`	M

8.3.3.6 StoredDataObject Type

The `StoredDataObject` type is a Constructed type that encodes a structure describing an object passed as a parameter of the Store Data command. This object is persistently stored in the personalization storage space of a TA or SD during the command operation.

The stored object can represent a cryptographic key object, a cryptographic key-pair object, or a data object as defined in [TEE Core API] section 5.1.

```
StoredDataObject ::= [APPLICATION 7] SEQUENCE {
    objId                   ObjectId,
    objType                 INTEGER,
    accessAndShareRights    INTEGER,
    attributes              SEQUENCE OF Attribute    OPTIONAL,
    datastream              OCTET STRING             OPTIONAL,
    metadata                [0] SEQUENCE {
                                        sizeInBits   INTEGER,
                                        usageFlags   INTEGER
                            }     OPTIONAL
}
```

With:

- **`objId`** – The object identifier that uniquely identifies the object in the `TEE_STORAGE_PERSO` storage space. The `ObjectId` type definition and encoding are defined in section 8.3.3.3.
- **`objType`** – The object type as defined in [TEE Core API] Table 6-13.
- **`accessAndShareRights`** – The object access and/or sharing rights. The possible values SHALL comply with the personalization data storage described in section 5.5 depending on whether the stored object is owned by a Trusted Application or a Security Domain. [TEE Core API] Table 5-3 defines the possible values that can be combined in a logical expression using the OR operator.
- **`attributes`** – Only present if the object to be stored is a key or key-pair object; a list of `Attribute` values as defined in section 8.3.3.1.
- **`datastream`** – The data stream associated with the object. A data object has only a data stream.
- **`metadata`** – The metadata associated with the object. Present only if the object is a key or key-pair object. The usage flags are described in [TEE Core API] Table 5-4 and can be combined in a logical expression using the OR operator.

The TLV encoding is defined as follows.

Table 8-14: StoredDataObject TLV Encoding

Tag	Length	Value Octets					Presence
`0x67`	L	`StoredDataObject` value					M
		Tag	**Length**	**Value Octets**			
		`0x44`	L	`objId`			M
		`0x02`	L	`objType`			M
		`0x02`	L	`accessAndShareRights`			M
		`0x30`	L	`attributes` value (a list of `Attribute` values)			C
		`0x04`	L	`datastream`			C
		`0xa0`	L	`metadata`			C
				Tag	**Length**	**Value Octets**	
				`0x02`	L	`sizeInBits`	M (1)
				`0x02`	L	`usageFlags`	M (1)

(1) Mandatory fields only when the `StoredDataObject` describes a key or a key-pair object.

8.3.3.7 UUIDVerificationParams Type

The `UUIDVerificationParams` type is a Constructed type that encodes necessary parameter values to perform a verification of proof of possession of a UUID.

```
UUIDVerificationParams ::= [APPLICATION 8] SEQUENCE {
    protocol          UUID,
    version           INTEGER,
    parameters        CHOICE {
                      uuidV5Params [0] UUIDV5Params, -- for the protocol corresponding to
                      the verification of UUID v5
                      . . . -- for future extensions
                      }
}
```

With:

- **`protocol`** – The UUID identifying the protocol to be used to verify the proof of possession of a UUID.
  - The specification defines a protocol value for the “UUID v5 protocol” only (see section 5.6). The corresponding UUID value of this protocol is `0x6bc2de43501248559c8eeaaf0cb9fde7`.
- **`version`** – The version of the protocol defined by the `protocol` field.
  - The current version value of the protocol “UUID v5” SHALL be `0x01` (version 1).
- **`parameters`** – the Type defining the necessary parameters required by the protocol used to verify the proof of possession of a UUID. This specification defines the following parameters to be used by the protocol defining the proof of possession of a UUID v5:

  ```
  UUIDV5Params ::= SEQUENCE {
       keyType           INTEGER,
       keySize           INTEGER,
       keyAttributes     SEQUENCE OF Attribute,
       signatureParams   CryptoOperationParameters,
       signature         OCTET STRING
  }
  ```

  With:

  - **`keyType`** – A key type as defined in section 5.6.1.
  - **`keySize`** – The key size in bits.
  - **`keyAttributes`** – The key attributes as defined in section 5.6.1 (in particular, the sequential order of these attributes is dependent on the type of key).
  - **`signatureParams`** – The signature parameters as defined in section 8.3.3.4.
  - **`signature`** – A signature as defined in section 5.6.2.

The TLV encoding of `UUIDVerificationParams` for the protocol verifying the proof of possession of a UUID v5 is defined as follows.

Table 8-15: UUIDVerificationParams TLV Encoding for UUID v5 Protocol

<table>
<tr><th>Tag</th><th>Length</th><th colspan="5">Value Octets</th><th>Presence</th></tr>
<tr><td rowspan="11">0x68</td><td rowspan="11">L</td><td colspan="5">UUIDVerificationParams value for UUID v5 protocol</td><td>M</td></tr>
<tr><th>Tag</th><th>Length</th><th colspan="3">Value Octets</th><th></th></tr>
<tr><td>0x43</td><td>0x10</td><td colspan="3">0x6bc2de43501248559c8eeaaf0cb9fde7
(UUID of the UUID v5 protocol)</td><td>M</td></tr>
<tr><td>0x02</td><td>0x01</td><td colspan="3">0x01
(version 1 of the UUID v5 protocol)</td><td>M</td></tr>
<tr><td rowspan="7">0xa0</td><td rowspan="7">L</td><td colspan="3">uuidV5Params value</td><td>M</td></tr>
<tr><th>Tag</th><th>Length</th><th>Value Octets</th><th></th></tr>
<tr><td>0x02</td><td>L</td><td>keyType</td><td>M</td></tr>
<tr><td>0x02</td><td>L</td><td>keySize</td><td>M</td></tr>
<tr><td>0x30</td><td>L</td><td>keyAttributes value
(a list of Attribute values)</td><td>M</td></tr>
<tr><td>0x65</td><td>L</td><td>signatureParams</td><td>M</td></tr>
<tr><td>0x04</td><td>L</td><td>signature</td><td>M</td></tr>
</table>

8.3.3.8 CryptographicData Type

The `CryptographicData` type is a Constructed type that encodes implementation-defined cryptographic data and the associated actions (identified by a cryptographic procedure identifier) that may have to be performed during the *Install SD* operation when passed as parameter of the Install SD command.

This type also describes any possible implementation-defined cryptographic data generated by the TEE device and returned by the Install SD command (see `InstallSDResp` type definition in section 8.5.1.2).

The cryptographic data are optional as well as a parameter or as a returned value of the Install SD command.

To satisfy the multiple use cases where such cryptographic data are passed back and forth between a remote entity and the TEE at SD creation time, this specification specifies an ‘open’ type that permits an implementation to define its own procedures and the corresponding cryptographic data represented as any implementation-defined type.

Version 1.0 of this specification did not mandate support for any specific use case implementation, but elaborated some examples.

However, this version makes those examples normative so that implementations can rely on their existence.

```
CryptographicData ::= [APPLICATION 9] SEQUENCE {
      cryptoProcID       INTEGER, -- identifies the type of crypto data and the procedure to
                         handle them
      cryptoData         OCTET STRING -- an ‘open’ type as mentioned in section 7.1 that
                         represents any implementation-defined type
}
```

With:

- **`cryptoProcID`** – A value that identifies the implementation-defined actions to be applied on the `cryptoData` Octets value content during the *Install SD* operation. See the examples provided in section 8.3.4.
- **`cryptoData`** – Any implementation-defined type value, dependent on the procedure identifier value (i.e. `cryptoProcID`). This specification provides some possible implementations of such procedures and associated cryptographic data in section 8.3.4. Implementations may add more.

The TLV encoding is defined as follows.

Table 8-16: CryptographicData TLV Encoding

Tag	Length	Value Octets			Presence
0x69	L	CryptographicData value			M
		Tag	**Length**	**Value Octets**	
		0x02	L	cryptoProcID	M
		0x04	L	cryptoData The TLV structure (DER encoded) of any implementation-defined type	M

8.3.3.9 Property Type

The Property type is a Constructed type that encodes a property name, type, and value.

```
Property ::= [APPLICATION 10] SEQUENCE {
    name        UTF8String,
    value       CHOICE {
                      boolean     BOOLEAN,
                      integer     INTEGER,
                      string      UTF8String,
                      binary      OCTET STRING,
                      uuid        UUID,
                      identity    SEQUENCE {
                                        loginMethod  INTEGER,
                                        uuid         UUID
                                  }
                }
}
```

With:

- **name** – Any ASCII string encoded as PrintableString according to [ASN.1].
- **value** – The property value of the property denoted by its name value. According to [TEE Core API] section 4.4, it SHALL be encoded as:
  - boolean – A Boolean value
  - integer – A 32-bit unsigned integer value
  - string – A UTF-8 string value
  - binary – A binary block value
  - uuid – A UUID value
  - identity – An identity value which consists of both a login method value (as defined in [TEE Core API]) and a UUID value

The TLV encoding is defined as follows.

Table 8-17: Property TLV Encoding

<table>
<tr><th>Tag</th><th>Length</th><th colspan="5">Value Octets</th><th>Presence</th></tr>
<tr><td rowspan="7">0x6a</td><td rowspan="7">L</td><td colspan="5">Property value</td><td>M</td></tr>
<tr><th>Tag</th><th>Length</th><th colspan="3">Value Octets</th><th></th></tr>
<tr><td>0x13</td><td>L</td><td colspan="3">name</td><td>M</td></tr>
<tr><td rowspan="4">0x01
or
0x02
or
0x0c
or
0x04
or
0x43
or
0x30</td><td rowspan="4">0x01
or
L
or
L
or
L
or
0x10
or
L</td><td colspan="3">boolean
or
integer
or
string (UTF-8)
or
binary
or
uuid
or
identity (a sequence value of)</td><td>M</td></tr>
<tr><th>Tag</th><th>Length</th><th>Value Octets</th><th></th></tr>
<tr><td>0x02</td><td>L</td><td>loginMethod</td><td>M (1)</td></tr>
<tr><td>0x43</td><td>0x10</td><td>uuid</td><td>M (1)</td></tr>
</table>

(1) Mandatory fields only when the Property value is an identity value.

8.3.3.10 SDPrivileges Type

The `SDPrivileges` type is a Constructed type that encodes the Security Domain's list of privileges and the optional root SD property. Each privilege is made of an identifier, an optional property to extend the scope of control of the privilege, and some optional parameters depending on the privilege identifier (reserved for future usage or needed for vendor-specific usage).

```
SDPrivileges ::= [APPLICATION 27] SEQUENCE {
     listOfPrivileges  SEQUENCE OF Privilege,
     isRootSD          BOOLEAN(TRUE)     OPTIONAL
}
```

With:

- **`listOfPrivileges`** – A list of privileges. The list SHALL NOT contain duplicate privilege values (i.e. two different `Privilege` data structures with the same `privilegeID` value). A privilege data structure is defined by:

  ```
  Privilege ::= SEQUENCE {
            privilegeID       INTEGER (1..255),
            privilegeParams   OCTET STRING     OPTIONAL
                              -- an 'open' type as mentioned in section 7.1
  }
  ```

  With:

  - **`privilegeID`** – The privilege identifier value (see Table 8-18).
  - **`privilegeParams`** – Optional privilege parameters, dependent on the identifier value, currently defined in Table 8-18.

- **`isRootSD`** – A property indicating that the Security Domain is a root SD (see section 4.1.3.3).
  - When present, this field SHALL have the Boolean TRUE value.

Table 8-18: Privilege Parameters Definition

privilegeID (in hex)	Privilege Name	privilegeParams Type	privilegeParams Value Octets
0x01 – 0x3f	RFU	RFU	The TLV structure (DER encoded) of the privilegeParams type
0x40	gpd.privilege.teeManagement	SHALL be absent	None These privileges have no privilegeParams value
0x41	gpd.privilege.sdManagement		
0x42	gpd.privilege. sdPersonalization		
0x43	gpd.privilege.taManagement		
0x44	gpd.privilege. taPersonalization		
0x45	gpd.privilege.rsdManagement		
0x46 – 0x7F	RFU	RFU	The TLV structure (DER encoded) of the privilegeParams type
0x80 – 0xFF	Proprietary	Proprietary	

An SDPrivileges value is encoded as follows.

Table 8-19: SDPrivileges TLV Encoding

<table>
<tr><th>Tag</th><th>Length</th><th colspan="7">Value Octets</th><th>Presence</th></tr>
<tr><td rowspan="11">0x7b</td><td rowspan="11">L</td><td colspan="7">SDPrivileges value</td><td>M</td></tr>
<tr><th>Tag</th><th>Len</th><th colspan="5">Value Octets</th><th></th></tr>
<tr><td rowspan="8">0x30</td><td rowspan="8">L</td><td colspan="5">A list of privileges (may be empty)</td><td>M</td></tr>
<tr><th>Tag</th><th>Len</th><th colspan="3">Value Octets</th><th></th></tr>
<tr><td rowspan="4">0x30</td><td rowspan="4">L</td><td colspan="3">Privilege #1</td><td>O</td></tr>
<tr><th>Tag</th><th>Len</th><th>Value Octets</th><th></th></tr>
<tr><td>0x02</td><td>L</td><td>privilegeID</td><td>M</td></tr>
<tr><td>0x04</td><td>L</td><td>privilegeParams
The TLV structure (DER encoded) of the privilegeParams type</td><td>O</td></tr>
<tr><td>…</td><td>…</td><td colspan="3">…</td><td>O</td></tr>
<tr><td>0x30</td><td>L</td><td colspan="3">Privilege #n</td><td>O</td></tr>
<tr><td>0x01</td><td>0x01</td><td colspan="5">0xFF (isRootSD property equals TRUE)</td><td>O</td></tr>
</table>

8.3.3.11 Authority Type

The `Authority` type is a Constructed type that encodes the optional information about a remote entity owning a Security Domain. It typically describes a name and an URL of this Authority.

```
Authority ::= [APPLICATION 28] SEQUENCE {
      name              UTF8String,
      urlInfo           UTF8String   OPTIONAL
}
```

With:

- **`name`** – An Authority name (may be an empty UTF-8 string)
- **`urlInfo`** – An optional URL for this Authority

The TLV encoding is defined as follows.

Table 8-20: Authority TLV Encoding

Tag	Length	Value Octets			Presence
`0x7c`	L	`Authority value`			M
		Tag	**Length**	**Value Octets**	
		`0x0c`	L	`name` (may be empty: L = `0`)	M
		`0x0c`	L	`urlInfo`	O

8.3.4 CryptoProcID

Note: In version 1.0 the values for `CryptoProcID` were given as examples. In version 1.1 these values have been formally defined and ranges defined for other values.

Table 8-20b: CryptoProcID Values

Range	Value
`0x00000001`	`INST_SD_GENERIC_PROC`
`0x00000002`	`INST_SD_GEN_RSA_KEYPAIR_PROC`
`0x00000003`	`INST_SD_GEN_SYMM_KEY_PROC`
`0x00000004 - 0xf0000000`	Reserved for GP use
`0xf0000001 - 0xfffffffe`	Implementation Defined
`0xffffffff`	Invalid ID Reserved for testing

The following methods use the powerful sub-typing notation adopted from [ASN.1]. This illustrates the possibilities offered by the usage of this abstract notation even though the meaning of this notation is very intuitive and in all cases explained via some comments in *italics*.

All the subsequent sections describe the usage of the Cryptographic data parameter that can be passed at SD installation (see Install SD command, section 8.5.1).

8.3.4.1 A Procedure Storing an Authorization Token Verification Key

Procedure Description

- A single RSA public key is provided by the remote Authority owning the newly created SD.
  - This RSA public key can be used by the newly installed Security Domain to verify Authorization Tokens.
  - This RSA public key is a permanent object that will be stored in the personalization storage of the installed Security Domain during this operation.
- No output cryptographic value is returned as the result of the Install SD command.

CryptographicData Type Definition

Referring to the `CryptographicData` type, we define a new `CryptoProcID` value:

Table 8-20c: INST_SD_GENERIC_PROC Defined CryptoProcID Value

`CryptoProcID`	Value
`INST_SD_GENERIC_PROC`	`0x00000001`

Then we associate with this procedure a set of possible `cryptoData` values formally represented by a new implementation-defined type. In the proposed example, this type is defined as a sub-type of the `StoredDataObject` type (defined in section 8.3.3.6). This sub-type is obtained by constraining the set of values of the `StoredDataObject` inner types.

The notation WITH COMPONENTS and INCLUDES is adopted from [ASN.1] to constrain the set of possible `StoredDataObject` values, then describing the new (sub-) type as follows:

```
GenericCryptoData ::= StoredDataObject
          (WITH COMPONENTS {
                 objId,                              -- any key identifier of type ObjectId
                 objType                    (TEE_TYPE_RSA_PUBLIC_KEY),  -- RSA public key
                 accessAndShareRights  (TEE_DATA_FLAG_ACCESS_WRITE),
                 attributes   (INCLUDES SEQUENCE (SIZE(2)) OF Attribute -- two attributes : modulus
                                                                            -- and public exponent
                                 ( WITH COMPONENTS {
                                       attributID    (TEE_ATTR_RSA_MODULUS |
                                                     TEE_ATTR_RSA_PUBLIC_EXPONENT),
                                       content ( WITH COMPONENTS { reference PRESENT} )
                                  })
                 ),
                 datastream         ABSENT,  -- the optional data stream SHALL NOT be present
                 metadata           (WITH COMPONENTS {
                                           sizeInBits          (2048..MAX),  -- constraint on key size >= 2048
                                           usageFlags         (TEE_USAGE_VERIFY)
                                           }
                                  )
         } )


   -- Finally, the CryptographicData type is constrained as follows:


   GenericCryptographicData ::= CryptographicData
        ( WITH COMPONENTS {
          cryptoProcID     (INST_SD_GENERIC_PROC),
          cryptoData        (INCLUDES OCTET STRING (CONTAINING GenericCryptoData))
        })


-- The DER-encoded TLV structure value of the GenericCryptoData type will be assigned to the Value Octets
of the cryptoData field of the CryptographicData type value in the Install SD command.
```

All the `TEE_*` integer constants used in this example are defined by [TEE Core API].

8.3.4.2 A Procedure Generating an RSA Public Key

Procedure Description

- A single RSA public key is provided by the corresponding remote Authority owning the newly created SD.
    - This RSA public key can be used by the newly installed Security Domain to verify Authorization Tokens.
    - This RSA public key is a permanent object that will be stored in the personalization storage of the installed Security Domain during this operation.
- At SD installation time, a new RSA key-pair is generated and stored as a permanent object (whose ID is provided by the Remote entity) in the personalization storage of the installed Security Domain during this operation.
- The generated RSA public key part is returned by the command and signed by the Security Domain performing the operation, using some given signature parameters as a proof of authenticity of the generated keys.

We assume in the following example that the Remote entity is providing the parameter values for the generated key and its signature.

CryptographicData Type Definition

Referring to the `CryptographicData` type, we define a new `CryptoProcID` value:

Table 8-20d: INST_SD_GEN_RSA_KEYPAIR_PROC Defined CryptoProcID Value

CryptoProcID	Value
INST_SD_GEN_RSA_KEYPAIR_PROC	0x00000002

Then we associate with this procedure a set of possible `cryptoData` values formally represented by a new implementation-defined type defined in this example, as follows:

```
RSAGenKeyData ::= SEQUENCE {
        inputRSAPubKey          GenericCryptoData, -- we reuse the type defined for the generic
                                                   -- procedure to describe the possible set of values
                                                   -- of the RSA public key provided by the
                                                   -- Remote entity
        genKeyDesc              SEQUENCE {    -- the inner type describing the key to be generated
                                        keyID           ObjectId,       -- key unique ID
                                        keyType         INTEGER,        -- key type
                                        keyUsage        INTEGER,        -- key usage
                                        keySize         INTEGER         -- key size in bits
                                },
        signatureInfos          KeyRefParameters        -- type defined in section 8.3.3.5 describing
                                                        -- the signature key and parameters
                                                        -- to be used to sign the generated
                                                        -- RSA public key
} (WITH COMPONENTS {
                inputRSAPubKey,  -- any RSA public key as defined by the GenericCryptoData type
                genKeyDesc      (WITH COMPONENTS { -- the constrained values of the generated key
                                        keyID,                          -- ID of the generated key
                                        keyType         (TEE_TYPE_RSA_KEYPAIR), -- RSA key pair
                                        keyUsage        (TEE_USAGE_ENCRYPT),    -- key encryption
                                        keySize         (2048..MAX)             -- minimum size in bits
                                } ),
                signatureInfos  -- any key and parameters as defined by the KeyRefParameters type
  } )

  -- Finally, the  CryptographicData type is constrained as follows:

  RSACryptographicData ::= CryptographicData
      ( WITH COMPONENTS {
        cryptoProcID    (INST_SD_GENERIC_PROC),
        cryptoData      (INCLUDES OCTET STRING (CONTAINING RSAGenKeyData))
      })

-- The DER-encoded TLV structure value of the RSAGenKeyData type will be assigned to the Value Octets of
the cryptoData field of the CryptographicData type value in the Install SD command.
```

All the `TEE_*` integer constants used in this example are defined by [TEE Core API].

This procedure requires that the Install SD command returns cryptographic data. Therefore, we define a new return type as follows:

```
RSAGenProcOutput ::= SEQUENCE {
        genKeyValue             SEQUENCE OF Attributes,
        signature               OCTET STRING
} (WITH COMPONENTS {
        genKeyValue (INCLUDES SEQUENCE (SIZE(2)) OF Attribute -- two mandatory attributes :
                                                                -- modulus and public exponent
                                ( WITH COMPONENTS {
                                    attributID    (TEE_ATTR_RSA_MODULUS |
                                                   TEE_ATTR_RSA_PUBLIC_EXPONENT),
                                    content ( WITH COMPONENTS { reference PRESENT} )
                                })
        ),
        signature -- signature calculated over the genKeyValue
} )
```

All the `TEE_*` integer constants used in this example are defined by [TEE Core API].

So, the returned `CryptographicData` type instantiating the set of possible returned values by the Install SD command looks like:

```
InstallSDResp ::=  CryptoGraphicData
        (WITH COMPONENTS {
                cryptoProcID    (INST_SD_GEN_RSA_KEYPAIR_PROC),
                cryptoData    ( INCLUDES OCTET STRING ( CONTAINING RSAGenProcOutput ) )
PRESENT -- a mandatory value of type RSAGenProcOutput
        } )
```

8.3.4.3 A Procedure Generating a Symmetric Secret Key

Procedure Description

- A single RSA public key is provided by the remote Authority owning the newly created SD.
    - This RSA public key can be used by the newly installed Security Domain to verify Authorization Tokens.
    - This RSA public key is a permanent object that will be stored in the personalization storage of the installed Security Domain during this operation.
- At SD installation time, a symmetric AES key is generated and stored as a permanent object in the personalization storage of the installed Security Domain during this operation (this secret key can be used to provision additional keys in a later stage)
- The generated symmetric key is returned by the command encrypted with a public RSA encryption key provided by the Remote entity. Then the Security Domain performing the operation signs the encrypted key as a proof of its authenticity.

We assume in the following example that the Remote entity is providing the parameter values for the generated key, its encryption and signature.

CryptographicData Type Definition

Referring to the `CryptographicData` type, we define a new `CryptoProcID` value:

Table 8-20e: INST_SD_GEN_SYMM_KEY_PROC Defined CryptoProcID Value

`CryptoProcID`	Value
`INST_SD_GEN_SYMM_KEY_PROC`	`0x00000003`

Then we associate with this procedure a set of possible `cryptoData` values formally represented by a new implementation-defined type defined in this example as follows:

```
SymmetricGenKeyData ::= SEQUENCE {
        inputRSAPubKey          GenericCryptoData,  -- we reuse the type defined for the generic
                                                    -- procedure to describe the RSA public key
                                                    --  provided by the Remote entity
        genKeyDesc              SEQUENCE {    -- the inner type describing the key to be generated
                                        keyID           ObjectId,       -- key unique ID
                                        keyType         INTEGER,        -- key type
                                        keyUsage        INTEGER,        -- key usage
                                        keySize         INTEGER         -- key size in bits
                                },
        encryptionKey           SEQUENCE {    -- information provided by the Remote entity
                                              -- about the key algorithm and key value to be used
                                              -- when encrypting the generated symmetric key
                                        algoID          INTEGER,
                                        keyType         INTEGER,
                                        keySize         INTEGER,
                                        keyAttributes   SEQUENCE OF Attribute
                                },
        signatureInfos          KeyRefParameters        -- information related to the signature key and
                                                        -- parameters to be used to sign the encrypted
                                                        -- symmetric key to be returned
} (WITH COMPONENTS {
                inputRSAPubKey,  -- any RSA public key as defined by the GenericCryptoData type
                genKeyDesc (WITH COMPONENTS { -- the constrained values of the generated key
                                keyID,                                          -- ID of the generated key
                                keyType         (TEE_TYPE_AES),                 -- an AES key
                                keyUsage        (TEE_USAGE_ENCRYPT),            -- its key usage
                                keySize         (256)                           -- its size in bits
                            } ),
                encryptionKey (WITH COMPONENTS { -- the input RSA public key encryption key
                                algoID  (TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA256),
                                keyType         (TEE_TYPE_RSA_PUBLIC_KEY),
                                keySize         (2048),
                                keyAttributes
                                        (INCLUDES SEQUENCE (SIZE(2)) OF Attribute
                                                                -- two mandatory attributes : modulus
                                                        -- and public exponent
                                        ( WITH COMPONENTS {
                                            attributID    (TEE_ATTR_RSA_MODULUS |
                                                           TEE_ATTR_RSA_PUBLIC_EXPONENT),
                                            content ( WITH COMPONENTS { reference PRESENT} )
                                        } )
                                  )
                         }),
                   signatureInfos   -- any key and parameters as defined by the KeyRefParameters type
  } )

  -- Finally, the  CryptographicData type is constrained as follows:


  SymmetricCryptographicData ::= CryptographicData
      ( WITH COMPONENTS {
        cryptoProcID     (INST_SD_GENERIC_PROC),
        cryptoData       (INCLUDES OCTET STRING (CONTAINING SymmetricGenKeyData))
      })

-- The DER-encoded TLV structure value of the SymmetricGenKeyData type will be assigned to the Value Octets of the cryptoData field of the CryptographicData type value in the Install SD command.
```

All the `TEE_*` integer constants used in this example are defined by [TEE Core API].

This procedure requires that the Install SD command is returning cryptographic data. So, we define this new return type as follows:

```
SymmetricGenProcOutput ::= SEQUENCE {
        encryptedKeyValue       OCTET STRING,  -- the encrypted generated key
        signature               OCTET STRING   -- the signature calculated over the encrypted value
}
```

So, the returned `CryptographicData` type instantiating the set of possible returned values by the Install SD command looks like:

```
 InstallSDResp ::=  CryptographicData
        (WITH COMPONENTS {
                cryptoProcID     (INST_SD_GEN_SYMM_KEY_PROC),
                cryptoData       ( INCLUDES OCTET STRING ( CONTAINING SymmetricGenProcOutput ))
 PRESENT -- a mandatory value of type SymmetricGenProcOutput
        } )
```

8.4 Trusted Application Commands

8.4.1 Install TA

The Install TA command performs the *Install Trusted Application* operation as defined in section 6.2.1.

8.4.1.1 Command Parameters

The `InstallTA` type is a Constructed type encoding the Install TA command and its parameters.

```
InstallTA ::= [APPLICATION 65] SEQUENCE {
    ta                       UUID,
    targetSD                 UUID,
    initialState             TALifecycleState,
    applicationFile          OCTET STRING,
    encryptionParams         CHOICE {
                                   param5      KeyRefParameters,
                                   null        NULL
                             } ,
    idVerificationParams     CHOICE {
                                   param6      UUIDVerificationParams,
                                   null        NULL
                             }
}
```

With the following attributes defining the command parameters:

- **`ta`** – The UUID of the TA to be installed
  - o UUID type definition and its encoding are defined in section 8.3.3.2.
- **`targetSD`** – The UUID of the SD the TA must be associated with
- **`initialState`** – The initial life cycle state of the application being installed.
  - o This can be used to install and lock or activate an application in a single operation.
  - o `TALifecycleState` type definition and its encoding are defined in section 9.3.1.
- **`applicationFile`** – The Application file contains the binary code and the application properties.
- **`encryptionParams`** – The encryption parameters used to encrypt the `applicationFile`.
  - o This parameter value SHALL be different from the NULL value if the `applicationFile` is encrypted.
  - o `KeyRefParameters` type definition and its encoding are defined in section 8.3.3.5.
- **`idVerificationParams`** – Parameter for the verification of the UUID (ID parameter) of the TA being installed if the UUID comprises the UUID v5 structure.
  - o If the UUID (ID parameter) does not comprise a UUID v5 structure, this parameter value SHALL be the NULL value (tag = `0x05` and length = `0x00`).
  - o `UUIDVerificationParams` type definition and its encoding are defined in section 8.3.3.7.

The general TLV encoding is defined as follows.

Table 8-21: Install TA Command TLV Encoding

Tag	Length	Value Octets			Presence
0x7f41	L	InstallTA parameters			M
		Tag	**Length**	**Value Octets**	
		0x43	L	ta	M
		0x43	L	targetSD	M
		0x53	L	initialState	M
		0x04	L	applicationFile	M
		0x66 *or* 0x05	L *or* 0x00	param5 *or* none	M
		0x68 *or* 0x05	L *or* 0x00	param6 *or* none	M

The *Application File* internal organization is out of scope of this specification and may depend on the language, compiler, Application Binary Interface (ABI), and underlying hardware.

However, it must encapsulate the following components:

- The Trusted Application binary code including the necessary metadata to be able to link it with embedded libraries
- The Trusted Application properties

8.4.1.2 Response Output

Not available

8.4.1.3 Return Codes

Table 8-22: Install TA Command Return Codes

Return Code	Reason
`TEE_SUCCESS`	Command has been successfully executed.
`TEE_ERROR_ACCESS_DENIED`	The security conditions to perform the operation are not satisfied.
`TEE_ERROR_BAD_PARAMETERS`	Invalid command parameter.
`TEE_ERROR_BAD_FORMAT`	Bad parameter format (duplicate constraint values if the Token is present, wrong `idVerificationParams` value when checking the UUID proof of possession…)
`TEE_ERROR_LIMIT_EXCEEDED`	The operation would take the TEE beyond its implementation limits.
`TEE_ERROR_STORAGE_NOT_AVAILABLE`	The persistent storage is currently not accessible.
`TEE_ERROR_STORAGE_NO_SPACE`	There is not enough space in the persistent storage to complete this operation.
`TEE_ERROR_OUT_OF_MEMORY`	There are not enough resources to perform this operation.
`TEE_ERROR_CORRUPT_OBJECT`	The decryption key object is corrupted.
`TEE_ERROR_BAD_STATE`	The target SD has a wrong life cycle state.
`TEE_ERROR_ITEM_NOT_FOUND`	The decryption key object cannot be found. The target SD cannot be found.

8.4.2 Uninstall TA

The Uninstall TA command performs the *Uninstall Trusted Application* operation as defined in section 6.2.2.

8.4.2.1 Command Parameters

The `UninstallTA` type is a Constructed type encoding the Uninstall TA command and its parameters.

```
UninstallTA ::= [APPLICATION 66] SEQUENCE {
    ta                UUID
}
```

With the following attribute defining the command parameters:

- **`ta`** – The UUID of the TA to be uninstalled
  - o UUID type definition and its encoding are defined in section 8.3.3.2.

The TLV encoding is defined as follows.

Table 8-23: Uninstall TA Command TLV Encoding

Tag	Length	Value Octets			Presence
`0x7f42`	`0x12`	`UninstallTA` parameters			M
		Tag	**Length**	**Value Octets**	
		`0x43`	`0x10`	`ta`	M

8.4.2.2 Response Output

Not available

8.4.2.3 Return Codes

Table 8-24: Uninstall TA Command Return Codes

Return Code	Reason
`TEE_SUCCESS`	Command has been successfully executed.
`TEE_ERROR_ACCESS_DENIED`	The security conditions to perform the operation are not satisfied.
`TEE_ERROR_BAD_PARAMETERS`	Invalid command parameter.
`TEE_ERROR_BAD_FORMAT`	Bad parameter format (duplicate constraint values if the Token is present…)
`TEE_ERROR_OUT_OF_MEMORY`	There are not enough resources to perform this operation.
`TEE_ERROR_ITEM_NOT_FOUND`	The TA to uninstall cannot be found.

8.4.3 Update TA

The Update TA command performs the *Update Trusted Application* operation as defined in section 6.2.3.

8.4.3.1 Command Parameters

The `UpdateTA` type is a Constructed type encoding the Update TA command and its parameters.

```
UpdateTA ::= [APPLICATION 67] SEQUENCE {
    ta                       UUID,
    newState                 TALifecycleState,
    applicationFile          OCTET STRING,
    encryptionParams         CHOICE {
                                   param4      KeyRefParameters,
                                   null        NULL
                             },
    idVerificationParams           CHOICE {
                                   param5      UUIDVerificationParams,
                                   null        NULL
                             }
}
```

With the following attributes defining the command parameters:

- **`ta`** – The UUID of the TA to be updated
  - UUID type definition and its encoding are defined in section 8.3.3.2.
- **`newState`** – The new life cycle state of the application being updated.
  - This can be used to update and lock or “activate” an application in a single operation.
  - `TALifecycleState` type definition and its encoding are defined in section 9.3.1.
- **`applicationFile`** – The Application file contains the binary code and the application properties.
- **`encryptionParams`** – The encryption parameters used to encrypt the `applicationFile`.
  - This parameter value SHALL be different from the NULL value if the `applicationFile` is encrypted.
  - `KeyRefParameters` type definition and its encoding are defined in section 8.3.3.5.
- **`idVerificationParams`** – Parameter for the verification of the UUID (id parameter) of the TA being installed if the UUID comprises the UUID v5 structure.
  - If the UUID (id parameter) does not comprise a UUID v5 structure, this parameter value SHALL be the NULL value (tag = `0x05` and length = `0x00`).
  - `UUIDVerificationParams` type definition and its encoding are defined in section 8.3.3.7.

The general TLV encoding is defined as follows.

Table 8-25: Update TA Command TLV Encoding

Tag	Length	Value Octets			Presence
0x7f43	L	UpdateTA parameters			M
		Tag	**Length**	**Value Octets**	
		0x43	L	ta	M
		0x53	L	newState	M
		0x04	L	applicationFile	M
		0x66 *or* 0x05	L *or* 0x00	param4 *or* none	M
		0x68 *or* 0x05	L *or* 0x00	param5 *or* none	M

The *Application File* internal organization is out of scope of this specification and may depend on the language, compiler, Application Binary Interface (ABI), and underlying hardware.

However, it must encapsulate the following components:

- The Trusted Application binary code including the necessary metadata to be able to link it with embedded libraries
- The Trusted Application properties

8.4.3.2 Response Output

Not available

8.4.3.3 Return Codes

Table 8-26: Update TA Command Return Codes

Return Code	Reason
`TEE_SUCCESS`	Command has been successfully executed.
`TEE_ERROR_ACCESS_DENIED`	The security conditions to perform the operation are not satisfied.
`TEE_ERROR_BAD_PARAMETERS`	Invalid command parameter.
`TEE_ERROR_BAD_FORMAT`	Bad parameter format (duplicate constraint values if the Token is present, wrong `idVerificationParams` value when checking the UUID proof of possession…).
`TEE_ERROR_LIMIT_EXCEEDED`	The operation would take the TEE beyond its implementation limits.
`TEE_ERROR_STORAGE_NOT_AVAILABLE`	The persistent storage is currently not accessible.
`TEE_ERROR_STORAGE_NO_SPACE`	There is not enough space in the persistent storage to complete this operation.
`TEE_ERROR_OUT_OF_MEMORY`	There are not enough resources to perform this operation.
`TEE_ERROR_CORRUPT_OBJECT`	The decryption key object is corrupted.
`TEE_ERROR_BAD_STATE`	The TA to update has a wrong life cycle state.
`TEE_ERROR_ITEM_NOT_FOUND`	The decryption key object cannot be found. The TA to update cannot be found.

8.4.4 Lock TA

The Lock TA command performs the *Lock Trusted Application* operation as defined in section 6.2.4.

8.4.4.1 Command Parameters

The `LockTA` type is a Constructed type that encodes the Lock TA command and its parameters.

```
LockTA ::= [APPLICATION 68] SEQUENCE {
    ta          UUID
}
```

With the following attribute defining the command parameter:

- **`ta`** – The UUID of the TA to be locked
  - UUID type definition and its encoding are defined in section 8.3.3.2.

The TLV encoding is defined as follows.

Table 8-27: Lock TA Command TLV Encoding

Tag	Length	Value Octets			Presence
`0x7f44`	`0x12`	`LockTA parameters`			M
		Tag	**Length**	**Value Octets**	
		`0x43`	`0x10`	`ta`	M

8.4.4.2 Response Output

Not available

8.4.4.3 Return Codes

Table 8-28: Lock TA Command Return Codes

Return Code	Reason
`TEE_SUCCESS`	Command has been successfully executed.
`TEE_ERROR_ACCESS_DENIED`	The security conditions to perform the operation are not satisfied.
`TEE_ERROR_BAD_PARAMETERS`	Invalid command parameter.
`TEE_ERROR_BAD_FORMAT`	Bad parameter format (duplicate constraint values if the Token is present…)
`TEE_ERROR_OUT_OF_MEMORY`	There are not enough resources to perform this operation.
`TEE_ERROR_BAD_STATE`	The TA to lock has a wrong life cycle state.
`TEE_ERROR_ITEM_NOT_FOUND`	The TA to lock cannot be found.

8.4.5 Unlock TA

The Unlock TA command performs the *Unlock Trusted Application* operation as defined in section 6.2.5.

8.4.5.1 Command Parameters

The `UnlockTA` type is a Constructed type that encodes the Unlock TA command and its parameters.

```
UnlockTA ::= [APPLICATION 69] SEQUENCE {
    ta          UUID
}
```

With the following attribute defining the command parameter:

- **`ta`** – The UUID of the TA to be unlocked
  - o UUID type definition and its encoding are defined in section 8.3.3.2.

The TLV encoding is defined as follows.

Table 8-29: Unlock TA Command TLV Encoding

<table>
<tr><th>Tag</th><th>Length</th><th colspan="3">Value Octets</th><th>Presence</th></tr>
<tr><td rowspan="3">0x7f45</td><td rowspan="3">0x12</td><td colspan="3">UnlockTA parameters</td><td>M</td></tr>
<tr><th>Tag</th><th>Length</th><th>Value Octets</th><th></th></tr>
<tr><td>0x43</td><td>0x10</td><td>ta</td><td>M</td></tr>
</table>

8.4.5.2 Response Output

Not available

8.4.5.3 Return Codes

Table 8-30: Unlock TA Command Return Codes

Return Code	Reason
TEE_SUCCESS	Command has been successfully executed.
TEE_ERROR_ACCESS_DENIED	The security conditions to perform the operation are not satisfied.
TEE_ERROR_BAD_PARAMETERS	Invalid command parameter.
TEE_ERROR_BAD_FORMAT	Bad parameter format (duplicate constraint values if the Token is present…)
TEE_ERROR_OUT_OF_MEMORY	There are not enough resources to perform this operation.
TEE_ERROR_BAD_STATE	The TA to unlock has a wrong life cycle state.
TEE_ERROR_ITEM_NOT_FOUND	The TA to unlock cannot be found.

8.4.6 Update TA and Data

New in version 1.1

The Update TA and Data command performs the *Update Trusted Application* operation as defined in section 6.2.3 and the *Store Data* operation as defined in section 6.4.1 as a single atomic operation.

8.4.6.1 Command Parameters

The `UpdateTAandData` type is a Constructed type encoding the Update TA and Data command and its parameters.

```
UpdateTAandData ::= [APPLICATION 70] SEQUENCE {
    ta                       UUID,
    newState                 TALifecycleState,
    applicationFile          OCTET STRING,
    encryptionParams         CHOICE {
                                   param4      KeyRefParameters,
                                   null        NULL
                             },
    idVerificationParams     CHOICE {
                                   param5      UUIDVerificationParams,
                                   null        NULL
                             },
    decryptionParams         CHOICE {
                                   param6      KeyRefParameters,
                                   null        NULL
                             },
    storedDataObject         CHOICE {
                                   cipheredText OCTET STRING,
                                   clearText   StoredDataObject
                             }
}
```

With the following attributes defining the command parameters:

- **`ta`** – The UUID of the TA to be updated.
    - UUID type definition and its encoding are defined in section 8.3.3.2.
- **`newState`** – The new life cycle state of the application being updated.
    - This can be used to update and lock or “activate” an application in a single operation.
    - `TALifecycleState` type definition and its encoding are defined in section 9.3.1.
- **`applicationFile`** – The Application file contains the binary code and the application properties.
- **`encryptionParams`** – The encryption parameters used to encrypt the `applicationFile`.
    - This parameter value SHALL be different from the NULL value if the `applicationFile` is encrypted.
    - `KeyRefParameters` type definition and its encoding are defined in section 8.3.3.5.

- **`idVerificationParams`** – Parameter for the verification of the UUID (id parameter) of the TA being installed if the UUID comprises the UUID v5 structure.
  - If the UUID (id parameter) does not comprise a UUID v5 structure, this parameter value SHALL be the NULL value (tag = `0x05` and length = `0x00`).
  - `UUIDVerificationParams` type definition and its encoding are defined in section 8.3.3.7.
- **`decryptionParams`** – A decryption parameter used to decrypt the `storedDataObject` value octets, if any
  - If not null, then the `storedDataObject` value is encoded in its `cipheredText` version (as an OCTET STRING). The `KeyRefParameters` type definition and encoding are defined in section 8.3.3.5. When a symmetric algorithm is used, the algorithm information of the `decryptionParams` value SHOULD specify an initial vector that was used to encrypt the `storedDataObject` value octets.
- **`storedDataObject`** – Object data (defined by section 8.3.3.6) to be stored in persistent personalization storage of the TA.

The general TLV encoding is defined as follows.

Table 8-30b: Update TA and Data Command TLV Encoding

Tag	Length	Value Octets			Presence
`0x7f46`	L	`UpdateTAandData` parameters			M
		Tag	**Length**	**Value Octets**	
		`0x43`	`L`	`ta`	M
		`0x53`	`L`	`newState`	M
		`0x04`	`L`	`applicationFile`	M

		`0x66`
		or
		`0x05`
			`L`
			or
			`0x00`
				`param4`
				or
				none
					M

		`0x68`
		or
		`0x05`
			`L`
			or
			`0x00`
				`param5`
				or
				none
					M

		`0x66`
		or
		`0x05`
			`L`
			or
			`0x00`
				`param6`
				or
				none
					M

		`0x04`
		or
		`0x67`
			`L`
			or
			`L`
				`cipheredText`
				or
				`clearText`
				(a `storedDataObject` value)
					M

The *Application File* internal organization is out of scope of this specification and may depend on the language, compiler, Application Binary Interface (ABI), and underlying hardware.

However, it must encapsulate the following components:

- The Trusted Application binary code including the necessary metadata to be able to link it with embedded libraries
- The Trusted Application properties

8.4.6.2 Response Output

Not available

8.4.6.3 Return Codes

Table 8-30c: Update TA and Data Command Return Codes

Return Code	Reason
`TEE_SUCCESS`	Command has been successfully executed.
`TEE_ERROR_ACCESS_DENIED`	The security conditions to perform the operation are not satisfied.
`TEE_ERROR_BAD_PARAMETERS`	Invalid command parameter.
`TEE_ERROR_BAD_FORMAT`	Bad parameter format (duplicate constraint values if the Token is present, wrong `idVerificationParams` value when checking the UUID proof of possession…).
`TEE_ERROR_LIMIT_EXCEEDED`	The operation would take the TEE beyond its implementation limits.
`TEE_ERROR_STORAGE_NOT_AVAILABLE`	The persistent storage is currently not accessible.
`TEE_ERROR_STORAGE_NO_SPACE`	There is not enough space in the persistent storage to complete this operation.
`TEE_ERROR_OUT_OF_MEMORY`	There are not enough resources to perform this operation.
`TEE_ERROR_CORRUPT_OBJECT`	The decryption key object is corrupted.
`TEE_ERROR_BAD_STATE`	The TA to update has a wrong life cycle state.
`TEE_ERROR_ITEM_NOT_FOUND`	The decryption key object cannot be found. The TA to update cannot be found.

8.5 Security Domain Commands

8.5.1 Install SD

The Install SD command performs the *Install Security Domain* operation as defined in section 6.3.1.

8.5.1.1 Command Parameters

The `InstallSD` type is a Constructed type that encodes the Install SD command and its parameters.

```
InstallSD ::= [APPLICATION 74] SEQUENCE {
    sd                       UUID,
    targetSD                 UUID,
    initialState             SDLifecycleState,
    privileges               SDPrivileges,
    authority                CHOICE {
                                   param5      Authority,
                                   null        NULL
                             },
    cryptographicData        CHOICE {
                                   param6      CryptographicData,
                                   null        NULL
                             },
    idVerificationParams     CHOICE {
                                   param7      UUIDVerificationParams,
                                   null        NULL
                             }
}
```

Where:

- **`sd`** – The UUID of the SD to be installed
  - UUID type definition and its encoding are defined in section 8.3.3.2.
- **`targetSD`** – The UUID of the SD the newly installed SD must be associated with
- **`initialState`** – The initial life cycle state of the Security Domain being installed
  - This can be used to install and restrict or activate a Security Domain in a single operation, as well as to install and block a Security Domain in a single operation.
  - `SDLifecycleState` type definition and encoding are defined in section 9.2.1.
- **`privileges`** – The privileges of the newly created Security Domain as defined in section 8.3.3.10.
- **`authority`** – Details (name and/or URL) of the Authority that manages this Security Domain
  - This parameter value SHALL be either a UTF8String value or a NULL value (tag = `0x05` and length = `0x00`).
- **`cryptographicData`** – Cryptographic data that is optionally provided by the remote Authority
  - This parameter value SHALL be either a `CryptographicData` value or a NULL value (tag = `0x05` and length = `0x00`). See section 8.3.3.8 for more explanations.

- **`idVerificationParams`** – Parameter for the verification of the UUID (id parameter) of the SD being installed if the UUID comprises the UUID v5 structure.
    - If the UUID (id parameter) does not comprise a UUID v5 structure, this parameter value SHALL be the NULL value (tag = `0x05` and length = `0x00`).
    - `UUIDVerificationParams` type definition and encoding are defined in section 8.3.1.

The general TLV encoding is defined as follows.

Table 8-31: Install SD Command TLV Encoding

Tag	Length	Value Octets			Presence
`0x7f4a`	L	`InstallSD` parameters			M
		Tag	**Length**	**Value Octets**	
		`0x43`	`0x10`	`sd`	M
		`0x43`	`0x10`	`targetSD`	M
		`0x51`	L	`initialState`	M
		`0x7b`	L	`privileges`	M

		`0x7c`
		or
		`0x05`
			L
			or
			`0x00`
				`param5`
				or
				none
					M

		`0x69`
		or
		`0x05`
			L
			or
			`0x00`
				`param6`
				or
				none
					M

		`0x68`
		or
		`0x05`
			L
			or
			`0x00`
				`param7`
				or
				none
					M

8.5.1.2 Response Output

This operation may *optionally* return cryptographic material data. In such a case, the returned structure SHALL be defined by the following `InstallSDResp` Constructed type:

```
  InstallSDResp ::= CryptographicData
```

The general TLV encoding is defined as follows.

Table 8-32: Install SD Response TLV Encoding

Tag	Length	Value Octets			Presence
`0x6b`	L	`InstallSDResp` parameters			M
		Tag	**Length**	**Value Octets**	
		`0x69`	L	`CryptographicData`	O

8.5.1.3 Return Codes

Table 8-33: Install SD Command Return Codes

Return Code	Reason
`TEE_SUCCESS`	Command has been successfully executed.
`TEE_ERROR_ACCESS_DENIED`	The security conditions to perform the operation are not satisfied.
`TEE_ERROR_BAD_PARAMETERS`	Invalid command parameter.
`TEE_ERROR_BAD_FORMAT`	Bad parameter format (duplicate constraint values if the Token is present, wrong `idVerificationParams` value when checking the UUID proof of possession…).
`TEE_ERROR_LIMIT_EXCEEDED`	The operation would take the TEE beyond its implementation limits.
`TEE_ERROR_STORAGE_NOT_AVAILABLE`	The persistent storage is currently not accessible.
`TEE_ERROR_STORAGE_NO_SPACE`	There is not enough space in the persistent storage to complete this operation.
`TEE_ERROR_OUT_OF_MEMORY`	There are not enough resources to perform this operation.
`TEE_ERROR_ITEM_NOT_FOUND`	The Target SD cannot be found.
`TEE_ERROR_NOT_SUPPORTED`	The TEE implementation does not support the requested type of `CryptographicData` parameter value.

8.5.2 Uninstall SD

The Uninstall SD command performs the *Uninstall Security Domain* operation as defined in section 6.3.2.

8.5.2.1 Command Parameters

The `UninstallSD` type is a Constructed type that encodes the Uninstall SD command and its parameters.

```
UninstallSD ::= [APPLICATION 75] SEQUENCE {
      sd                UUID,
      recursive         BOOLEAN
}
```

With the following attributes defining the command parameters:

- **`sd`** – The UUID of the SD to be uninstalled
  - o UUID type definition and its encoding are defined in section 8.3.3.2.
- **`recursive`** – A Boolean value indicating:
  - o When TRUE, that any SD directly or indirectly associated with `sd` SHALL also be removed under the following conditions:
    - ▪ `sd` is the UUID of an existing SD that is an rSD as defined in section 4.1.3.3.
    - ▪ Any SD to remove SHALL be ‘empty’; i.e. there is no TA directly associated with this SD.
  - o When FALSE, that `sd` SHALL have neither child TA nor child SD.

The TLV encoding is defined as follows.

Table 8-34: Uninstall SD Command TLV Encoding

<table>
<tr><th>Tag</th><th>Length</th><th colspan="3">Value Octets</th><th>Presence</th></tr>
<tr><td rowspan="4">0x7f4b</td><td rowspan="4">0x12</td><td colspan="3">UninstallSD parameters</td><td>M</td></tr>
<tr><td><b>Tag</b></td><td><b>Length</b></td><td><b>Value Octets</b></td><td></td></tr>
<tr><td>0x43</td><td>0x10</td><td>sd</td><td>M</td></tr>
<tr><td>0x01</td><td>0x01</td><td>recursive
0x00 (FALSE value)
<i>or</i>
0xFF (TRUE value)</td><td>O</td></tr>
</table>

8.5.2.2 Response Output

Not available

8.5.2.3 Return Codes

Table 8-35: Uninstall SD Command Return Codes

Return Code	Reason
`TEE_SUCCESS`	Command has been successfully executed.
`TEE_ERROR_ACCESS_DENIED`	The security conditions to perform the operation are not satisfied.
`TEE_ERROR_BAD_PARAMETERS`	Invalid command parameter.
`TEE_ERROR_BAD_FORMAT`	Bad parameter format (duplicate constraint values if the Token is present …).
`TEE_ERROR_OUT_OF_MEMORY`	There are not enough resources to perform this operation.
`TEE_ERROR_ITEM_NOT_FOUND`	The SD to uninstall cannot be found.

8.5.3 Block SD

The Block SD command performs the *Block Security Domain* operation as defined in section 6.3.3.

8.5.3.1 Command Parameters

The `BlockSD` type is a Constructed type that encodes the Block SD command and its parameters.

```
BlockSD ::= [APPLICATION 77] SEQUENCE {
    sd                UUID,
    lockFlag          BOOLEAN
}
```

With the following attributes defining the command parameters:

- **`sd`** – The UUID of the SD to be blocked
    - o UUID type definition and its encoding are defined in section 8.3.3.2.
- **`lockFlag`** – This flag is reserved for future use.

The TLV encoding is defined as follows.

Table 8-36: Block SD Command TLV Encoding

Tag	Length	Value Octets			Presence
`0x7f4d`	`0x15`	`BlockSD` parameters			M
		Tag	**Length**	**Value Octets**	
		`0x43`	`0x10`	`sd`	M
		`0x01`	`0x01`	`lockFlag`	M

Backward Compatibility

Version 1.0 provided the `lockflag` but did not describe its effect.

Descriptions of the state transition elsewhere in this document stated all Trusted Applications in the Security Domain will be locked in all cases.

Therefore version 1.1 redefines this flag as RFU.

8.5.3.2 Response Output

Not available

8.5.3.3 Return Codes

Table 8-37: Block SD Command Return Codes

Return Code	Reason
`TEE_SUCCESS`	Command has been successfully executed.
`TEE_ERROR_ACCESS_DENIED`	The security conditions to perform the operation are not satisfied.
`TEE_ERROR_BAD_PARAMETERS`	Invalid command parameter.
`TEE_ERROR_BAD_FORMAT`	Bad parameter format (duplicate constraint values if the Token is present, etc.).
`TEE_ERROR_OUT_OF_MEMORY`	There are not enough resources to perform this operation.
`TEE_ERROR_ITEM_NOT_FOUND`	The SD to block cannot be found.

8.5.4 Unblock SD

The Unblock SD command performs the *Unblock Security Domain* operation as defined in section 6.3.4.

8.5.4.1 Command Parameter

The `UnblockSD` type is a Constructed type that encodes the Unblock SD command and its parameters.

```
UnblockSD ::= [APPLICATION 78] SEQUENCE {
    sd          UUID
}
```

With the following attribute defining the command parameter:

- **`sd`** – The UUID of the SD to be unblocked
    - o UUID type definition and its encoding are defined in section 8.3.3.2.

The TLV encoding is defined as follows.

Table 8-38: Unblock SD Command TLV Encoding

Tag	Length	Value Octets			Presence
`0x7f4e`	`0x12`	`UnblockSD` parameters			M
		Tag	**Length**	**Value Octets**	
		`0x43`	`0x10`	`sd`	M

8.5.4.2 Response Output

Not available

8.5.4.3 Return Codes

Table 8-39: Unblock SD Command Return Codes

Return Code	Reason
`TEE_SUCCESS`	Command has been successfully executed.
`TEE_ERROR_ACCESS_DENIED`	The security conditions to perform the operation are not satisfied.
`TEE_ERROR_BAD_PARAMETERS`	Invalid command parameter.
`TEE_ERROR_BAD_FORMAT`	Bad parameter format (duplicate constraint values if the Token is present …).
`TEE_ERROR_OUT_OF_MEMORY`	There are not enough resources to perform this operation.
`TEE_ERROR_ITEM_NOT_FOUND`	The SD to unblock cannot be found.

8.5.5 Restrict SD

The Restrict SD command performs the *Restrict Security Domain* operation as defined in section 6.3.5.

8.5.5.1 Command Parameters

The `RestrictSD` type is a Constructed type that encodes the Restrict SD command and its parameters.

```
RestrictSD ::= [APPLICATION 79] SEQUENCE {
    sd          UUID
}
```

With the following attribute defining the command parameter:

- **`sd`** – The UUID of the SD to be restricted
  - o UUID type definition and its encoding are defined in section 8.3.3.2.

The TLV encoding is defined as follows.

Table 8-40: Restrict SD Command TLV Encoding

Tag	Length	Value Octets			Presence
`0x7f4f`	`0x12`	`RestrictSD` parameters			M
		Tag	**Length**	**Value Octets**	
		`0x43`	`0x10`	`sd`	M

8.5.5.2 Response Output

Not available

8.5.5.3 Return Codes

Table 8-41: Restrict SD Command Return Codes

Return Code	Reason
`TEE_SUCCESS`	Command has been successfully executed.
`TEE_ERROR_ACCESS_DENIED`	The security conditions to perform the operation are not satisfied.
`TEE_ERROR_BAD_PARAMETERS`	Invalid command parameter.
`TEE_ERROR_BAD_FORMAT`	Bad parameter format (duplicate constraint values if the Token is present …).
`TEE_ERROR_OUT_OF_MEMORY`	There are not enough resources to perform this operation.
`TEE_ERROR_BAD_STATE`	The SD to restrict has a wrong life cycle state.
`TEE_ERROR_ITEM_NOT_FOUND`	The SD to restrict cannot be found.

8.5.6 Unrestrict SD

The Unrestrict SD command performs the *Unrestrict Security Domain* operation as defined in section 6.3.6.

8.5.6.1 Command Parameters

The `UnrestrictSD` type is a Constructed type that encodes the Unrestrict SD command and its parameters.

```
UnrestrictSD ::= [APPLICATION 80] SEQUENCE {
    sd                UUID
}
```

With the following attribute defining the command parameter:

- **`sd`** – The UUID of the SD to be restricted
  - o UUID type definition and its encoding are defined in section 8.3.3.2.

The TLV encoding is defined as follows.

Table 8-42: Unrestrict SD Command TLV Encoding

Tag	Length	Value Octets			Presence
`0x7f50`	`0x12`	`UnrestrictSD` parameters			M
		Tag	**Length**	**Value Octets**	
		`0x43`	`0x10`	`sd`	M

8.5.6.2 Response Output

Not available

8.5.6.3 Return Codes

Table 8-43: Unrestrict SD Command Return Codes

Return Code	Reason
`TEE_SUCCESS`	Command has been successfully executed.
`TEE_ERROR_ACCESS_DENIED`	The security conditions to perform the operation are not satisfied.
`TEE_ERROR_BAD_PARAMETERS`	Invalid command parameter.
`TEE_ERROR_BAD_FORMAT`	Bad parameter format (duplicate constraint values if the Token is present …).
`TEE_ERROR_OUT_OF_MEMORY`	There are not enough resources to perform this operation.
`TEE_ERROR_BAD_STATE`	The SD to unrestrict has a wrong life cycle state.
`TEE_ERROR_ITEM_NOT_FOUND`	The SD to unrestrict cannot be found.

8.6 Commands Common to SD and TA

8.6.1 Store Data

The Store Data command performs the *Store Data* operation as defined in section 6.4.1.

8.6.1.1 Command Parameters

The `StoreData` type is a Constructed type that encodes the Store Data command and its parameters.

```
StoreData ::= [APPLICATION 85] SEQUENCE {
    taORsd              UUID,
    decryptionParams    CHOICE {
                              param2      KeyRefParameters,
                              null        NULL
                        },
    storedDataObject    CHOICE {
                              cipheredText      OCTET STRING,
                              clearText         StoredDataObject
                        }
}
```

With the following attributes defining the command parameters:

- **`taORsd`** – The UUID of the Trusted Application or Security Domain the data must be stored in (UUID type definition and its encoding are defined in section 8.3.3.2)
- **`decryptionParams`** – A decryption parameter used to decrypt the `storedDataObject` value octets, if any
  - If not null, then the `storedDataObject` value is encoded in its `cipheredText` version (as an OCTET STRING). The `KeyRefParameters` type definition and encoding are defined in section 8.3.3.5. When a symmetric algorithm is used, the algorithm information of the `decryptionParams` value SHOULD specify an initial vector that was used to encrypt the `storedDataObject` value octets.
- **`storedDataObject`** – Object data (defined by section 8.3.3.6) to be stored in persistent personalization storage of the TA or SD

The TLV encoding is defined as follows.

Table 8-44: Store Data Command TLV Encoding

<table>
<tr><th>Tag</th><th>Length</th><th colspan="3">Value Octets</th><th>Presence</th></tr>
<tr><td rowspan="5">0x7f55</td><td rowspan="5">L</td><td colspan="3">StoreData parameters</td><td>M</td></tr>
<tr><td><b>Tag</b></td><td><b>Length</b></td><td><b>Value Octets</b></td><td></td></tr>
<tr><td>0x43</td><td>0x10</td><td>taORsd</td><td>M</td></tr>
<tr><td>0x66
<i>or</i>
0x05</td><td>L
<i>or</i>
0x00</td><td>param2
<i>or</i>
none</td><td>M</td></tr>
<tr><td>0x04
<i>or</i>
0x67</td><td>L
<i>or</i>
L</td><td>cipheredText
<i>or</i>
clearText
(a storedDataObject value)</td><td>M</td></tr>
</table>

8.6.1.2 Response Output

Not available

8.6.1.3 Return Codes

Table 8-45: Store Data Command Return Codes

Return Code	Reason
TEE_SUCCESS	Command has been successfully executed.
TEE_ERROR_ACCESS_DENIED	The security conditions to perform the operation are not satisfied.
TEE_ERROR_BAD_PARAMETERS	Invalid command parameter.
TEE_ERROR_BAD_FORMAT	Bad parameter format (duplicate constraint values if the Token is present, object’s attribute with a wrong format …).
TEE_ERROR_LIMIT_EXCEEDED	The operation would take the TEE beyond its implementation limits.
TEE_ERROR_OUT_OF_MEMORY	There are not enough resources to perform this operation.
TEE_ERROR_STORAGE_NOT_AVAILABLE	The persistent storage where the object is stored or retrieved (for replacement) is currently inaccessible.
TEE_ERROR_CORRUPT_OBJECT	The object to be created or replaced is corrupt.
TEE_ERROR_NOT_SUPPORTED	The TEE implementation does not support the type or the length of an object’s attribute
TEE_ERROR_BAD_STATE	The TA or SD for which this operation is performed has a wrong life cycle state.
TEE_ERROR_ITEM_NOT_FOUND	The TA or SD for which this operation is performed cannot be found.

8.6.2 Delete Data

The Delete Data command performs the *Delete Data* operation as defined in section 6.4.2.

8.6.2.1 Command Parameters

The `DeleteData` type is a Constructed type that encodes the Delete Data command and its parameters.

```
DeleteData ::= [APPLICATION 86] SEQUENCE {
    taORsd      UUID,
    objId       ObjectId
}
```

With the following attributes defining the command parameters:

- **`taORsd`** – The UUID of the Trusted Application or Security Domain the data must be removed from (UUID type definition and its encoding are defined in section 8.3.3.2.)
- **`objId`** – The object identifier to be retrieved for removal

The TLV encoding is defined as follows.

Table 8-46: Delete Data Command TLV Encoding

Tag	Length	Value Octets			Presence
`0x7f56`	L	`DeleteData` parameters			M
		Tag	**Length**	**Value Octets**	
		`0x43`	`0x10`	`taORsd`	M
		`0x44`	L	`objId`	M

8.6.2.2 Response Output

Not available.

8.6.2.3 Return Codes

Table 8-47: Delete Data Command Return Codes

Return Code	Reason
`TEE_SUCCESS`	Command has been successfully executed.
`TEE_ERROR_ACCESS_DENIED`	The security conditions to perform the operation are not satisfied.
`TEE_ERROR_BAD_PARAMETERS`	Invalid command parameter.
`TEE_ERROR_BAD_FORMAT`	Bad parameter format (duplicate constraint values if the Token is present …).
`TEE_ERROR_OUT_OF_MEMORY`	There are not enough resources to perform this operation.
`TEE_ERROR_STORAGE_NOT_AVAILABLE`	The persistent storage where the object is stored is currently inaccessible.
`TEE_ERROR_BAD_STATE`	The TA or SD for which this operation is performed has a wrong life cycle state.
`TEE_ERROR_ITEM_NOT_FOUND`	The TA or SD for which this operation is performed cannot be found.

8.6.3 List Objects

The List Objects command performs the *List Objects* operation as defined in section 6.4.3.

8.6.3.1 Command Parameters

The `ListObjects` type is a Constructed type that encodes the List Objects command and its parameters.

```
ListObjects ::= [APPLICATION 87] SEQUENCE {
    taORsd            UUID
}
```

With the following attribute defining the command parameter:

- **`taORsd`** – The UUID of the Trusted Application or Security Domain to retrieve the personalized objects for (UUID type definition and its encoding are defined in section 8.3.3.2)

The TLV encoding is defined as follows.

Table 8-48: List Objects Command TLV Encoding

Tag	Length	Value Octets			Presence
`0x7f57`	L	`ListObjects` parameters			M
		Tag	**Length**	**Value Octets**	
		`0x43`	`0x10`	`taORsd`	M

8.6.3.2 Response Output

The `ListObjectsResp` type is a Constructed type that encodes the data returned by the List Objects command.

```
ListObjectsResp ::= [APPLICATION 25] SEQUENCE OF ObjectId
```

The general TLV encoding is defined as follows.

Table 8-49: List Objects Response TLV Encoding

Tag	Length	Value Octets			Presence
`0x79`	L [1]	`ListObjectsResp` parameters			M
		Tag	**Length**	**Value Octets**	
		`0x44`	L	`ObjectId #1`	O
		…	…	…	O
		`0x44`	L	`ObjectId #n`	O

[1] An empty list (L = `0x00`) is returned if no object was stored.

3010 8.6.3.3 Return Codes

3011 **Table 8-50: List Objects Command Return Codes**

Return Code	Reason
TEE_SUCCESS	Command has been successfully executed.
TEE_ERROR_ACCESS_DENIED	The security conditions to perform the operation are not satisfied.
TEE_ERROR_BAD_PARAMETERS	Invalid command parameter.
TEE_ERROR_BAD_FORMAT	Bad parameter format (duplicate constraint values if the Token is present…).
TEE_ERROR_LIMIT_EXCEEDED	The operation would take the TEE beyond its implementation limits.
TEE_ERROR_OUT_OF_MEMORY	There are not enough resources to perform this operation.
TEE_ERROR_STORAGE_NOT_AVAILABLE	The persistent storage where the objects are stored is currently inaccessible.
TEE_ERROR_CORRUPT_OBJECT	Objects to be retrieved seem corrupt.
TEE_ERROR_BAD_STATE	The TA or SD for which this operation is performed has a wrong life cycle state.
TEE_ERROR_ITEM_NOT_FOUND	The TA or SD for which this operation is performed cannot be found.

8.6.4 Fetch Object

New in version 1.1

The Fetch Object command performs the *Fetch Object* operation as defined in section 6.4.4.

8.6.4.1 Command Parameters

The `FetchObject` type is a Constructed type that encodes the Fetch Object command and its parameters.

```
FetchObject ::= [APPLICATION 88] SEQUENCE {
    sd                  UUID,
    ObjectId            Object identifier,
    ObjectId            Object identifier,
    algorithmID         INTEGER
}
```

With the following attributes defining the command parameter:

- **`sd`** – The UUID of the Security Domain to retrieve the personalized object for (UUID type definition and its encoding are defined in section 8.3.3.2)
- **`ObjectId`** – The Object identifier of a public key or asymmetric key pair stored in private storage of the SD.
- **`ObjectId`** – The Object identifier of an asymmetric private key or key pair with which to sign the returned object or a symmetric key with which to create a Message Authentication Code of the object.
- **`algorithmID`** – The algorithm identifier (as defined in [TEE Core API] Table 6-11) identifying the signature or MAC algorithm to use. This must be present if there is an object identifier for the signing key.

The TLV encoding is defined as follows.

Table 8-50b: Fetch Object Command TLV Encoding

Tag	Length	Value Octets			Presence
`0x7f58`	L	`FetchObject` parameters			M
		Tag	**Length**	**Value Octets**	
		`0x43`	`0x10`	`sd`	M
		`0x44`	L	`ObjectId`	M
		`0x44`	L	`ObjectId`	O
		`0x02`	`0x04`	`algorithmID`	O

8.6.4.2 Response Output

The `FetchObjectResp` type is a Constructed type that encodes the data returned by the Fetch Object command.

```
FetchObjectResp ::= [APPLICATION 103] SEQUENCE OF CryptographicData
```

The general TLV encoding is defined as follows.

Table 8-50c: Fetch Object Response TLV Encoding

Tag	Length	Value Octets			Presence
0x7f67	L (1)	FetchObjectResp parameters			M
		Tag	**Length**	**Value Octets**	
		0x69	L	CryptographicData #1	M
		0x69	L	CryptographicData #2	O
		0x69	L	CryptographicData #3	O

(1) An empty list (L = `0x00`) is returned if no object can be returned.

8.6.4.3 Return Codes

Table 8-50d: Fetch Object Command Return Codes

Return Code	Reason
TEE_SUCCESS	Command has been successfully executed.
TEE_ERROR_ACCESS_DENIED	The security conditions to perform the operation are not satisfied.
TEE_ERROR_BAD_PARAMETERS	Invalid command parameter.
TEE_ERROR_BAD_FORMAT	Bad parameter format (duplicate constraint values if the Token is present…).
TEE_ERROR_OUT_OF_MEMORY	There are not enough resources to perform this operation.
TEE_ERROR_STORAGE_NOT_AVAILABLE	The persistent storage where the objects are stored is currently inaccessible.
TEE_ERROR_CORRUPT_OBJECT	Objects to be retrieved seem corrupt.
TEE_ERROR_BAD_STATE	The TA or SD for which this operation is performed has a wrong life cycle state.
TEE_ERROR_ITEM_NOT_FOUND	The TA or SD for which this operation is performed cannot be found.

8.7 TEE Commands

8.7.1 Lock TEE

The Lock TEE command performs the *Lock TEE* operation as defined in section 6.5.1.

8.7.1.1 Command Parameters

The `LockTEE` type is a Constructed type that encodes the Lock TEE command and its parameters.

```
  LockTEE ::= [APPLICATION 90] SEQUENCE {}
```

The type has a tagged empty structure; that is, this operation has no parameters.

The TLV encoding is defined as follows.

Table 8-51: Lock TEE Command TLV Encoding

Tag	Length	Value Octets	Presence
0x7f5a	0x00		M

8.7.1.2 Response Output

Not available

8.7.1.3 Return Codes

Table 8-52: Lock TEE Command Return Codes

Return Code	Reason
TEE_SUCCESS	Command has been successfully executed.
TEE_ERROR_ACCESS_DENIED	The security conditions to perform the operation are not satisfied.
TEE_ERROR_BAD_PARAMETERS	Invalid command parameter.
TEE_ERROR_BAD_FORMAT	Bad parameter format (duplicate constraint values if the Token is present…).
TEE_ERROR_OUT_OF_MEMORY	There are not enough resources to perform this operation.
TEE_ERROR_BAD_STATE	The TEE has a wrong life cycle state.

8.7.2 Unlock TEE

The Unlock TEE command performs the *Unlock TEE* operation as defined in section 6.5.2.

8.7.2.1 Command Parameters

The `UnlockTEE` type is a Constructed type that encodes the Unlock TEE command and its parameters.

```
UnlockTEE ::= [APPLICATION 91] SEQUENCE {}
```

The type has a tagged empty structure; that is, this operation has no parameters.

The TLV encoding is defined as follows.

Table 8-53: Unlock TEE Command TLV Encoding

Tag	Length	Value Octets	Presence
`0x7f5b`	`0x00`		M

8.7.2.2 Response Output

Not available

8.7.2.3 Return Codes

Table 8-54: Unlock TEE Command Return Codes

Return Code	Reason
`TEE_SUCCESS`	Command has been successfully executed.
`TEE_ERROR_ACCESS_DENIED`	The security conditions to perform the operation are not satisfied.
`TEE_ERROR_BAD_PARAMETERS`	Invalid command parameter.
`TEE_ERROR_BAD_FORMAT`	Bad parameter format (duplicate constraint values if the Token is present…).
`TEE_ERROR_OUT_OF_MEMORY`	There are not enough resources to perform this operation.
`TEE_ERROR_BAD_STATE`	The TEE has a wrong life cycle state.

8.7.3 Store TEE Property

The Store TEE Property command performs the *Store TEE Property* operation as defined in section 6.5.3.

8.7.3.1 Command Parameters

The `StoreTEEProperty` type is a Constructed type that encodes the Store TEE Property command and its parameters.

```
StoreTEEProperty ::= [APPLICATION 92] SEQUENCE {
    property    Property
}
```

With the following attribute defining the command parameter:

- **`property`** – The data used to initialize the modifiable TEE Property.

The TLV encoding is defined as follows.

Table 8-55: Store TEE Property Command TLV Encoding

Tag	Length	Value Octets			Presence
`0x7f5c`	L	`StoreTEEProperty` parameters			M
		Tag	**Length**	**Value Octets**	
		`0x6c`	L	`property`	M

8.7.3.2 Response Output

Not available

8.7.3.3 Return Codes

Table 8-56: Store TEE Properties Command Return Codes

Return Code	Reason
`TEE_SUCCESS`	Command has been successfully executed.
`TEE_ERROR_ACCESS_DENIED`	The security conditions to perform the operation are not satisfied.
`TEE_ERROR_BAD_PARAMETERS`	Invalid command parameter.
`TEE_ERROR_BAD_FORMAT`	Bad parameter format (duplicate constraint values if the Token is present, the property to be stored has a wrong format…).
`TEE_ERROR_LIMIT_EXCEEDED`	The operation would take the TEE beyond its implementation limits.
`TEE_ERROR_OUT_OF_MEMORY`	There are not enough resources to perform this operation.
`TEE_ERROR_EXCESS_DATA`	Unexpected oversized property value.

8.7.4 Factory Reset

The Factory Reset command performs the *Factory Reset* operation as defined in section 6.5.4.

This specification defines the following modifiable TEE property for the *Factory Reset* operation:

Table 8-57: TEE Property for *Factory Reset* Operation

Property Name	Type	Meaning
`gpd.tee.tmf.resetpreserved.entities`	Binary (Base64 encoded)	A list of concatenated UUIDs of entities to be preserved during a *Factory Reset* operation on a TEE.

8.7.4.1 Command Parameters

The `FactoryReset` type is a Constructed type that encodes the Factory Reset command and its parameters.

```
FactoryReset ::= [APPLICATION 93] SEQUENCE {}
```

The type has a tagged empty structure; that is, this operation has no parameters.

The TLV encoding is defined as follows.

Table 8-58: Factory Reset Command TLV Encoding

Tag	Length	Value Octets	Presence
`0x7f5d`	`0x00`		M

8.7.4.2 Response Output

None applicable.

8.7.4.3 Return Codes

Table 8-59: Factory Reset Command Return Codes

Return Code	Reason
`TEE_SUCCESS`	Command has been successfully executed.
`TEE_ERROR_ACCESS_DENIED`	The security conditions to perform the operation are not satisfied.
`TEE_ERROR_BAD_PARAMETERS`	Invalid command parameter.
`TEE_ERROR_BAD_FORMAT`	Bad parameter format (duplicate constraint values if the Token is present…).
`TEE_ERROR_OUT_OF_MEMORY`	There are not enough resources to perform this operation.

8.8 Unprivileged Audit Commands

The subsequent unprivileged audit operations can be submitted to:

- The TMF audit SD (as described in section 4.5), whatever the TEE life cycle state is.
- Any SD that is not in the Blocked life cycle state, provided that the TEE is in the `TEE_SECURED` life cycle state.
- Any SD with the `gpd.privilege.teeManagement` privilege, provided that the SD is not in the Blocked life cycle state.

The TMF audit SD is identified on the TEE by the reserved GlobalPlatform UUID value defined in Table 8-60.

Table 8-60: TMF Audit SD UUID for Audit Operations

2329A4EA-B484-47E4-9B65-262D726B3438

Operations Return Codes

When the TEE cannot read the internal information to be returned by one of any subsequent audit operations, the `TEE_ERROR_INTERNAL` error code is returned.

When the response demands more space than the TEE is able to provide in a single response, then the `TEE_ERROR_LIMIT_EXCEEDED` error code is returned.

8.8.1 Get TEE Definition

The Get TEE Definition command performs the *Get TEE Definition* operation as defined in section 6.6.1.

8.8.1.1 Command Parameters

The `GetTEEDef` type is a Constructed type that encodes the Get TEE Definition command and its parameters.

```
GetTEEDef ::= [APPLICATION 97] SEQUENCE {}
```

The type has a tagged empty structure; that is, this operation has no parameters.

The TLV encoding is defined as follows.

Table 8-61: Get TEE Definition Command TLV Encoding

Tag	Length	Value Octets	Presence
`0x7f61`	`0x00`		M

8.8.1.2 Response Output

The `GetTEEDefResp` type is a Constructed type that encodes the data returned by the Get TEE Definition command.

```
GetTEEDefResp ::= Tee
```

Where the `Tee` type definition and its encoding value are defined in section 9.1.6.

The TLV encoding is defined as follows.

Table 8-62: Get TEE Definition Response TLV Encoding

<table>
<tr><th>Tag</th><th>Length</th><th colspan="3">Value Octets</th><th>Presence</th></tr>
<tr><td rowspan="3">0x7f68</td><td rowspan="3">L [(1)]</td><td colspan="3">GetTEEDefResp parameters</td><td>M</td></tr>
<tr><th>Tag</th><th>Length</th><th>Value Octets</th><th></th></tr>
<tr><td>0x70</td><td>L</td><td>Tee</td><td>M</td></tr>
</table>

8.8.1.3 Return Codes

Table 8-63: Get TEE Definition Command Return Codes

Return Code	Reason
`TEE_SUCCESS`	Command has been successfully executed.
`TEE_ERROR_BAD_FORMAT`	Bad parameter format (duplicate constraint values if the Token is present…).
`TEE_ERROR_INTERNAL`	The *TEE definition* could not be read, or some other unspecified internal error.
`TEE_ERROR_LIMIT_EXCEEDED`	The response demands more space than the TEE is able to provide in a single response.
`TEE_ERROR_OUT_OF_MEMORY`	There are not enough resources to perform this operation.
`TEE_ERROR_SHORT_BUFFER`	The response demands more space than the caller has indicated it is prepared to accept.

8.8.2 Get SD Definition

The Get SD Definition command performs the *Get SD Definition* operation as defined in section 6.6.2.

8.8.2.1 Command Parameters

The `GetSDDef` type is a Constructed type that encodes the Get SD Definition command and its parameters.

```
GetSDDef ::= [APPLICATION 98] SEQUENCE {
          sd           UUID
}
```

With the following attribute defining the command parameter:

- **`sd`** – The UUID of the Security Domain to retrieve the definition of.

The TLV encoding is defined as follows.

Table 8-64: Get SD Definition Command TLV Encoding

Tag	Length	Value Octets			Presence
0x7f62	0x12	GetSDDef parameters			M
		Tag	**Length**	**Value Octets**	
		0x43	0x10	sd	M

8.8.2.2 Response Output

The `GetSDDefResp` type is a Constructed type that encodes the data returned by the Get SD Definition command.

```
GetSDDefResp ::= SecurityDomain
```

Where the `SecurityDomain` type definition and its encoding value are defined in section 9.2.2.

The TLV encoding is defined as follows.

Table 8-65: Get SD Definition Response TLV Encoding

Tag	Length	Value Octets			Presence
0x7f69	L [(1)]	GetSDDefResp parameters			M
		Tag	**Length**	**Value Octets**	
		0x72	L	SecurityDomain	M

8.8.2.3 Return Codes

Table 8-66: Get SD Definition Command Return Codes

Return Code	Reason
`TEE_SUCCESS`	Command has been successfully executed.
`TEE_ERROR_BAD_PARAMETERS`	Invalid command parameter.
`TEE_ERROR_BAD_FORMAT`	Bad parameter format (duplicate constraint values if the Token is present…).
`TEE_ERROR_INTERNAL`	The *SD Definition* could not be read, or some other unspecified internal error.
`TEE_ERROR_LIMIT_EXCEEDED`	The response demands more space than the TEE is able to provide in a single response.
`TEE_ERROR_SHORT_BUFFER`	The response demands more space than the caller has indicated it is prepared to accept.
`TEE_ERROR_OUT_OF_MEMORY`	There are not enough resources to perform this operation.
`TEE_ERROR_ITEM_NOT_FOUND`	The caller requested a definition for an SD which does not exist.

8.8.3 Get List of Trusted Applications

The Get List of Trusted Applications command performs the *Get List of Trusted Applications* operation as specified in section 6.6.3.

8.8.3.1 Command Parameters

The `GetListOfTA` type is a Constructed type that encodes the Get List of Trusted Applications command and its parameters.

```
GetListOfTA ::= [APPLICATION 99] SEQUENCE {
          sd          UUID
}
```

With the following attribute defining the command parameter:

- **`sd`** – The UUID of the SD from which the list of direct TAs is to be retrieved.

The TLV encoding is defined as follows.

Table 8-67: Get List of TAs Command TLV Encoding

Tag	Length	Value Octets			Presence
`0x7f63`	`0x12`	`GetListOfTA` parameters			M
		Tag	**Length**	**Value Octets**	
		`0x43`	`0x10`	`sd`	M

8.8.3.2 Response Output

The `GetListOfTAResp` type is a Constructed type that encodes the data returned by the Get List of Trusted Applications command.

```
GetListOfTAResp ::= [APPLICATION 26] SEQUENCE OF UUID
```

The general TLV encoding is defined as follows.

Table 8-68: Get List of TAs Response TLV Encoding

Tag	Length	Value Octets			Presence
`0x7a`	L [1] (18 * number of UUID)	`GetListOfTAResp` parameters			M
		Tag	**Length**	**Value Octets**	
		`0x43`	`0x10`	UUID #1	O
		…	…	…	O

(1) An empty list (L = `0x00`) may be returned, if there are no Trusted Applications.

8.8.3.3 Return Codes

Table 8-69: Get List of TAs Command Return Codes

Return Code	Reason
TEE_SUCCESS	Command has been successfully executed.
TEE_ERROR_BAD_PARAMETERS	Invalid command parameter.
TEE_ERROR_BAD_FORMAT	Bad parameter format (duplicate constraint values if the Token is present…).
TEE_ERROR_LIMIT_EXCEEDED	The response demands more space than the TEE is able to provide in a single response.
TEE_ERROR_OUT_OF_MEMORY	There are not enough resources to perform this operation.
TEE_ERROR_SHORT_BUFFER	The response demands more space than the caller has indicated it is prepared to accept.
TEE_ERROR_ITEM_NOT_FOUND	The caller requested the list of Trusted Applications for a Security Domain which does not exist.

8.8.4 Get TA Definition

The Get TA Definition command performs the *Get TA Definition* operation as specified in section 6.6.4.

8.8.4.1 Command Parameters

The `GetTADef` type is a Constructed type that encodes the Get TA Definition command and its parameters.

```
GetTADef ::= [APPLICATION 100] SEQUENCE {
           ta          UUID
}
```

With the following attribute defining the command parameter:

- **`ta`** – The UUID of the TA whose definition is to be retrieved.

The TLV encoding is defined as follows.

Table 8-70: Get TA Definition Command TLV Encoding

Tag	Length	Value Octets			Presence
`0x7f64`	`0x12`	`GetTADef` parameters			M
		Tag	**Length**	**Value Octets**	
		`0x43`	`0x10`	`ta`	M

8.8.4.2 Response Output

The `GetTADefResp` type is a Constructed type that encodes the data returned by the Get TA Definition command.

```
GetTADefResp ::= TrustedApplication
```

Where the `TrustedApplication` type definition and its encoding are defined in section 9.3.2.

The TLV encoding is defined as follows.

Table 8-71: Get TA Definition Response TLV Encoding

Tag	Length	Value Octets			Presence
`0x7f`	`0x12`	`GetTADefResp` parameters			M
		Tag	**Length**	**Value Octets**	
		`0x74`	`L`	`TrustedApplication`	M

8.8.4.3 Return Codes

Table 8-72: Get TA Definition Command Return Codes

Return Code	Reason
`TEE_SUCCESS`	Command has been successfully executed.
`TEE_ERROR_BAD_PARAMETERS`	Invalid command parameter.
`TEE_ERROR_BAD_FORMAT`	Bad parameter format (duplicate constraint values if the Token is present…).
`TEE_ERROR_INTERNAL`	The *Trusted Application definition* could not be read, or some other unspecified internal error.
`TEE_ERROR_LIMIT_EXCEEDED`	The response demands more space than the TEE is able to provide in a single response.
`TEE_ERROR_OUT_OF_MEMORY`	There are not enough resources to perform this operation.
`TEE_ERROR_SHORT_BUFFER`	The response demands more space than the caller has indicated it is prepared to accept.
`TEE_ERROR_ITEM_NOT_FOUND`	The caller requested a definition of a Trusted Application which does not exist.

8.8.5 Get TA Definition 1

New in version 1.1

The Get TA Definition 1 command performs the *Get TA Definition 1* operation as specified in section 6.6.5.

8.8.5.1 Command Parameters

The `GetTADef1` type is a Constructed type that encodes the Get TA Definition 1 command and its parameters.

```
GetTADef1 ::= [APPLICATION 101] SEQUENCE {
          ta          UUID,
          version     INTEGER
}
```

With the following attributes defining the command parameter:

- **`ta`** – The UUID of the TA whose definition is to be retrieved.
- **`version`** – The required version of the structure to include in the response. For version 1.1. of this specification, only version 0 is supported.

The TLV encoding is defined as follows.

Table 8-73: Get TA Definition 1 Command TLV Encoding

Tag	Length	Value Octets			Presence
0x7f65	0x12	GetTADef1 parameters			M
		Tag	**Length**	**Value Octets**	
		0x43	0x10	ta	M
		0x03	0x04	version	M

8.8.5.2 Response Output

The `GetTADef1Resp` type is a Constructed type that encodes the data returned by the Get TA Definition command.

```
GetTADef1Resp ::= TrustedApplication1
```

Where the `TrustedApplication1` type definition and its encoding are defined in section 9.3.3.

The server must not return a structure version higher than requested in the command.

If the client requests a version higher than the server understands, the server should return the highest known version.

The TLV encoding is defined as follows.

Table 8-74: Get TA Definition 1 Response TLV Encoding

Tag	Length	Value Octets			Presence
0x7f66	L	GetTADef1Resp parameters			M
		Tag	**Length**	**Value Octets**	
		0x7e	L	TrustedApplication1	M

8.8.5.3 Return Codes

Table 8-75: Get TA Definition 1 Command Return Codes

Return Code	Reason
TEE_SUCCESS	Command has been successfully executed.
TEE_ERROR_BAD_PARAMETERS	Invalid command parameter.
TEE_ERROR_BAD_FORMAT	Bad parameter format (duplicate constraint values if the Token is present…).
TEE_ERROR_INTERNAL	The *Trusted Application definition* could not be read, or some other unspecified internal error.
TEE_ERROR_LIMIT_EXCEEDED	The response demands more space than the TEE is able to provide in a single response.
TEE_ERROR_OUT_OF_MEMORY	There are not enough resources to perform this operation.
TEE_ERROR_SHORT_BUFFER	The response demands more space than the caller has indicated it is prepared to accept.
TEE_ERROR_ITEM_NOT_FOUND	The caller requested a definition of a Trusted Application which does not exist.

9 Audit Information Encoding

9.1 TEE Characteristics

This section defines structures to describe the characteristics of the TEE based on the grammar and rules defined in Chapter 7.

This description is designed to be updated when the TEE software is updated; it is otherwise static.

9.1.1 SecureLayerAuditInfo Type

The `SecureLayerAuditInfo` type is a Constructed type which describes the information related to the Security Layer implementation supported by the Trusted OS of the TEE. The protocol information values defined hereafter may be used to describe a specific secure channel as well as any mechanism proving the trustworthiness of an SD installed on the TEE (e.g. a certificate signed by a trusted parent SD…).

Future GlobalPlatform specifications related to the Security Layer implementation (see section 8.2) SHOULD specify protocol UUIDs and their associated parameters that will be returned to a remote Authority using the TEE or SD audit commands.

```
SecureLayerAuditInfo ::= [APPLICATION 29] SEQUENCE {
      protocol          UUID,
      protocolInfo      OCTET STRING OPTIONAL -- an ‘open’ type as mentioned in
                        section 7.1
}
```

With:

- **`protocol`** – A UUID defining a protocol identifier. This specification defines one protocol identifier value (see Table A-3) corresponding to the generic protocol as defined in section 8.2.
- **`protocolInfo`** – Optional information data related to the protocol (e.g. reference to certificates, keys…) used by an SD (or the BD) supporting such a protocol. It SHALL be defined as an ‘open’ type for future definitions.

The TLV encoding is defined as follows.

Table 9-1: SecureLayerAuditInfo TLV Encoding

Tag	Length	Value Octets			Presence
`0x7d`	L	`SecureLayerAuditInfo` value			M
		Tag	**Length**	**Value Octets**	
		`0x43`	`0x10`	`protocol`	M
		`0x04`	L	`protocolInfo` The TLV structure (DER encoded) of the ‘open’ type defining the protocol information data	O

9.1.2 Option Type

The `Option` type is a Constructed type which describes an element consisting of a name and a version.

```
  Option ::= [APPLICATION 12] SEQUENCE {
      name        UTF8String,
      version     INTEGER
  }
```

With:

- **name** – Any UTF-8 string encoded as UTF8String according to [ASN.1].
- **version** – A version number encoded as described in section A.4.

The TLV encoding is defined as follows.

Table 9-2: Option TLV Encoding

Tag	Length	Value Octets			Presence
0x6c	L	Option value			M
		Tag	**Length**	**Value Octets**	
		0x0c	L	name	M
		0x02	L	version	M

9.1.3 Device Type

The `Device` type is a Constructed type which describes the details of the device (e.g. device name, device identifier …).

```
Device ::= [APPLICATION 13] SEQUENCE {
    name                UTF8String,
    id                  UUID                OPTIONAL,
    manufacturer        UTF8String,
    firmwareVersion     PrintableString,
    type                UTF8String          OPTIONAL
}
```

With:

- **name** – The name of the device encoded as PrintableString according to [ASN.1]. It denotes the “default” name of the TEE that a device application could refer to when establishing a connection to this particular TEE (as specified in [TEE Client]).
- **id** – The value of the `gpd.tee.deviceID` property encoded as a UUID. For privacy reasons, this field may be optional.
- **manufacturer** – The value of the `gpd.tee.firmware.manufacturer` property encoded as UTF8String according to [ASN.1].
- **firmwareVersion** – The firmware version of the device encoded as PrintableString according to [ASN.1].
- **type** – Describes the type of device encoded as UTF8String according to [ASN.1]. This field is optional.

The TLV encoding is defined as follows.

Table 9-3: Device TLV Encoding

Tag	Length	Value Octets			Presence
0x6d	L	Device value			M
		Tag	**Length**	**Value Octets**	
		0x0c	L	name	M
		0x43	0x10	id	O
		0x0c	L	manufacturer	M
		0x12	L	firmwareVersion	M
		0x0c	L	type	O

9.1.4 ISA Type

The ISA type is a Constructed type which describes the details of an Instruction set and architecture which can be used by Trusted Applications running in the TEE.[1]

```
ISA ::= [APPLICATION 14] SEQUENCE {
    name               UTF8String,
    processorType      UTF8String,
    instructionSet     PrintableString,
    addressSize        INTEGER,
    abi                PrintableString,
    endianness         INTEGER { little(0), big(1), middle(2) }
}
```

With:

- **name** – Specifies a human readable description of the environment, encoded as UTF8String according to [ASN.1].
- **processorType** – Indicates the type of the processor as a string, encoded as UTF8String according to [ASN.1].
- **instructionSet** – Specifies the instruction set as a string, encoded as PrintableString according to [ASN.1].
- **addressSize** – Specifies the size of addresses in bits as a number, encoded as INTEGER according to [ASN.1].
- **abi** – Specifies the Application Binary Interface which is in use, encoded as PrintableString according to [ASN.1].
- **endianness** – Specifies how values greater than 1 byte in length are stored.

The TLV encoding is defined as follows.

Table 9-4: ISA TLV Encoding

<table>
<tr><th>Tag</th><th>Length</th><th colspan="3">Value Octets</th><th>Presence</th></tr>
<tr><td rowspan="8">0x6e</td><td rowspan="8">L</td><td colspan="3">ISA value</td><td>M</td></tr>
<tr><th>Tag</th><th>Length</th><th>Value Octets</th><th></th></tr>
<tr><td>0x0c</td><td>L</td><td>name</td><td>M</td></tr>
<tr><td>0x0c</td><td>0x10</td><td>processorType</td><td>M</td></tr>
<tr><td>0x12</td><td>L</td><td>instructionSet</td><td>M</td></tr>
<tr><td>0x02</td><td>L</td><td>addressSize</td><td>M</td></tr>
<tr><td>0x12</td><td>L</td><td>abi</td><td>M</td></tr>
<tr><td>0x02</td><td>0x01</td><td>endianness (in range [0..2])</td><td>M</td></tr>
</table>

[1] Remember that this can be totally different from that which is in use in the REE.

9.1.5 TrustedOS Type

The `TrustedOS` type is a Constructed type which describes the details of the Trusted OS being run.

```
  TrustedOS ::= [APPLICATION 15] SEQUENCE {
       name               UTF8String,
       manufacturer       UTF8String,
       version            PrintableString,
       isaSet             SEQUENCE OF ISA,
       options            [0] SEQUENCE OF Option   OPTIONAL,
       protocols          [1] SEQUENCE OF SecureLayerAuditInfo  OPTIONAL
  }
```

With:

- **`name`** – The name of the Trusted OS as a UTF-8 string, encoded as UTF8String according to [ASN.1].
- **`manufacturer`** – The value of the `gpd.tee.trustedos.manufacturer` property, encoded as UTF8String according to [ASN.1].
- **`version`** – The value of the `gpd.tee.trustedos.implementation.version` property, encoded as PrintableString according to [ASN.1].
- **`isaSet`** – A list of instruction sets and architectures (for ISA type, see section 9.1.4), supported by the TEE. This list must consist of at least one element and be encoded as a `SEQUENCE OF ISA` according to [ASN.1].
- **`options`** – List of options (for `Option` type, see section 9.1.2) supported by TEE. Trusted OS may support additional options not specified by GlobalPlatform which may provide additional APIs which are useful to applications. Each such option is indicated by an `Option` type. The valid options are defined by the Trusted OS and are vendor specific. This element is optional and if present, the list must contain at least one element and be encoded as a `SEQUENCE OF Option` according to [ASN.1].
- **`protocols`** – A list of protocols supported by the Trusted OS related to the Security Layer implementation (see section 9.1.1).

The general TLV encoding is defined as follows.

Table 9-5: TrustedOS TLV Encoding

Tag	Length	Value Octets			Presence
0x6f	L	TrustedOS value			M
		Tag	**Length**	**Value Octets**	
		0x0c	L	name	M
		0x0c	L	manufacturer	M
		0x12	L	version	M
		0x30	L	isaSet (list of ISA values)	M
		0xa0	L	options (list of Option values)	O
		0xa1	L	protocols (list of SecureLayerAuditInfo values)	O

9.1.6 Tee Type

The `Tee` type is a Constructed type which describes a structure to describe the TEE characteristics and capabilities retrieved using the Get TEE Definition command (see section 8.8.1). There is one per TEE.

```
Tee ::= [APPLICATION 16] SEQUENCE {
    device                          Device,
    trustedOs                       TrustedOS,
    state                           INTEGER { locked(0), secure(1) },
    roots                           SEQUENCE OF UUID,
    optionalApis                    [0] SEQUENCE OF Option   OPTIONAL,
    teeImplementationProperties     [1] SEQUENCE OF Property OPTIONAL,
    teePlatformLabel                UTF8String
}
```

With:

- **`device`** – Details about the device, encoded according to section 9.1.3.
- **`trustedOs`** – Details about the TEE, encoded according to section 9.1.5.
- **`state`** – The current life cycle state of the TEE. This specification defines only the `TEE_LOCKED` and `TEE_SECURED` states.
- **`roots`** – The list of rSDs installed in the TEE and identified by their UUID (see section 4.1.3.3 for the definition of an rSD).
- **`optionalApis`** – An optional list of optional APIs which are implemented:
  - The list of valid API strings will be defined by the individual specifications.
  - Each optional API which is implemented is encoded into this list as an `Option` type according to section 9.1.2.
  - If this element is present it must contain at least one element. The list itself must be encoded as a `SEQUENCE OF Option` values according to [ASN.1].
- **`teeImplementationProperties`** – An optional list of TEE properties encoded as a `SEQUENCE OF Property` values, where `Property` type is defined in section 8.3.3.9.
- **`teePlatformLabel`** – The value of the `gpd.tee.platformLabel` property encoded as UTF8String according to [ASN.1]. This value reflects an indication about the certification by GlobalPlatform of the TEE.

The general TLV encoding is defined as follows.

Table 9-6: Tee TLV Encoding

Tag	Length	Value Octets			Presence
0x70	L	Tee value			M
		Tag	**Length**	**Value Octets**	
		0x6d	L	device	M
		0x6f	L	trustedOs	M
		0x02	0x01	state (either 0 = locked or 1 = secure)	M
		0x30	L	roots (list of UUID type values)	M
		0xa0	L	optionalApis (list of Option values; see Table 9-7)	O
		0xa1	L	teeImplementationProperties (list of Property values; see example below)	O
		0x0c	L (possibly equals zero for a null string)	teePlatformLabel	M

The following table defines the valid API name strings to be used for optionalApis attributes.

Table 9-7: Internal API Names Strings Definition

Strings	Description
TrustedUI	Trusted UI API
SE	Secure Element API
Debug-PMR	Debug PMR API
Debug-DLM	Debug DLM API
Sockets	Sockets API
TMF	ASN.1 Profile of TEE Management Framework

The following informal description is an example of the `teeImplementationProperties` attribute:

```
teeImplementationProperties {
        Property { name "gpd.tee.apiversion", value (UTF-8 string) "1.1"},
        Property { name "gpd.tee.internalCore.version", value (integer) 0x01010200}, -- 1.1.2
        Property { name "gpd.tee.description", value (UTF-8 string) "Trustonic's latest and greatest" },
        Property { name "gpd.tee.deviceID", value (UUID) <deviceUUID> },  -- as an OCTET STRING
        Property { name "gpd.tee.systemTime.protectionLevel", value (integer) 0x3e8 }, -- 1000
        Property { name "gpd.tee.TAPersistentTime.protectionLevel", value (integer) 0x64}, -- 100
        Property { name "gpd.tee.trustedos.implementation.version", value (UTF-8 string)  “1.3pl94” },
        Property { name "gpd.tee.firmware.manufacturer", value (UTF-8 string)  “xxxyyyy” },
        Property { name "gpd.tee.tmf.resetpreserved.entities", value (binary)  <Base64(concatenated
UUIDS)> } – as an OCTET STRING
        ....
 }
```

9.2 SD Characteristics

This section defines the structures to describe the characteristics of a Security Domain based on the grammar and rules defined in Chapter 7.

9.2.1 SDLifecycleState Type

The `SDLifecycleState` type is a Primitive type which describes the current life cycle of a Security Domain (see section 4.4).

Any possible values of this integer type are in the range [1..127] where the standard values denoting the Blocked, Active, and Restricted life cycle states are defined by the following values:

Table 9-7b: SDLifecycleState Values

SDLifecycleState	Value
sdBlockedState	INTEGER ::= 0
sdActiveState	INTEGER ::= 1
sdRestrictedState	INTEGER ::= 2

An `SDLifecycleState` value is expressed by the following type that combines these possible values with some extended RFU or vendor-specific values (denoted by the extension marker "…" according to [ASN.1]):

```
SDLifecycleState ::= [APPLICATION 17] INTEGER(0..127) (sdBlockedState |
sdActiveState | sdRestrictedState, … )
```

The TLV encoding is defined as follows.

Table 9-8: SDLifecycleState TLV Encoding

Tag	Length	Value Octets	Presence
0x51	0x01	`SDLifecycleState` value in range [0..2] for the standard states, in range [3..63] for values reserved for future usage, and in range [64..127] for vendor-specific values.	M

9.2.2 SecurityDomain Type

The `SecurityDomain` type is a Constructed type which defines a structure to describe the characteristics of a Security Domain retrieved using the Get SD Definition command (see section 8.8.2). There is one of these for each Security Domain in each TEE.

```
SecurityDomain ::= [APPLICATION 18] SEQUENCE {
    id                UUID,
    parent            UUID                    OPTIONAL,
    lifecycleState    SDLifecycleState,
    authority         Authority               OPTIONAL,
    privileges        SDPrivileges            OPTIONAL,
    subdomains        [0] SEQUENCE OF UUID    OPTIONAL,
    protocols         [1] SEQUENCE OF SecureLayerAuditInfo  OPTIONAL
}
```

With:

- **`id`** – The UUID used by client entities to communicate with the Security Domain.
- **`parent`** – The identity of the parent Security Domain, if any. If present, this element is encoded as a UUID.
- **`lifecycleState`** – The current life cycle state of the Security Domain encoded according to section 9.2.1.
- **`authority`** – Optional details (name and/or URL) of the Authority that manages this Security Domain.
- **`privileges`** – The privileges of this Security Domain. This information is optional to let an implementation decide whether or not to publish it outside the TEE.
- **`subdomains`** – An optional list of the child Security Domains below this one. Each child Security Domain is specified using the corresponding UUID. If this element is present, the list must contain at least one element and be encoded as `SEQUENCE OF` according to [ASN.1].
- **`protocols`** – A list of protocols supported by this SD related to the Security Layer implementation (see section 9.1.1).

The TLV encoding is defined as follows.

Table 9-9: Security Domain Characteristics TLV Encoding

Tag	Length	Value Octets			Presence
`0x72`	`L`	`SecurityDomain` value			M
		Tag	**Length**	**Value Octets**	
		`0x43`	`0x10`	`id`	M
		`0x43`	`0x10`	`parent`	O
		`0x51`	`0x01`	`lifecycleState`	M
		`0x7c`	`L`	`authority`	O
		`0x7b`	`L`	`privileges`	O
		`0xa0`	`L`	`subdomains` (list of UUID values)	O
		`0xa1`	`L`	`protocols` (list of `SecureLayerAuditInfo` values)	O

9.3 TA Characteristics

This section defines the structures to describe the characteristics of a Trusted Application based on the grammar and rules defined in Chapter 7.

9.3.1 TALifecycleState Type

The `TALifecycleState` type is a Primitive type which describes the life cycle state of a TA (see section 4.3).

Any possible values of this integer type are in the range [1..127]. The standard values denoting the Inactive, Executable, and Locked life cycle states are defined as follows:

Table 9-9b: TALifecycleState Values

TALifecycleState	**Value**
taInactiveState	INTEGER ::= 0
taExecutableState	INTEGER ::= 1
taLockedState	INTEGER ::= 2

The `TALifecycleState` value is expressed by the following type that combines these possible values with some extended RFU or vendor-specific values (denoted by the extension marker “…” according to [ASN.1]):

```
TALifecycleState ::= [APPLICATION 19] INTEGER(0..127) (taInactiveState |
taExecutableState | taLockedState, … )
```

The TLV encoding is defined as follows.

Table 9-10: TALifecycleState TLV Encoding

Tag	Length	Value Octets	Presence
0x53	0x01	`TALifecycleState` value in range [0..2] for the standard states, in range [3..63] for values reserved for future usage, and in range [64..127] for vendor-specific values, if any.	M

9.3.2 TrustedApplication Type

The `TrustedApplication` type is a Constructed type which describes the structure defining the characteristics and capabilities of a TA. This structure is returned as a result of the Get TA Definition command (see section 8.8.4).

```
TrustedApplication ::= [APPLICATION 20] SEQUENCE {
    id                UUID,
    parent            UUID,
    lifecycleState    TALifecycleState,
    version           PrintableString
}
```

With:

- **`id`** – The UUID of the TA
- **`parent`** – The value of the `gpd.ta.parentSD` property, encoded as UUID
- **`lifecycleState`** – The state of the TA in its life cycle
- **`version`** – The value of the `gpd.ta.version` property, encoded as PrintableString according to [ASN.1]

The general TLV encoding is defined as follows.

Table 9-11: Trusted Application TLV Encoding

Tag	Length	Value Octets			Presence
0x74	L	TrustedApplication value			M
		Tag	**Length**	**Value Octets**	
		0x43	0x10	id	M
		0x43	0x10	parent	M
		0x53	0x01	lifecycleState	M
		0x12	L	version	M

9.3.3 TrustedApplication1 Type

New in version 1.1

The `TrustedApplication1` type is a Constructed type which describes the structure defining the characteristics and capabilities of a TA. This structure is returned as a result of the Get TA Definition 1 command (see section 8.8.5).

```
TrustedApplication1 ::= [APPLICATION 30] SEQUENCE {
    structureVersion   Integer                 MANDATORY,
    id                 UUID                    MANDATORY,
    parent             UUID                    MANDATORY,
    lifecycleState     TALifecycleState        MANDATORY,
    version            PrintableString         MANDATORY,
    versionNumber      Integer                 MANDATORY
}
```

With:

- **`structureVersion`** – For version 1.1 of this specification, the only valid value is 0. Future versions of this specification may introduce new `structureVersion` values to enable more fields to be selected.
- **`id`** – The UUID of the TA
- **`parent`** – The value of the `gpd.ta.parentSD` property, encoded as UUID
- **`lifecycleState`** – The state of the TA in its life cycle
- **`version`** – The value of the `gpd.ta.version` property, encoded as PrintableString according to [ASN.1]
- **`versionNumber`** – The value of the `gpd.ta.version.number` property, encoded as Integer according to [ASN.1]

The general TLV encoding is defined as follows.

Table 9-12: Trusted Application 1 TLV Encoding

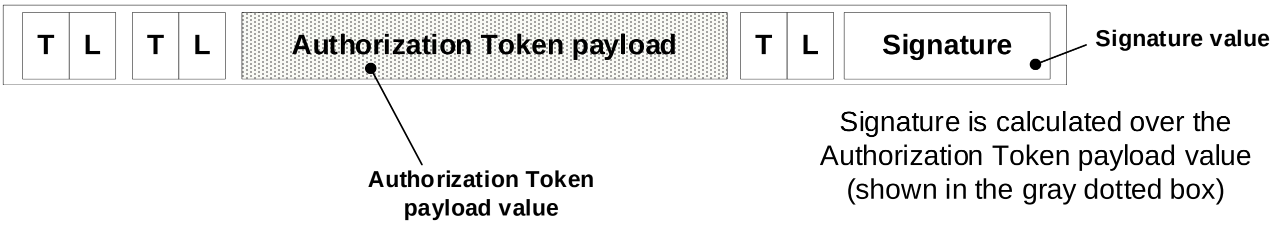
<table>
<tr><th>Tag</th><th>Length</th><th colspan="3">Value Octets</th><th>Presence</th></tr>
<tr><td rowspan="8">0x7e</td><td rowspan="8">L</td><td colspan="3">TrustedApplication1 value</td><td>M</td></tr>
<tr><th>Tag</th><th>Length</th><th>Value Octets</th><th></th></tr>
<tr><td>0x03</td><td>0x04</td><td>structureVersion</td><td>M</td></tr>
<tr><td>0x43</td><td>0x10</td><td>id</td><td>M</td></tr>
<tr><td>0x43</td><td>0x10</td><td>parent</td><td>M</td></tr>
<tr><td>0x53</td><td>0x01</td><td>lifecycleState</td><td>M</td></tr>
<tr><td>0x12</td><td>L</td><td>version</td><td>M</td></tr>
<tr><td>0x02</td><td>0x04</td><td>versionNumber</td><td>M</td></tr>
</table>

10 Authorization Token Format

The Authorization Token payload and its signature, which guarantees its integrity, are encoded using the TLV format rules defined in Chapter 7.

The following figure provides an overview of the Authorization Token.

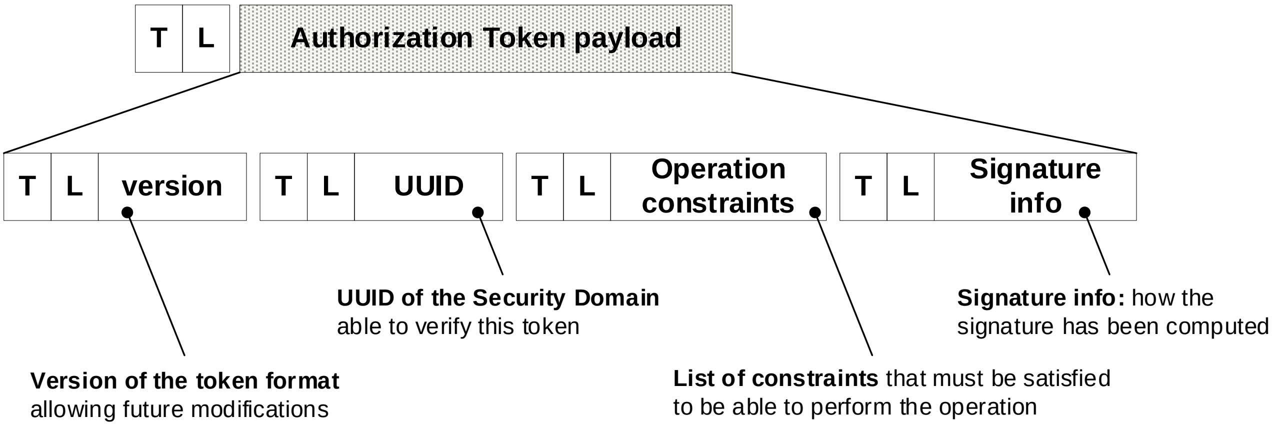
Figure 10-1: Authorization Token Format


T
L
T
L
Authorization Token payload
T
L
Signature
Signature value
Authorization Token payload value
Signature is calculated over the Authorization Token payload value (shown in the gray dotted box)

The TLV encoding is described in section 10.1.4.

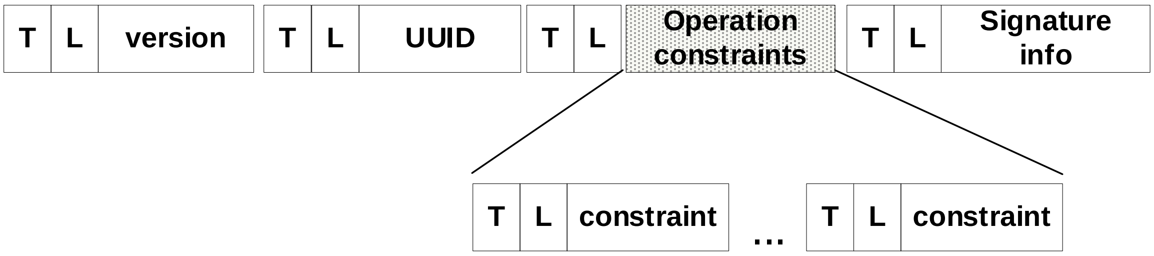
The following figure illustrates the Authorization Token payload, containing the token information.

Figure 10-2: Authorization Token Payload Format


T
L
Authorization Token payload
T
L
version
T
L
UUID
T
L
Operation constraints
T
L
Signature info
UUID of the Security Domain able to verify this token
Signature info: how the signature has been computed
Version of the token format allowing future modifications
List of constraints that must be satisfied to be able to perform the operation

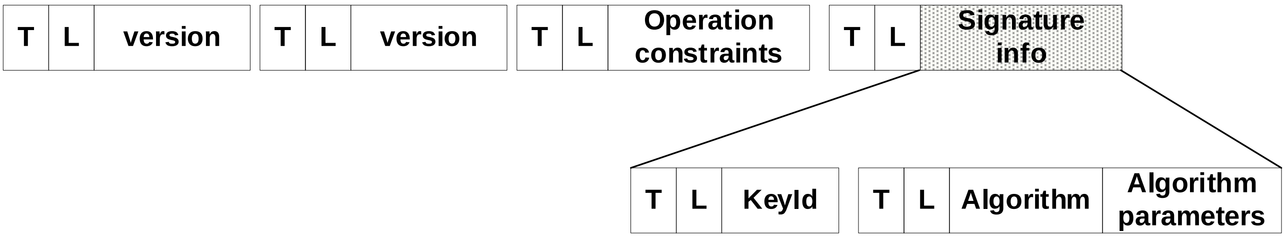
The following figure illustrates the operation constraints format.

Figure 10-3: Authorization Token: Operation Constraints Format



The following figure illustrates the Signature information.

Figure 10-4: Authorization Token: Signature Info Format



10.1 TLV Structure Definitions

10.1.1 TokenConstraint Type

The `TokenConstraint` is either a Primitive or a Constructed type with a 'Private' class tag which defines different constraints that can be included in an Authorization Token and that must then be satisfied to allow the operation execution.

A `TokenConstraint` type can be informally represented by the following description:

`TokenConstraint ::= [PRIVATE <ConstraintTag>]` <any type definition according to [ASN.1]>

The TLV encoding is defined as follows.

Table 10-1: TokenConstraint TLV Encoding

Tag	Length	Value Octets	Presence
`<ConstraintTag>`	L	Token constraint value (see below).	M

The following table defines each available `ConstraintTag` Octet Identifier (as a Private class tag), its meaning, and the corresponding contents.

Table 10-2: ConstraintTag Octet Identifier Values

Constraint Name	ITU-X680 notation	`ConstraintTag` Octet Identifier Values	Constraint Value Octets
ConstraintDeviceId	[PRIVATE 1] UUID	`0xc1`	The UUID of the device encoded as a UUID type
ConstraintModelId	[PRIVATE 2] UUID	`0xc2`	The UUID of the model encoded as a UUID type
ConstraintMinVersion	[PRIVATE 3] INTEGER	`0xc3`	Minimal Version of the TA encoded as INTEGER
ConstraintMaxVersion	[PRIVATE 4] INTEGER	`0xc4`	Maximal Version of the TA encoded as INTEGER
ConstraintParamsDigest	See ConstraintParamsDigest type definition in section 10.1.2 (a Private class tag of a Constructed type)	`0xe0`	Digest over the command parameters encoded as ConstraintParamsDigest.

Reserved for future use	Any Private class Primitive type with tag numbers in range [6 – 30]
	Any Private class Constructed type with tag numbers in range [1 – 30]
		`[0xc5 – 0xde]`
		`[0xe1 – 0xfe]`
			Reserved

Proprietary extensions	See Chapter 7		Vendor-specific

See section A.8 for a formal description of the constraint types.

10.1.2 ConstraintParamsDigest Type

The `ConstraintParamsDigest` type is a Private Constructed type which defines a structure that encapsulates information about a digest value and the corresponding algorithm having been used for the calculation of the operation constraint over the whole or a part of the command parameters.

```
ConstraintParamsDigest ::= [PRIVATE 0] SEQUENCE {
    algorithmID  INTEGER,
    bitmap       INTEGER,
    digest       OCTET STRING
}
```

With:

- **`algorithmID`** – The identifier of the algorithm used to calculate the digest value (as defined in [TEE Core API])
  - See Table A-8 for the mandatory and optional algorithms.
- **`bitmap`** – A bitmap value where bit `0` (the least significant bit) corresponds to the command tag (defined by Table 8-7), bit 1 corresponds to the first parameter, bit 2 corresponds to the second parameter, and bit N corresponds to the Nth parameter of the administrative command on which the digest value is calculated
  - If a bit is set to 1, it indicates that the corresponding parameter value has been included in the computation of the `digest` value.
  - The bitmap value SHALL NOT be equal to zero (at least one bit SHOULD be set).
  - Verification of the constraint SHALL fail if the bitmap value indicates some parameter numbers that are not defined by a given command.
    - For example, the bitmap value for a constraint on parameters of a Lock TEE command (see section 8.7.1.1) cannot be different from 1 (bit `0` set to 1 – the command has only a command tag and no parameters).
- **`digest`** – The digest value itself
  - The digest value has been computed as follows (and SHALL be verified accordingly):
    - In order from the least significant to the most significant bits of the `bitmap` value, concatenate the values of all parameters that are set to 1 in the bitmap.
      - For bit `0` (the least significant), the value is the command tag value (see Table 8-7).
      - For other bits, the value is the tag-length-value octets of the corresponding parameter number as encoded in the administrative command.
    - Then, apply the digest operation (defined by the `algorithmID` field) on the list of concatenated values.

The general TLV encoding is defined as follows.

Table 10-3: ConstraintParamsDigest TLV Encoding

Tag	Length	Value Octets			Presence
`0xe0`	`L`	`ConstraintParamsDigest` value			M
		Tag	**Length**	**Value Octets**	
		`0x02`	`0x04`	`algorithmID`	M
		`0x02`	`L`	`bitmap`	M
		`0x04`	`L`	`digest`	M

10.1.3 AuthorizationTokenPayload Type

The `AuthorizationTokenPayload` type is a Constructed type that defines the content of an Authorization Token.

```
AuthorizationTokenPayload ::= [APPLICATION 21] SEQUENCE {
    version                 INTEGER,
    authorizingSd           UUID,
    constraintsList         SEQUENCE OF TokenConstraint,
    signatureInfo           KeyRefParameters
}
```

With:

- **`version`** – The version of this specification identified by the `gpd.tee.tmf.version` property (see Table A-4), or any prior version
- **`authorizingSd`** – The UUID of the Security Domain, which is able to verify this token
- **`constraintsList`** – The list of constraints that must be satisfied to be able to perform the operation. There SHALL NOT be duplicate values of constraints.
- **`signatureInfo`** – The information indicating the signature key identifier and how the signature has been calculated (algorithm, extra parameters associated with the algorithm…)

The general TLV encoding is defined as follows.

Table 10-4: AuthorizationTokenPayload TLV Encoding

<table>
<tr><th>Tag</th><th>Length</th><th colspan="3">Value Octets</th><th>Presence</th></tr>
<tr><td rowspan="6">0x75</td><td rowspan="6">L</td><td colspan="3">AuthorizationTokenPayload value</td><td>M</td></tr>
<tr><th>Tag</th><th>Length</th><th>Value Octets</th><th></th></tr>
<tr><td>0x02</td><td>L</td><td>gpd.tee.tmf.version
(the current value of this property or a value identifying a prior version)</td><td>M</td></tr>
<tr><td>0x43</td><td>0x10</td><td>authorizingSd</td><td>M</td></tr>
<tr><td>0x30</td><td>L</td><td>constraintsList
(list of TokenConstraint values)</td><td>M</td></tr>
<tr><td>0x66</td><td>L</td><td>signatureInfo</td><td>M</td></tr>
</table>

10.1.4 AuthorizationToken Type

The `AuthorizationToken` type is a Constructed type that defines the structure of the Authorization Token Payload associated with its signature. This is this structure that can be optionally passed to the administration commands defined by this specification.

```
AuthorizationToken ::= [APPLICATION 22] SEQUENCE {
    payload            AuthorizationTokenPayload,
    signature          OCTET STRING
}
```

With:

- **`payload`** – The Authorization Token payload
- **`signature`** – The sequence of bytes of the payload signature. The signature is performed over the Authorization Token payload value octets as depicted in Figure 10-3.

The general TLV encoding is defined as follows.

Table 10-5: AuthorizationToken TLV Encoding

<table>
<tr><th>Tag</th><th>Length</th><th colspan="3">Value Octets</th><th>Presence</th></tr>
<tr><td rowspan="4">0x76</td><td rowspan="4">L</td><td colspan="3">AuthorizationToken value</td><td>M</td></tr>
<tr><th>Tag</th><th>Length</th><th>Value Octets</th><th></th></tr>
<tr><td>0x75</td><td>L</td><td>payload</td><td>M</td></tr>
<tr><td>0x04</td><td>L</td><td>signature</td><td>M</td></tr>
</table>

11 Forcing the Shutdown of a Trusted Application

At a number of points in this specification, the actions of the administration commands will shut down all sessions to a currently executing TA, effectively closing the TA.

For the purposes of this section, a TA shutdown state is a TA state with no active TA sessions or TA instance data.

Current examples of commands that invoke a TA shutdown state are Factory Reset, Lock TEE, Lock TA, Block SD, and Uninstall TA.

There are two scenarios for a TA shutdown:

- Shut down by related manager entity

  In this scenario the clients of the TA can be assumed to be known to the managing entity and as such the TA's clients can be informed through side channels that the shutdown is about to occur.

  Given these circumstances, it is recommended that the client close all active sessions before informing the local agent for the management entity that it is ready for the management entity to go ahead with the relevant management command.

- Shut down by unrelated manager entity

  In this scenario the clients of the TA cannot be assumed to be known to the managing entity and as such the TA's clients cannot be informed that the shutdown is about to occur.

In either of those scenarios the TA will be closed by following the sequence described in section 11.1.

Forcing Shutdown of Uncooperative TAs

Because a TA itself can, with good reason or bad coding, be written to not respond to various levels of legitimate session or instance close command, the sequence described in section 11.1 defines how a TA will act upon such shutdown commands.

From the point of view of this TA shutdown functionality, if a TA is not correctly shutting down given legitimate calls to its command interfaces, it is considered to be in an erroneous programming state and therefore the actions of this process at that point are equivalent to a Panic occurring due to other bad coding in the TA (see [TEE Core API] section 2.2.3). This may seem harsh but it must be assumed that the remote entity with the right to issue the relevant TMF command has good reason to force such a state change. If the remote entity does not wish to force a shutdown of a TA with live sessions or active instance data, then it must interact with either the TA or its Client Applications to protect against such an eventuality.

11.1 TA Shutdown Sequence

The following sequence SHALL be performed when a TA is shut down by a TMF command.

1. **Block further commands to the TA.**

   No further commands SHOULD be sent to the TA (either through `TEEC_CommandInvoke`, `TEE_InvokeTACommand`, `TEE_OpenTASession`, or `TEEC_OpenSession` function calls) and any attempt to do so SHALL receive `TEE_ERROR_TARGET_DEAD` with the origin `TEE_ORIGIN_TEE`.

2. **Cancel unprocessed commands in the TA's command queue.**

   All current commands in the command queue to the TA, but not being acted upon by the TA, SHALL be cancelled as though a client had called the `TEEC_RequestCancellation` function (see [TEE Core API] section 4.10 – Cancellation Functions).

   The corresponding `TEEC_CommandInvoke`, `TEE_InvokeTACommand`, `TEE_OpenTASession`, or `TEEC_OpenSession` function call SHALL return the `TEEC_ERROR_CANCEL` error code to the relevant REE or TEE Client Application.

3. **Cancel commands currently being processed by the TA.**

   Any command that the TA is currently acting on SHALL be cancelled as though the client had called the `TEEC_RequestCancellation` function. If the TA is engaged in a call with another TA, the cancellation request SHALL be propagated as stated in [TEE Core API] section 4.10 – Cancellation Functions.

   a. **I/O based Wait events**

   The `TEE_Wait` function calls and similar I/O events such as `TEE_TUIDisplayScreen` function calls are cancellable. They will return the `TEE_ERROR_EXTERNAL_CANCEL` return code if pending.

   b. **Panic if the command will not cancel.**

   If the command being processed by a TA does not return in a timely manner (1) then the TEE SHALL assume the TA is in an endless loop, and it SHALL effect a Panic on the TA with the panic context `PANIC_FAILED_COMMAND_SHUTDOWN` (see section A.1).

4. **Close all open sessions to the TA.**

   When there are no commands in the TA command queue or being acted upon by the TA, and no TA Panic invoked, then the TA SHALL receive the equivalent of a `TEEC_CloseSession(&session)` function call for all open sessions (i.e. `TA_CloseSessionEntryPoint(&session)` SHALL be called with the session context for each current session associated with the TA).

   a. **Panic if the session will not close.**

   If a TA does not return from `TA_CloseSessionEntryPoint` function call in a timely manner (1) then the TEE SHALL perform a `TEE_Panic` function call (see [TEE Core API] section 4.8) on the TA with the panic context `PANIC_FAILED_SESSION_SHUTDOWN` (see section A.1).

(1) Timely manner is implementation dependent, but should be no longer than 1 second.

5. Close any instance data of the TA.

When all sessions are successfully closed and no TA Panic invoked, the TA instance SHALL be closed (i.e. `TA_DestroyEntryPoint(void)` will be called). In an exception to the rule stated in [TEE Core API] Table 4-11, this SHALL occur even if the TA has `gpd.ta.instanceKeepAlive =` true. The return code from the `TA_DestroyEntryPoint` call SHALL be discarded.

a. Panic if the instance will not close.

If an instance does not return from `TA_DestroyEntryPoint(void)` in a timely manner [1] then the TEE SHALL perform a `TEE_Panic` function call (see [TEE Core API] section 4.8) on the TA with the panic context `PANIC_FAILED_INSTANCE_SHUTDOWN` (see section A.1).

6. The TA is now shut down.

The TEE SHALL have performed any internal housekeeping and all the TA's instances will be closed.

Any further attempts to start a TA Session SHALL return the error codes depending on the cause of shutdown (See Table 11-1). This allows the TA client to take appropriate action.

[1] Timely manner is implementation dependent, but should be no longer than 1 second.

11.2 Client API Error Codes Due to Administration State Changes

The following error codes will be received by client API users when affected by TEE state changes.

Table 11-1: Client Session Error Codes

System State	Client API Error Code	Value

TA locked	`TEEC_ERROR_TA_LOCKED,`
	`TEE_ERROR_TA_LOCKED`
		`0xFFFF0012`

SD Blocked	`TEEC_ERROR_SD_BLOCKED,`
	`TEE_ERROR_SD_LOCKED`
		`0xFFFF0013`

TEE Locked	`TEEC_ERROR_TEE_LOCKED,`
	`TEE_ERROR_TEE_LOCKED`
		`0xFFFF0014`

TA Uninstalled and session lost	`TEEC_ERROR_TA_UNINSTALLED,`
	`TEE_ERROR_TA_UNINSTALLED`
		`0xFFFF0015`

TEE Factory reset and session lost	`TEEC_ERROR_TEE_FACTORY_RESET,`
	`TEE_ERROR_TEE_FACTORY_RESET`
		`0xFFFF0016`

As the system may have multiple simultaneous states (e.g. a TA is locked cause its parent SD has been blocked, or an SD is blocked but the TEE is locked…), it is reasonable to establish a kind of precedence order over the system states information returned as the result of the TA shutdown sequence.

So, if the TA is shut down when:

- The TEE is locked or being locked, then the `TEEC_ERROR_TEE_LOCKED` (or `TEE_ERROR_TEE_LOCKED`) error code is returned.
- The parent SD is blocked or being blocked, then the `TEEC_ERROR_SD_BLOCKED` (or `TEE_ERROR_SD_BLOCKED`) error code is returned.
- The TA is being locked, then the `TEEC_ERROR_TA_LOCKED` (or `TEEC_ERROR_TA_LOCKED`) error code is returned.
- The TA is being uninstalled, then the `TEEC_ERROR_TA_UNINSTALLED` (or `TEE_ERROR_TA_UNINSTALLED`) error code is returned.
- The TEE is being reset, then the `TEEC_ERROR_TEE_FACTORY_RESET` (or `TEE_ERROR_TEE_FACTORY_RESET`) error code is returned.

Annex A Assigned Values (Normative)

A.1 Panic Context

If this specification is used in conjunction with the TEE TA Debug Specification ([TEE TA Debug]), then the specification number is 120 and the values listed in Table A-1 SHALL be associated with the described context.

Table A-1: Panic Context Identification

Context Identifier	Panic Context Identification in Hexadecimal
PANIC_FAILED_COMMAND_SHUTDOWN	0x101
PANIC_FAILED_SESSION_SHUTDOWN	0x102
PANIC_FAILED_INSTANCE_SHUTDOWN	0x103

A.2 Tag Definitions

Table A-2: List of Tags Defined by This Specification

Tag Name	Value	Type Description	Definition Reference
BOOLEAN	0x01	ITU standard [ASN.1]	Chapter 7
INTEGER	0x02	ITU standard [ASN.1]	Chapter 7
OCTET STRING	0x04	ITU standard [ASN.1]	Chapter 7
NULL	0x05	ITU standard [ASN.1]	Chapter 7
PrintableString	0x13	ITU standard [ASN.1]	Chapter 7
UTF8String	0x0c	ITU standard [ASN.1]	Chapter 7
SEQUENCE/SEQUENCE OF	0x30	ITU standard [ASN.1]	Chapter 7
APPLICATION 0	0x60	CmdReqPayload	Section 8.3.1
APPLICATION 1	0x61	CmdRespPayload	Section 8.3.2
APPLICATION 2	0x62	Attribute	Section 8.3.3.1
APPLICATION 3	0x43	UUID	Section 8.3.3.2
APPLICATION 4	0x44	ObjectId	Section 8.3.3.3
APPLICATION 5	0x65	CryptoOperationParameters	Section 8.3.3.4
APPLICATION 6	0x66	KeyRefParameters	Section 8.3.3.5
APPLICATION 7	0x67	StoredDataObject	Section 8.3.3.6
APPLICATION 8	0x68	UUIDVerificationParams	Section 8.3.3.7
APPLICATION 9	0x69	CryptographicData	Section 8.3.3.8
APPLICATION 10	0x6a	Property	Section 8.3.3.9
APPLICATION 11	0x6b	InstallSDResp	Section 8.5.1.2
APPLICATION 12	0x6c	Option	Section 9.1.2

Tag Name	Value	Type Description	Definition Reference
APPLICATION 13	0x6d	Device	Section 9.1.3
APPLICATION 14	0x6e	ISA	Section 9.1.4
APPLICATION 15	0x6f	TrustedOS	Section 9.1.5
APPLICATION 16	0x70	Tee	Section 9.1.6
APPLICATION 17	0x51	SDLifecycleState	Section 9.2.1
APPLICATION 18	0x72	SecurityDomain	Section 9.2.2
APPLICATION 19	0x53	TALifecycleState	Section 9.3.1
APPLICATION 20	0x74	TrustedApplication	Section 9.3.2
APPLICATION 21	0x75	AuthorizationTokenPayload	Section 10.1.3
APPLICATION 22	0x76	AuthorizationToken	Section 10.1.4
APPLICATION 23	0x77	SecurityContainer	Section 8.2
APPLICATION 25	0x79	ListObjectsResp	Section 8.6.3.2
APPLICATION 26	0x7a	GetListOfTAResp	Section 8.8.3.2
APPLICATION 27	0x7b	SDPrivileges	Section 8.3.3.10
APPLICATION 28	0x7c	Authority	Section 8.3.3.11
APPLICATION 29	0x7d	SecureLayerAuditInfo	Section 9.1.1
APPLICATION 30	0x7e	TrustedApplication1	Section 9.3.3
APPLICATION 31	0x7f	GetTADefResp	Section 8.8.4.2
PRIVATE 0	0xe0	ConstraintParamsDigest	Section 10.1.2
PRIVATE 1 PRIVATE 2 PRIVATE 3 PRIVATE 4	0xc1 0xc2 0xc3 0xc4	ConstraintDeviceId ConstraintModelId ConstraintMinVersion ConstraintMaxVersion	Section 10.1.1
APPLICATION 65	0x7f41	InstallTA	Section 8.4.1.1
APPLICATION 66	0x7f42	UninstallTA	Section 8.4.2.1
APPLICATION 67	0x7f43	UpdateTA	Section 8.4.3.1
APPLICATION 68	0x7f44	LockTA	Section 8.4.4.1
APPLICATION 69	0x7f45	UnlockTA	Section 8.4.5.1
APPLICATION 70	0x7f46	UpdateTAandData	Section 8.4.6.1
APPLICATION 74	0x7f4a	InstallSD	Section 8.5.1.1
APPLICATION 75	0x7f4b	UninstallSD	Section 8.5.2.1
APPLICATION 77	0x7f4d	BlockSD	Section 8.5.3.1
APPLICATION 78	0x7f4e	UnblockSD	Section 8.5.4.1
APPLICATION 79	0x7f4f	RestrictSD	Section 8.5.5.1

Tag Name	Value	Type Description	Definition Reference
APPLICATION 80	`0x7f50`	`UnrestrictSD`	Section 8.5.6.1
APPLICATION 85	`0x7f55`	`StoreData`	Section 8.6.1.1
APPLICATION 86	`0x7f56`	`DeleteData`	Section 8.6.2.1
APPLICATION 87	`0x7f57`	`ListObjects`	Section 8.6.3.1
APPLICATION 88	`0x7f58`	`FetchObject`	Section 8.6.4.1
APPLICATION 90	`0x7f5a`	`LockTEE`	Section 8.7.1.1
APPLICATION 91	`0x7f5b`	`UnlockTEE`	Section 8.7.2.1
APPLICATION 92	`0x7f5c`	`StoreTEEProperty`	Section 8.7.3.1
APPLICATION 93	`0x7f5d`	`FactoryReset`	Section 8.7.4.1
APPLICATION 97	`0x7f61`	`GetTEEDef`	Section 8.8.1.1
APPLICATION 98	`0x7f62`	`GetSDDef`	Section 8.8.2.1
APPLICATION 99	`0x7f63`	`GetListOfTA`	Section 8.8.3.1
APPLICATION 100	`0x7f64`	`GetTADef`	Section 8.8.5.1
APPLICATION 101	`0x7f65`	`GetTADef1`	Section 8.8.5.1
APPLICATION 102	`0x7f66`	`GetTADef1Resp`	Section 8.8.5.2
APPLICATION 103	`0x7f67`	`FetchObjectResp`	Section 8.6.4.2
APPLICATION 104	`0x7f68`	`GetTEEDefResp`	Section 8.8.1.2
APPLICATION 105	`0x7f69`	`GetSDDefResp`	Section 8.8.2.2

A.3 Specification UUIDs

Table A-3: Specification Reserved UUIDs

Reserved UUID	Description
`2329A4EA-B484-47E4-9B65-262D726B3438`	The UUID of the TMF audit SD able to perform any unprivileged audit commands.
`6BC2DE43-5012-4855-9C8E-EAAF0CB9FDE7`	The UUID of the “UUID v5 protocol” to verify the proof of possession of a UUID v5.
`87B16ABA-879B-4C7E-91CE-DD4B600F1390`	The UUID identifying the generic protocol corresponding to the usage of the generic container type in the Security Layer as defined in section 8.2.
`2f4d366b-2184-4879-9563-d123c1d6087e`	The UUID identifying the TMF: Symmetric Cryptography Security Layer
`96cc8dac-9b56-4f85-8dd1-5c30cd7037ea`	The UUID identifying the TMF: Asymmetric Cryptography Security Layer

A.4 Specification Version Numbers

Several type structures defined in this document reference the version number of a GlobalPlatform specification. Each such version number SHALL be encoded as an unsigned 32-bit integer where the bytes are filled as:

Byte `0` (least significant byte):	Reserved for future usage (currently SHALL be zero)
Byte 1:	Maintenance version number from relevant GlobalPlatform specification (SHALL be zero when not used)
Byte 2:	Minor version number from relevant GlobalPlatform specification
Byte 3:	Major version number from relevant GlobalPlatform specification

For example, the version number of the first release of this document is encoded with the hexadecimal value `0x01000000`.

A.5 Specification Properties

The `gpd.tee.tmf.*` properties listed in Table A-4 can be retrieved by the generic Property Access Functions with the `TEE_PROPSET_TEE_IMPLEMENTATION` pseudo-handle (see [TEE Core API]).

- The property `gpd.ta.parentSD` can be retrieved by a TA using these generic functions with the `TEE_PROPSET_CURRENT_TA` pseudo-handle.
- The property `gpd.client.parentSD` can be retrieved by a TA (called by a client TA) using these generic functions with the `TEE_PROPSET_CURRENT_CLIENT` pseudo-handle.

The `gpd.sd.isRootSD` property of an SD is flagged internally by the TEE at SD installation time and SHOULD NOT be retrieved using these generic functions.

Table A-4: Specification Reserved Properties

Property	Property Type	Comment
`gpd.sd.isRootSD`	`boolean`	Property that is set internally by the TEE when successfully installing a new rSD.
`gpd.ta.parentSD`	`UUID`	The UUID of the direct parent SD of a TA. (See section 4.1.2.)
`gpd.client.parentSD`	`UUID`	The UUID of the direct parent SD of a TA. (See section 4.1.2.)
`gpd.tee.tmf.hierarchies` (1)	`uint32_t`	Maximum number of SD hierarchies (equals the maximum number of root SDs).
`gpd.tee.tmf.hierarchy.max_depth` (1)	`uint32_t`	Maximum depth of a hierarchy (i.e. maximum distance from an SD to its rSD).
`gpd.tee.tmf.hierarchy.max_domains` (1)	`uint32_t`	Maximum number of SDs per hierarchy.
`gpd.tee.tmf.max_tee_apps` (1)	`uint32_t`	Maximum number of TAs in the TEE.
`gpd.tee.tmf.resetpreserved.entities`	`binary`	A base64 encoded list of concatenated UUID values. Each UUID represents an entity to be preserved across a *Factory Reset* operation on TEE.
`gpd.tee.tmf.sd.max_subdomains` (1)	`uint32_t`	Maximum number of direct or indirect sub-domains per SD.
`gpd.tee.tmf.sd.max_tee_apps` (1)	`uint32_t`	Maximum number of TAs per SD.
`gpd.tee.tmf.version`	`uint32_t`	The version of this specification, encoded as specified in section A.4.Currently this property indicates the version of both TMF and the ASN.1 Profile. It is anticipated that the ASN.1 Profile may become a separate specification in the future, at which point a separate property will be defined for the ASN.1 Profile version.

(1) While these properties define the maximum numbers that may be installed in the TEE when empty, the TEE may additionally be limited by dynamic resource availability. This particularly applies to `gpd.tee.tmf.max_tee_apps` (e.g. one unusually large TA might potentially fill a TEE storage facility even though it may normally host many normal TAs). If one of these properties contains the value UINT32_MAX, then the TEE has no fixed maximum for that property but (except where dynamic resources are exceeded) will support a minimum value that will be defined for a given TEE TMF configuration. The minimum requirements for GlobalPlatform TMF configurations will be defined in a separate future document.

A.6 Specification Return Codes

Table A-5: Specification Return Codes

Return Code	Value
TEE_ERROR_LIMIT_EXCEEDED	0xF0270001

A.7 Specification Return Code Origins

Table A-6: Specification Return Code Origins

Constant Name	Constant Value
TEEC_ORIGIN_TRUSTED_SD, TEE_ORIGIN_TRUSTED_SD	0x00000005

A.8 ASN.1 Syntax of the TEE Management Framework

```
TEEManagementFrameworkModule-v1000  DEFINITIONS IMPLICIT TAGS ::=
BEGIN

--
-- Some useful types and values
--
OneTo255Integer ::= INTEGER (1..255)
OneTo127Integer ::= INTEGER (1..127)
ZeroTo255Integer ::= INTEGER (0..255)
ZeroTo127Integer ::= INTEGER (0..127)

TMFversion ::= INTEGER { gpd-tee-tmf-version-v1000 (16777216) } -- The TEE Management Framework
versions with the current named version 1.0.0.0 encoded as the hexadecimal value 0x01000000

--
-- Common types definitions section 8.3
--

Attribute ::= [APPLICATION 2] SEQUENCE {  -- section 8.3.3.1
        attributID        INTEGER,
        content           CHOICE {
                              reference          OCTET STRING,
                              value              SEQUENCE {
                                                         a        INTEGER,
                                                         b        INTEGER
                                                 }
                          }
        }

UUID ::= [APPLICATION 3] OCTET STRING  -- section 8.3.3.2

ObjectId ::= [APPLICATION 4] OCTET STRING (SIZE(0..64))   -- section 8.3.3.3

CryptoOperationParameters ::= [APPLICATION 5] SEQUENCE {  -- section 8.3.3.4
        algorithmID       INTEGER,
        operationMode     INTEGER,
        algoParams        CHOICE {
                                  iv              OCTET STRING,
                                  attrValue       Attribute,
                                  aeValue         SEQUENCE {
        nonce                     OCTET STRING,
        tag               [0]     OCTET STRING           OPTIONAL,
        tagLen            [1]     INTEGER                OPTIONAL,
        aad               [2]     OCTET STRING           OPTIONAL,
        aadLen            [3]     INTEGER                OPTIONAL,
        payloadLen        [4]     INTEGER                OPTIONAL
                                                  }
                          }       OPTIONAL
}

KeyRefParameters ::= [APPLICATION 6] SEQUENCE {  -- section 8.3.3.5
        keyID             ObjectId,
        keyID2            ObjectId           OPTIONAL,
        cryptoParams      CryptoOperationParameters
}
```

```
StoredDataObject ::= [APPLICATION 7] SEQUENCE {  -- section 8.3.3.6
          objId                  ObjectId,
          objType                INTEGER,
          accessAndShareRights   INTEGER,
          attributes             SEQUENCE OF Attribute        OPTIONAL,
          datastream             OCTET STRING                 OPTIONAL,
          metadata               [0] SEQUENCE {
                                               sizeInBits     INTEGER,
                                               usageFlags     INTEGER
                                 }        OPTIONAL
}

UUIDV5Params ::= SEQUENCE {  -- section 8.3.3.7
          keyType                INTEGER,
          keySize                INTEGER,
          keyAttributes          SEQUENCE OF Attribute,
          signatureParams        CryptoOperationParameters,
          signature              OCTET STRING
}

UUIDVerificationParams ::= [APPLICATION 8] SEQUENCE {  -- section 8.3.3.7
          protocol        UUID,
          version         INTEGER,
          parameters      CHOICE {
                                 uuidV5Params [0] UUIDV5Params,
                                          -- for the protocol corresponding to the verification of UUID v5
                                 . . .    -- for future extensions
                                 }
}

CryptographicData ::= [APPLICATION 9] SEQUENCE {  -- section 8.3.3.8
          cryptoProcID           INTEGER,
          cryptoData             OCTET STRING -- an ‘open’ type as mentioned in section 7.1
}

Property ::= [APPLICATION 10] SEQUENCE {  -- section 8.3.3.9
          name            UTF8String,
          value           CHOICE {
                                 boolean         BOOLEAN,
                                 integer         INTEGER,
                                 string          UTF8String,
                                 binary          OCTET STRING,
                                 uuid            UUID,
                                 identity        SEQUENCE {
                                                          loginMethod     INTEGER,
                                                          uuid            UUID
                                                 }
                          }
}

--
-- SD Privileges : Types and values defined by this specification document
--

gpd-privilege-teeManagement              INTEGER ::= 64

gpd-privilege-sdManagement               INTEGER ::= 65

gpd-privilege-sdPersonalization  INTEGER ::= 66
```

```
gpd-privilege-taManagement              INTEGER ::= 67

gpd-privilege-taPersonalization  INTEGER ::= 68

gpd-privilege-rsdManagement             INTEGER ::= 69

-- Possible standard integer values extendable (using the extension marker "…") with any RFU or vendor-
specific values in range [1..255]

PrivilegeIDType ::= OneTo255Integer (gpd-privilege-teeManagement  | gpd-privilege-sdManagement  |
gpd-privilege-sdPersonalization  | gpd-privilege-taManagement  | gpd-privilege-taPersonalization  | gpd-
privilege-rsdManagement  , …) -- may support extensions

Privilege ::= SEQUENCE {
                                        privilegeID             PrivilegeIDType,
                                        privilegeParams         OCTET STRING  OPTIONAL
                                                                   -- an 'open' type as mentioned in section 7.1
}

SDPrivileges ::= [APPLICATION 27] SEQUENCE { -- section 8.3.3.10
          listOfPrivileges              SEQUENCE OF Privilege,
          isRootSD                      BOOLEAN(TRUE)   OPTIONAL
}


Authority ::= [APPLICATION 28] SEQUENCE {  -- section 8.3.3.11
          name                UTF8String,
          urlInfo             UTF8String      OPTIONAL
}


--
-- Audit information types definitions (Chapter 9)
--
SecureLayerAuditInfo ::= [APPLICATION 29] SEQUENCE {  -- section 9.1.1
          protocol            UUID,
          protocolInfo        OCTET STRING           OPTIONAL -- an 'open' type as mentioned in section 7.1
}

Option ::= [APPLICATION 12] SEQUENCE {  -- section 9.1.2
          name                UTF8String,
          version             INTEGER  -- section A.4
}

Device ::= [APPLICATION 13] SEQUENCE {  -- section 9.1.3
          name                          UTF8String,
          id                            UUID                    OPTIONAL,
          manufacturer                  UTF8String,
          firmwareVersion               PrintableString,
          type                          UTF8String              OPTIONAL
}

ISA ::= [APPLICATION 14] SEQUENCE {  -- section 9.1.4
          name                          UTF8String,
          processorType                 UTF8String,
          instructionSet                PrintableString,
          addressSize                   INTEGER,
          abi                           PrintableString,
          endianness              INTEGER { little(0), big(1), middle(2) }
}

TrustedOS ::= [APPLICATION 15] SEQUENCE {  -- section 9.1.5
          name            UTF8String,
          manufacturer    UTF8String,
          version         PrintableString,
          isaSet          SEQUENCE OF ISA,
          options         [0] SEQUENCE OF Option        OPTIONAL,
          protocols       [1] SEQUENCE OF SecureLayerAuditInfo  OPTIONAL
}

Tee ::= [APPLICATION 16] SEQUENCE {  -- section 9.1.6
          device                              Device,
          trustedOs                           TrustedOS,
          state                               INTEGER {locked(0), secure(1)},
          roots                               SEQUENCE OF UUID,
          optionalApis                        [0] SEQUENCE OF Option        OPTIONAL,
          teeImplementationProperties         [1] SEQUENCE OF Property      OPTIONAL,
          teePlatformLabel                    UTF8String
}

-- SD Lifecycle encoding, section 9.2.1

sdBlockedState INTEGER ::= 0
sdActiveState INTEGER ::= 1
sdRestrictedState INTEGER ::= 2
SDLifecycleState ::= [APPLICATION 17] ZeroTo127Integer (sdBlockedState | sdActiveState |
sdRestrictedState, … ) -- the extension marker indicates that other values (RFU or vendor-specific) are
allowed

SecurityDomain ::= [APPLICATION 18] SEQUENCE {  -- section 9.2.2
          id                      UUID,
          parent                  UUID                    OPTIONAL,
          lifecycleState          SDLifecycleState,
          authority               Authority               OPTIONAL,
          privileges              SDPrivileges            OPTIONAL,
          subdomains              [0] SEQUENCE OF UUID          OPTIONAL,
          protocols               [1] SEQUENCE OF SecureLayerAuditInfo  OPTIONAL
}

-- TA Lifecycle encoding, Section 9.3.1

taInactiveState INTEGER ::= 0
taExecutableState INTEGER ::= 1
taLockedState INTEGER ::= 2

TALifecycleState ::= [APPLICATION 19] ZeroTo127Integer (taInactiveState | taExecutableState |
taLockedState, … )              -- the extension marker indicates that other values
                                        -- (RFU or vendor-specific) are allowed

TrustedApplication ::= [APPLICATION 20] SEQUENCE {
          id                      UUID,
          parent                  UUID,
          lifecycleState          TALifecycleState,
          version                 PrintableString
}
```

```
--
-- Authorization Token types definitions (Chapter 10)
--
ConstraintParamsDigest ::= [PRIVATE 0] SEQUENCE {  -- section 10.1.2
        algorithmID      INTEGER,
        bitmap           INTEGER,
        digest           OCTET STRING
}
ConstraintDeviceId ::= [PRIVATE 1] UUID
ConstraintModelId ::= [PRIVATE 2] UUID
ConstraintMinVersion ::= [PRIVATE 3] INTEGER
ConstraintMaxVersion ::= [PRIVATE 4] INTEGER

TokenConstraint ::= CHOICE {   -- section 10.1.1: only the constraints defined by this specification
        device     ConstraintDeviceId,
        model      ConstraintModelId,
        minVer     ConstraintMinVersion,
        maxVer     ConstraintMaxVersion,
        params     ConstraintParamsDigest,
        . . .    --  constraint extensions may be defined after this marker
}

AuthorizationTokenPayload ::= [APPLICATION 21] SEQUENCE {  -- section 10.1.3
        version                 TMFversion DEFAULT gpd-tee-tmf-version-v1000,  -- section A.4
        authorizingSd           UUID,
        constraintsList         SEQUENCE OF TokenConstraint,
        signatureInfo           KeyRefParameters
}

AuthorizationToken ::= [APPLICATION 22] SEQUENCE {  -- section 10.1.4
        payload                 AuthorizationTokenPayload,
        signature               OCTET STRING
}

--
--  Administration command types definitions for Trusted Applications (section 8.3.4)
--
InstallTA ::= [APPLICATION 65] SEQUENCE {  -- section 8.4.1.1
        ta                              UUID,
        targetSD                        UUID,
        initialState                    TALifecycleState,
        applicationFile                 OCTET STRING,
        encryptionParams                CHOICE {
                                                param5          KeyRefParameters,
                                                null            NULL
                                        },
        idVerificationParams            CHOICE {!
                                                param6          UUIDVerificationParams,
                                                null            NULL
                                        }
}

UninstallTA ::= [APPLICATION 66] SEQUENCE {  -- section 8.4.2.1
        ta                      UUID
}

UpdateTA ::= [APPLICATION 67] SEQUENCE {  -- section 8.4.3.1
        ta                              UUID,
        newState                    TALifecycleState,
        applicationFile             OCTET STRING,
        encryptionParams            CHOICE {
                                            param4          KeyRefParameters,
                                            null            NULL
                                    },
        idVerificationParams        CHOICE {
                                            param5          UUIDVerificationParams,
                                            null            NULL
                                    }
}

LockTA ::= [APPLICATION 68] SEQUENCE {  -- section 8.4.4.1
        ta              UUID
}

UnlockTA ::= [APPLICATION 69] SEQUENCE {  -- section 8.4.5.1
        ta              UUID
}

--
--  Administration command types definitions for Security Domains (section 8.5)
--
InstallSD ::= [APPLICATION 74] SEQUENCE {  -- section 8.5.1.1
        sd                      UUID,
        targetSD                UUID,
        initialState            SDLifecycleState,
        privileges              SDPrivileges,
        authority               CHOICE {
                                            param5          Authority,
                                            null            NULL
                                    },
        cryptographicData       CHOICE {
                                            param6          CryptographicData,
                                            null            NULL
                                    }, !
        idVerificationParams    CHOICE {
                                            param7          UUIDVerificationParams,
                                            null            NULL
                                    }
}

InstallSDResp ::= CryptographicData                     -- section 8.5.1.2


UninstallSD ::= [APPLICATION 75] SEQUENCE {  -- section 8.5.2.1
        sd                      UUID,
        recursive               BOOLEAN
}

BlockSD ::= [APPLICATION 77] SEQUENCE {  -- section 8.5.3.1
        sd                      UUID,
        lockFlag                BOOLEAN
}

UnblockSD ::= [APPLICATION 78] SEQUENCE {  -- section 8.5.4.1
        sd              UUID
}
```

```
RestrictSD ::= [APPLICATION 79] SEQUENCE {  -- section 8.5.5.1
        sd              UUID
}

UnrestrictSD ::= [APPLICATION 80] SEQUENCE {  -- section 8.5.6.1
        sd              UUID
}

--
--  Administration command types definitions common to Security Domains and
--  Trusted Applications (section 8.6)
--
StoreData ::= [APPLICATION 85] SEQUENCE {  -- section 8.6.1.1
        taORsd                  UUID,
        decryptionParams        CHOICE {
                                        param2          KeyRefParameters,
                                        null            NULL
                                },
        storedDataObject        CHOICE {
                                        cipheredText    OCTET STRING,
                                        clearText       StoredDataObject
                                }
}

DeleteData ::= [APPLICATION 86] SEQUENCE {  -- section 8.6.2.1
        taORsd                  UUID,
        objId                   ObjectId
}

ListObjects ::= [APPLICATION 87] SEQUENCE {  -- section 8.6.3.1
        taORsd                  UUID
}

ListObjectsResp ::= [APPLICATION 25] SEQUENCE OF ObjectId  -- section 8.6.3.2

--
--  Administration command types definitions for TEE management (section 8.7)
--
LockTEE ::= [APPLICATION 90] SEQUENCE {}  -- section 8.7.1.1

UnlockTEE ::= [APPLICATION 91] SEQUENCE {}  -- section 8.7.2.1

StoreTEEProperty ::= [APPLICATION 92] SEQUENCE {  -- section 8.7.3.1
        property        Property
}

FactoryReset ::= [APPLICATION 93] SEQUENCE {}  -- section 8.7.4.1

--  Audit administration command types definitions (section 8.8)

GetTEEDef ::= [APPLICATION 97] SEQUENCE {}  -- section 8.8.1.1

GetTEEDefResp ::= Tee                           -- section 8.8.1.2

GetSDDef ::= [APPLICATION 98] SEQUENCE {  -- section 8.8.2.1
        sd              UUID
}

GetSDDefResp ::= SecurityDomain                 -- section 8.8.2.2
```

```
GetListOfTA ::= [APPLICATION 99] SEQUENCE {  -- section 8.8.3.1
        sd                  UUID
}

GetListOfTAResp ::= [APPLICATION 26] SEQUENCE OF UUID    -- section 8.8.3.2

GetTADef ::= [APPLICATION 100] SEQUENCE {  -- section 8.8.5.1
        ta                  UUID
}

GetTADefResp ::= TrustedApplication              -- section 8.8.5.2



--
--   Main types for administration command request and response payloads (section 8.3)
--
CmdReqPayload ::= [APPLICATION 0] SEQUENCE {  -- section 8.3.1
        version             TMFversion DEFAULT gpd-tee-tmf-version-v1000,  -- see section A.4
        token               AuthorizationToken       OPTIONAL,
        command             CHOICE {
                                installTAcmd                    InstallTA,
                                uninstallTAcmd                  UninstallTA,
                                updateTAcmd                     UpdateTA,
                                lockTAcmd                       LockTA,
                                unlockTAcmd                     UnlockTA,
                                installSDcmd                    InstallSD,
                                uninstallSDcmd                  UninstallSD,
                                blockSDcmd                      BlockSD,
                                unblockSDcmd                    UnblockSD,
                                restrictSDcmd                   RestrictSD,
                                unrestrictSDcmd                 UnrestrictSD,
                                storeDatacmd                    StoreData,
                                deleteDatacmd                   DeleteData,
                                listObjectscmd                  ListObjects,
                                lockTEEcmd                      LockTEE,
                                unlockTEEcmd                    UnlockTEE,
                                storeTEEpropertycmd             StoreTEEProperty,
                                factoryResetcmd                 FactoryReset,
                                retrieveTEEdefcmd               GetTEEDef,
                                retrieveSDdefcmd                GetSDDef,
                                retrieveListOfTAcmd             GetListOfTA,
                                retrieveTADefcmd                GetTADef,
                                . . .    -- new commands may be defined after this marker
                            }
}

CmdRespPayload ::= [APPLICATION 1] SEQUENCE {  -- section 8.3.2
        returnCode          INTEGER,
        response            CHOICE {
                                installSDresp                   InstallSDResp,
                                listObjectsresp                 ListObjectsResp,
                                retrieveTEEdefresp              GetTEEDefResp,
                                retrieveSDdefresp               GetSDDefResp,
                                retrieveListOfTAresp            GetListOfTAResp,
                                retrieveTAdefresp               GetTADefResp,
                                . . .    --  new command responses may be defined after this marker
                            }               OPTIONAL
}
```

```
--
-- Security Layer types definition (section 8.2)
--
ContainerType ::= OneTo255Integer -- see Table 8-5


ContainerContent ::= SEQUENCE {
        type            ContainerType,
        header          OCTET STRING OPTIONAL,  -- an ‘open’ type as mentioned in section 7.1
        payload         CHOICE {
                                anyData                 [0] OCTET STRING,
                                cmdReqPayload           CmdReqPayload,
                                cmdRespPayload          CmdRespPayload
                                }
}

admin-generic-cont-type ContainerType ::= 1  -- the type value of the generic container

GenericContainerType ::= ContainerType (admin-generic-cont-type)


GenericContainerContent ::= ContainerContent
        ( WITH COMPONENTS {
                type    (GenericContainerType),
                header ABSENT ,
                payload
        })

SecurityContainer ::=  [APPLICATION 23] SEQUENCE {
        version         TMFversion DEFAULT gpd-tee-tmf-version-v1000,  -- section A.4
        content         ContainerContent
}

END
```

A.9 Specification Object Identifiers

Table A-7: Specification Object Identifiers

Object	Object Identifier value	Description
SD Authority information	`0x5344417574686f72697479496e666f73`	Identifies an SD Authority information object. This object SHOULD be stored in the SD private storage (see section 5.5). The object data value SHOULD be encoded as the DER-encoded value of an Authority type (see section 8.3.3.11).

A.10 Required Cryptographic Algorithms

Table A-8 lists the algorithms that can be used depending on the cryptographic operation contexts.

For each operation context, a GlobalPlatform ASN.1 TMF compliant implementation SHALL support at least one of the algorithms shown in bold characters and marked "M" for mandatory.

To support any required specific use cases, an implementation MAY also support any algorithm marked "O" for **O**ptional or any vendor-specific algorithm not listed in this table.

This is why the audit command exposing the TEE characteristics (see section 9.1) about a TEE vendor identifier will help to inform about the usage of cryptographic algorithms supported by an implementation of this specification.

Table A-8: Mandatory and Optional Cryptographic Algorithms

Cryptographic Operation context	Algorithms	Mandatory/ Optional
Authorization Token Signature/Verification (Chapter 10 and section 5.3.3)	**Asymmetric Algorithms**[2]	
	TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA256	**M**
	TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA384 TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA512 TEE_ALG_RSASSA_PKCS1_V1_5_SHA256	O
	TEE_ALG_DSA_SHA256 with 2048-bit key	**M**
	TEE_ALG_DSA_SHA256 with 3072-bit key	O
	TEE_ALG_ECDSA_SHA512 TEE_ALG_ECDSA_SHA384 TEE_ALG_ECDSA_SHA256	O
	Symmetric Algorithms	
	TEE_ALG_HMAC_SHA256	**M**
	TEE_ALG_HMAC_SHA384 TEE_ALG_HMAC_SHA512	O
Application File encryption/decryption (sections 6.2.1, 6.2.3, 8.4.1.1, and 8.4.3.1)	**Symmetric Algorithms**	
	TEE_ALG_AES_CCM	**M**
	TEE_ALG_AES_CTR TEE_ALG_AES_CTS TEE_ALG_AES_XTS TEE_ALG_AES_GCM	O

[2] The asymmetric algorithms shown as changed in v1.0.1 of this specification were deprecated in [TEE Core API] in lieu of the new names shown.

Cryptographic Operation context	Algorithms	Mandatory/ Optional

`StoreData` command confidentiality
(sections 6.4.1 and 8.6.1)
	The symmetric algorithms listed above in the context of Application File encryption/decryption operations, plus
	`TEE_ALG_RSAES_PKCS1_V1_5`
	`TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA1`
	`TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA224`
	`TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA256`
	`TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA384`
	`TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA512`
	`TEE_ALG_RSA_NOPAD`
		M

	`TEE_ALG_SM2_PKE` (if supported)	O
UUID v5 Signature/ Verification (section 5.6)	Only asymmetric algorithms as listed in the context of Authorization Token signature/verification operations (see above).	
Command parameters constraint digest (sections 5.3.2, 5.3.3, 10.1.1, and 10.1.2)	**`TEE_ALG_SHA256`**	**M**
	`TEE_ALG_SHA384` `TEE_ALG_SHA512`	O

The following table provides the necessary parameters that can be required when performing a cryptographic operation using one of the following algorithms.

Table A-9: Algorithm Parameters

Algorithm	Parameters

`TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA256`
`TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA384`
`TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA512`
	The salt length value

`TEE_ALG_AES_CTR`
`TEE_ALG_AES_CTS`
`TEE_ALG_AES_CBC_NOPAD`
	An optional nonce value (Initial Vector)

`TEE_ALG_AES_XTS`	A random value (aka the initial 'tweak' value)
`TEE_ALG_AES_CCM`	A nonce value, an authentication tag length, the payload length, and additional authentication data
`TEE_ALG_AES_GCM`	A nonce value and an authentication tag length

The only recommendation regarding the strength of the keys used in a particular cryptographic operation context is the adoption of the best practices associated with the choice of such algorithms at the time of a specific implementation. Today, this specification mandates at least 2048 bits for DSA and strongly recommends the same strength for the RSA algorithms. Table A-10 provides the normative references links where these best practices can be found.

> If asymmetric encryption is used, the payload must be shorter than the modulus – asymmetric encryption is therefore normally used to encrypt a symmetric key.

Backward Compatibility

In version 1.0 only symmetric algorithms were specified as suitable for the `StoreData` command even though the TEE Core API supports asymmetric encryption with RSA and SM2.

Table A-10: Normative References for Algorithms

Name	References	URL

`TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA256`
`TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA384`
`TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA512`
`TEE_ALG_RSASSA_PKCS1_V1_5_SHA256`
`TEE_ALG_RSAES_PKCS1_V1_5`
`TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA1`
`TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA224`
`TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA256`
`TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA384`
`TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA512`
`TEE_ALG_RSA_NOPAD`
	PKCS #1 (RSA, PKCS1 v1.5, PSS)
	FIPS 180-4
		ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf
		http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf

`TEE_ALG_DSA_SHA256`	FIPS 180-4 FIPS 186-2 (DSA)	http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf http://csrc.nist.gov/publications/fips/archive/fips186-2/fips186-2.pdf

`TEE_ALG_ECDSA_P521`
`TEE_ALG_ECDSA_P384`
`TEE_ALG_ECDSA_P256`
	FIPS 186-4
	ANSI X9.62
		http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf
		http://webstore.ansi.org/RecordDetail.aspx?sku=ANSI+X9.62%3A2005

`TEE_ALG_HMAC_SHA256`
`TEE_ALG_HMAC_SHA384`
`TEE_ALG_HMAC_SHA512`
	RFC 4231	http://tools.ietf.org/html/rfc4231

`TEE_ALG_AES_CCM`	FIPS 197 (AES) RFC 3610 (CCM)	http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf http://tools.ietf.org/html/rfc3610

Name	References	URL
TEE_ALG_AES_CTR	FIPS 197 (AES) NIST SP800-38A (ECB, CBC, CTR)	http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf
TEE_ALG_AES_CTS	FIPS 197 (AES) NIST SP800-38A Addendum (CTS = CBC-CS3)	http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf http://csrc.nist.gov/publications/nistpubs/800-38a/addendum-to-nist_sp800-8A.pdf
TEE_ALG_AES_XTS	IEEE Std 1619-2007	http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4493431
TEE_ALG_AES_GCM	FIPS 197 (AES) NIST 800-38D (GCM)	http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf
TEE_ALG_SH256 TEE_ALG_SHA384 TEE_ALG_SHA512	FIPS 180-4	http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf

Annex B Examples (Informative)

B.1 Security Domain Associations

The following set of examples is illustrative of some of the many possible configurations achievable using the TMF functionality. The figures generally show a simple arrangement to show one management structure that can be achieved, and further SDs, rSDs, and TAs can potentially be added in the context of each example. Architectures based on combinations of the examples are generally possible, though in some cases rSD creation rules may restrict this.

The diagrams indicate a "*created by*" relationship between SDs. This reflects an SD parent that meets the restricted capability rules set out in section 4.1.3.3.

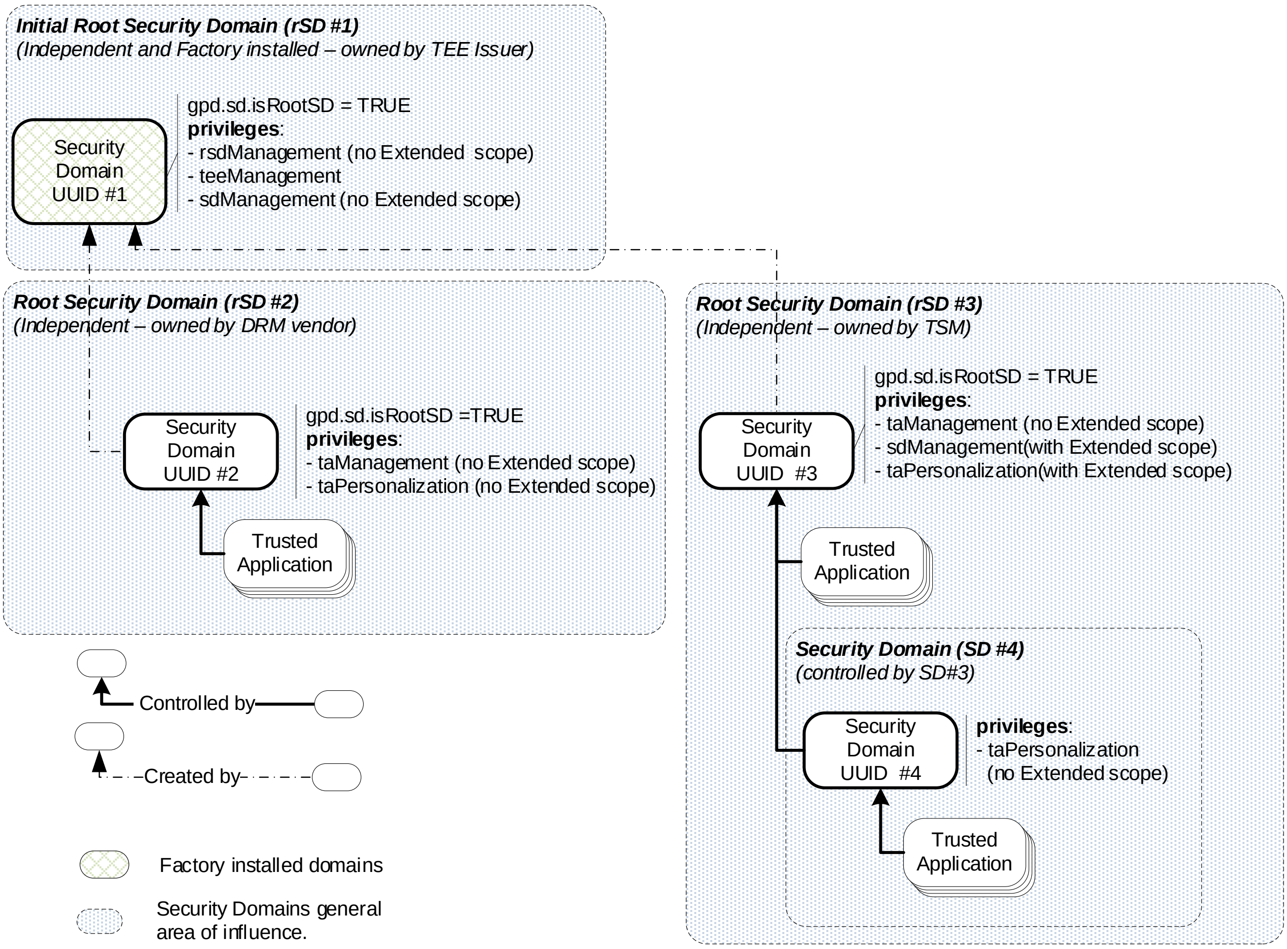
B.1.1 Security Domain Associations – Single Initial Domain Example

Figure B-1 illustrates an example of Security Domain associations and configurations:

- The TEE issuer has an initial root Security Domain (rSD #1)
    - The domain is installed and initially personalized in the factory.
    - No entity other than the owner has any control over it. This limitation is what enables rSD#1 to claim to also be a root Security Domain.
    - rSD#1's owner has limited ability to authorize management commands creating direct child Security Domains and to manage the TEE life cycle.
    - No Trusted Application can be deployed in this domain with the domains assigned privileges.
    - Because of rSD#1s having the `gpd.privilege.rsdManagement` privilege, it is allowed to create any direct Security Domain qualified as Root Security Domains.
- Another Security Domain (rSD#2) exists as a root Security Domain.
    - This domain is installed in the field by rSD#1, but rSD#1 has a strictly limited set of control and so cannot later interfere with SD#2 or its children. This limitation is what enables rSD#2 to claim to also be a root Security Domain.
    - rSD#2's owner is able to authorize management commands to manage and personalize its directly controlled set of Trusted Applications.
    - The Trusted Applications in this Security Domain are only controlled by commands authorized by the owner of rSD#2 and no other SD.
    - rSD#2's owner can neither change rSD#2's initial settings nor create further Security Domains.
    - rSD#2's owner is not able to manage the TEE life cycle.
- Another Security Domain (rSD#3) also exists as a root Security Domain.
    - This domain is installed in the field by rSD#1, but rSD#1 has a strictly limited set of control and so cannot interfere with SD#3 or its children. This limitation is what enables rSD#3 to claim to also be a root Security Domain.
    - rSD#3's owner is able to authorize management commands for managing and personalizing its own set of Trusted Applications, but not those of its direct or indirect child Security Domains.
    - rSD#3's owner is able to authorize management commands for creation and control of direct and indirect child Security Domains in this tree.
    - Neither rSD#3 nor its direct or indirect children are able to manage the TEE life cycle.

  - The Trusted Applications in rSD#3 are not subject to commands authorized by any domain other than rSD#3.
  - Because of its lack of `gpd.privilege.rsdManagement` privilege, no Security Domain it creates will qualify as Root Security Domains.
- Finally, a Security Domain (SD#4) has been created in the field as a sub-domain of rSD#3.
  - SD#4 itself may only be managed by commands authorized by the owner of rSD#3.
  - SD#4's owner cannot authorize commands to create further Security Domains.
  - SD#4's owner cannot authorize commands to manage the TEE.
  - Management of TAs in this SD can only be authorized by the owner of rSD#3.
  - Personalization of TAs in this SD can only be authorized by the owner of rSD#3 or SD#4.

Figure B-1: Example of Security Domain Associations – Single Initial Domain


Initial Root Security Domain (rSD #1)
(Independent and Factory installed – owned by TEE Issuer)
Security Domain UUID #1
gpd.sd.isRootSD = TRUE
privileges:
- rsdManagement (no Extended scope)
- teeManagement
- sdManagement (no Extended scope)
Root Security Domain (rSD #2)
(Independent – owned by DRM vendor)
Security Domain UUID #2
gpd.sd.isRootSD =TRUE
privileges:
- taManagement (no Extended scope)
- taPersonalization (no Extended scope)
Trusted Application
Root Security Domain (rSD #3)
(Independent – owned by TSM)
Security Domain UUID #3
gpd.sd.isRootSD = TRUE
privileges:
- taManagement (no Extended scope)
- sdManagement(with Extended scope)
- taPersonalization(with Extended scope)
Trusted Application
Security Domain (SD #4)
(controlled by SD#3)
Security Domain UUID #4
privileges:
- taPersonalization (no Extended scope)
Trusted Application
Controlled by
Created by
Factory installed domains
Security Domains general area of influence.

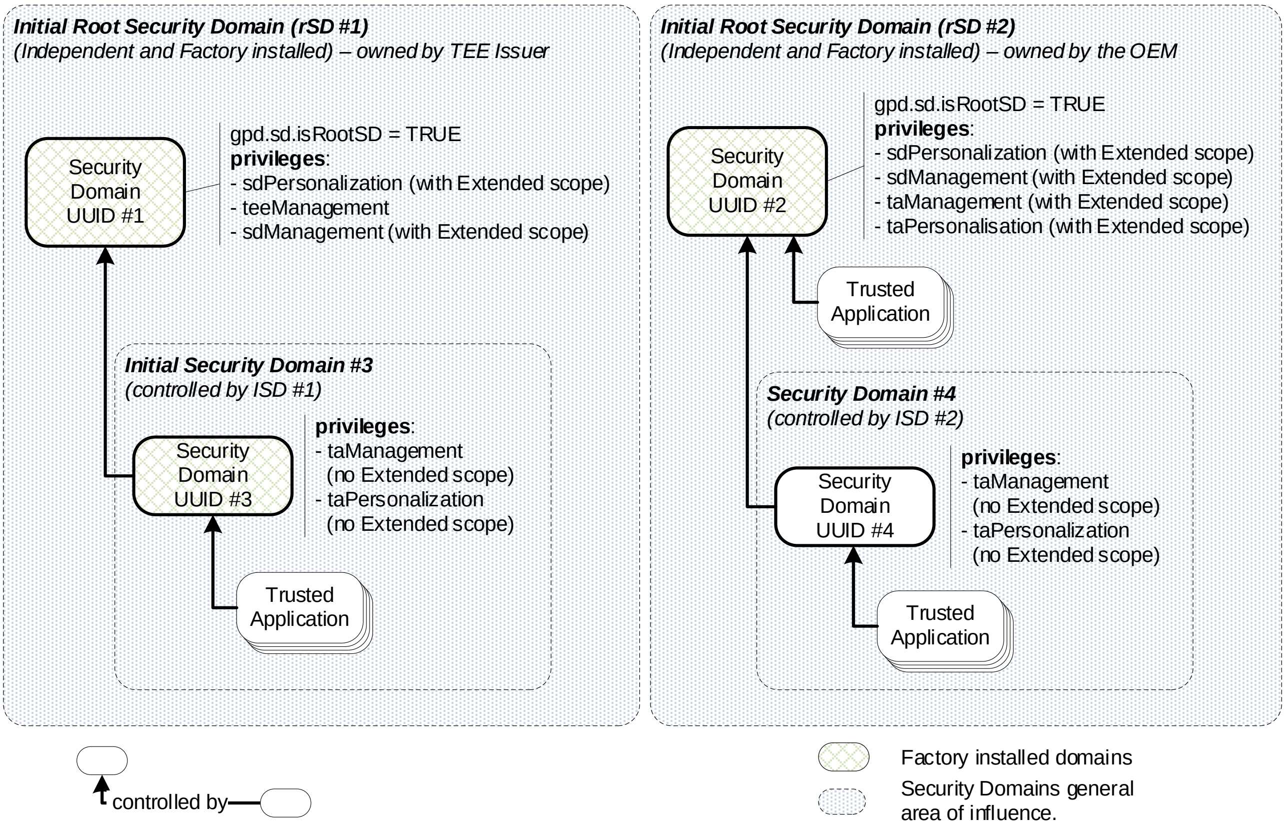
B.1.2 Security Domain Associations – Multiple Initial Domain Example

Figure B-2 illustrates an example of Security Domain associations and configurations:

- The TEE issuer has an initial root Security Domain (rSD #1)
  - The domain is installed and initially personalized in the factory.
  - Only the owner has any control over it. This limitation is what enables rSD#1 to claim to also be a root Security Domain.
  - rSD#1's owner is able to authorize management commands creating sub-domains and to manage the TEE life cycle.
  - No Trusted Application can be deployed in this domain with the domains assigned privileges.
  - Because of its lack of `gpd.privilege.rsdManagement` privilege, no Security Domain it creates will qualify as Root Security Domains.
- The OEM has an initial root Security Domain (rSD#2).
  - The domain is installed and initially personalized in the factory.
  - Only the owner has any control over it. This limitation is what enables rSD#2 to claim to also be a root Security Domain.
  - rSD#2's owner is able to authorize management commands to control its directly controlled set of Trusted Applications.
  - rSD#2's owner is able to authorize management commands creating sub-domains.
  - rSD#2's owner is not able to authorize TEE administration commands.
  - The Trusted Applications in this Security Domain are only controlled by commands authorized by rSD#2's owner and no other SD owner.
  - Because of its lack of `gpd.privilege.rsdManagement` privilege, no Security Domain it creates will qualify as Root Security Domains.
- Another Security Domain (SD#3) exists as a child of rSD#1.
  - The domain is installed and initially personalized in the factory.
  - SD#3's owner is able to authorize management commands for its own set of Trusted Applications.
  - SD#3's owner is not able to authorize commands to manage the TEE life cycle.
  - SD#3's owner is not able to authorize management commands to manage SD#3, or create and personalize child sub Security Domains of SD#3.
  - SD#1's owner is able to authorize management commands to manage SD#3, or create and personalize child sub Security Domains of SD#3.
  - The Trusted Applications in this Security Domain are not subject to commands authorized by any domain owners other than that of SD#3.

- Finally, a Security Domain (SD#4) exists as a child of rSD #2
    - This domain has been created in the field by rSD#2.
    - The owners of SD#4 and rSD#2 are both able to authorize management and personalization commands for SD#4's set of Trusted Applications.
    - SD#4's owner is not able to authorize management commands for creation of direct or indirect child Security Domains that might be created in the future by rSD#2.
    - SD#4's owner is not able to authorize management commands to manage the TEE life cycle.
    - The Trusted Applications in this Security Domain are not subject to commands authorized by owners of any domain other than SD#4 and rSD#2.

Figure B-2: Example of Security Domain Associations – Multiple Initial Domains


Initial Root Security Domain (rSD #1)
(Independent and Factory installed) – owned by TEE Issuer
Security Domain UUID #1
gpd.sd.isRootSD = TRUE
privileges:
- sdPersonalization (with Extended scope)
- teeManagement
- sdManagement (with Extended scope)
Initial Security Domain #3
(controlled by ISD #1)
Security Domain UUID #3
privileges:
- taManagement (no Extended scope)
- taPersonalization (no Extended scope)
Trusted Application
Initial Root Security Domain (rSD #2)
(Independent and Factory installed) – owned by the OEM
Security Domain UUID #2
gpd.sd.isRootSD = TRUE
privileges:
- sdPersonalization (with Extended scope)
- sdManagement (with Extended scope)
- taManagement (with Extended scope)
- taPersonalisation (with Extended scope)
Trusted Application
Security Domain #4
(controlled by ISD #2)
Security Domain UUID #4
privileges:
- taManagement (no Extended scope)
- taPersonalization (no Extended scope)
Trusted Application
controlled by
Factory installed domains
Security Domains general area of influence.

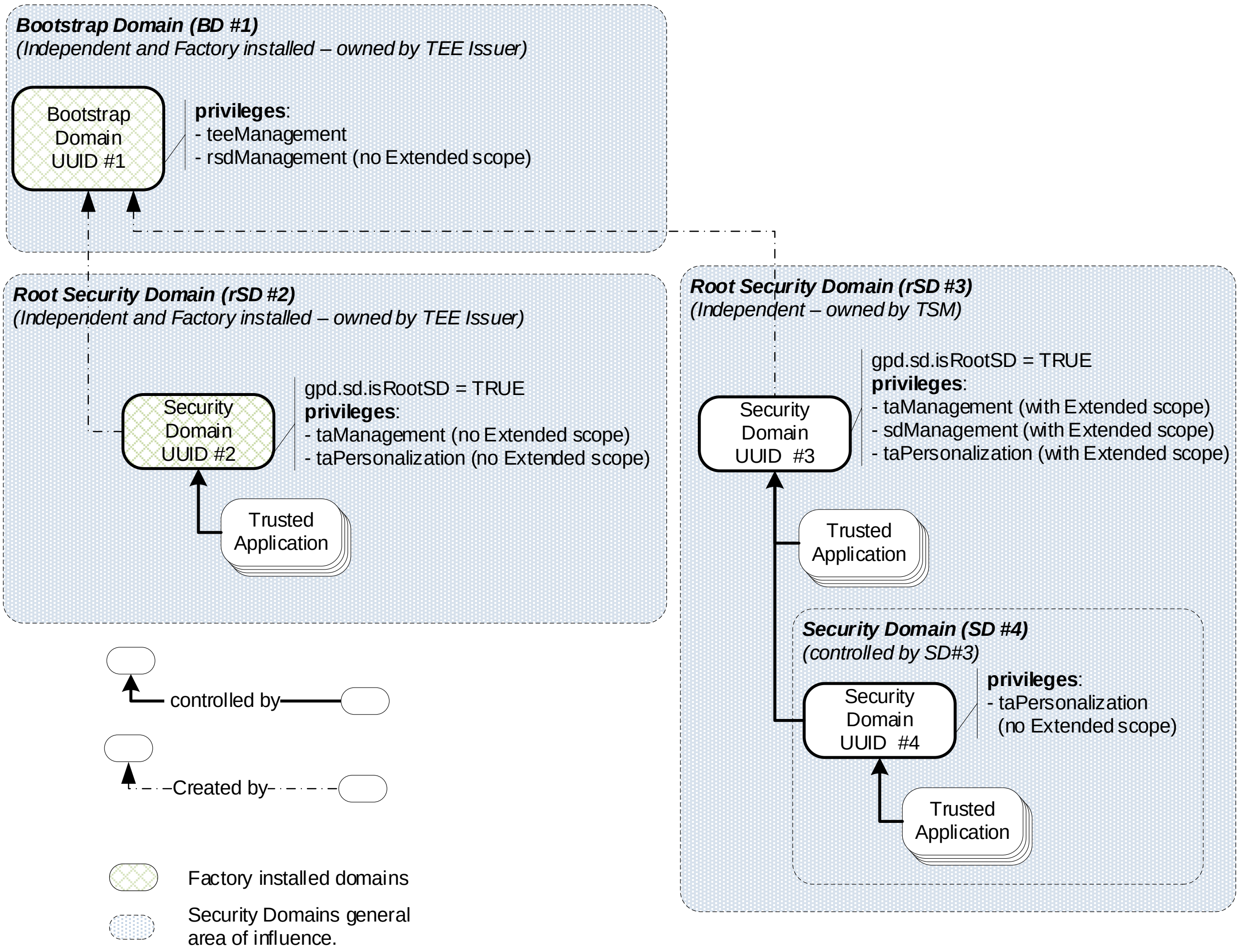
In this example, because of the rights of the existing root SDs, no domain shown is capable of creating a further root SD. While this figure shows two rSDs and one SD installed in the factory, there is no specified restriction on the number of factory installed SDs and rSDs. It is possible that one of these other factory installed rSDs may be restricted in such a manner as to enable it to create further rSDs in the field.

B.1.3 Security Domain Associations – Bootstrap Domain Example 1

Figure B-3 illustrates an example of Security Domain associations and configurations:

- The TEE issuer has an initial root Bootstrap Domain (BD #1)
  - The domain is installed and initially personalized in the factory and no entity has any control over BD#1. This limitation is what enables BD#1 to claim to also be a root Security Domain.
  - This domain does NOT use the GlobalPlatform TMF command protocols and so does not qualify as a GlobalPlatform domain.
  - While BD#1 does not use the GlobalPlatform TMF command protocols, its capabilities can be mapped on to the GlobalPlatform TMF privileges set.
  - BD#1's owner is able to authorize proprietary administration commands creating GlobalPlatform TMF compliant root sub-domains and has the ability to manage the TEE life cycle.
  - No GlobalPlatform Trusted Application can be deployed or personalized in this domain with the domains assigned privileges.
- Another Security Domain (rSD#2) exists as a root.
  - This domain is installed in the field by BD#1, but BD#1 has a strictly limited set of control and so cannot later interfere with rSD#2 or its children.
  - rSD#2's owner is able to authorize management commands to control its directly controlled set of Trusted Applications.
  - The Trusted Applications in this Security Domain are only controlled by commands authorized by rSD#2's owner and no other SD owner.
  - rSD#2 owner can neither change its initial settings nor create further Security Domains.
  - rSD#2 owner is not able to manage the TEE life cycle.
- Another Security Domain (rSD#3) also exists as a root.
  - This domain is installed in the field by BD#1, but BD#1 has a strictly limited set of control and so cannot interfere with rSD#3 or its children.
  - rSD#3's owner is able to authorize management commands for its own set of Trusted Applications and those of any of its child domains.
  - rSD#3's owner is able to authorize management commands for creation of child Security Domains in this tree.
  - Neither the owner of rSD#3 nor its child Security Domains are able to manage the TEE life cycle.
  - The Trusted Applications in this Security Domain are not subject to commands authorized by owners of any domain other than rSD#3.
  - Because of its lack of `gpd.privilege.rsdManagement` privilege, no Security Domain it creates will qualify as Root Security Domains.
- Finally, a Security Domain (SD#4) has been created in the field as a sub-domain of rSD#3,
  - SD#4 itself may only be managed by commands authorized by the owner of rSD#3.
  - SD#4's owner cannot authorize commands to create further domains.
  - SD#4's owner cannot authorize commands to manage the TEE.
  - Management of TAs in this domain can only be authorized by the owner of rSD#3.
  - Personalization of TAs in this domain can be authorized by the owners of rSD#3 or SD#4.

Figure B-3: Example of Security Domain Associations – Bootstrap Domain Example 1


Bootstrap Domain (BD #1)
(Independent and Factory installed – owned by TEE Issuer)
Bootstrap Domain UUID #1
privileges:
- teeManagement
- rsdManagement (no Extended scope)
Root Security Domain (rSD #2)
(Independent and Factory installed – owned by TEE Issuer)
Security Domain UUID #2
gpd.sd.isRootSD = TRUE
privileges:
- taManagement (no Extended scope)
- taPersonalization (no Extended scope)
Trusted Application
Root Security Domain (rSD #3)
(Independent – owned by TSM)
Security Domain UUID #3
gpd.sd.isRootSD = TRUE
privileges:
- taManagement (with Extended scope)
- sdManagement (with Extended scope)
- taPersonalization (with Extended scope)
Trusted Application
Security Domain (SD #4)
(controlled by SD#3)
Security Domain UUID #4
privileges:
- taPersonalization (no Extended scope)
Trusted Application
controlled by
Created by
Factory installed domains
Security Domains general area of influence.

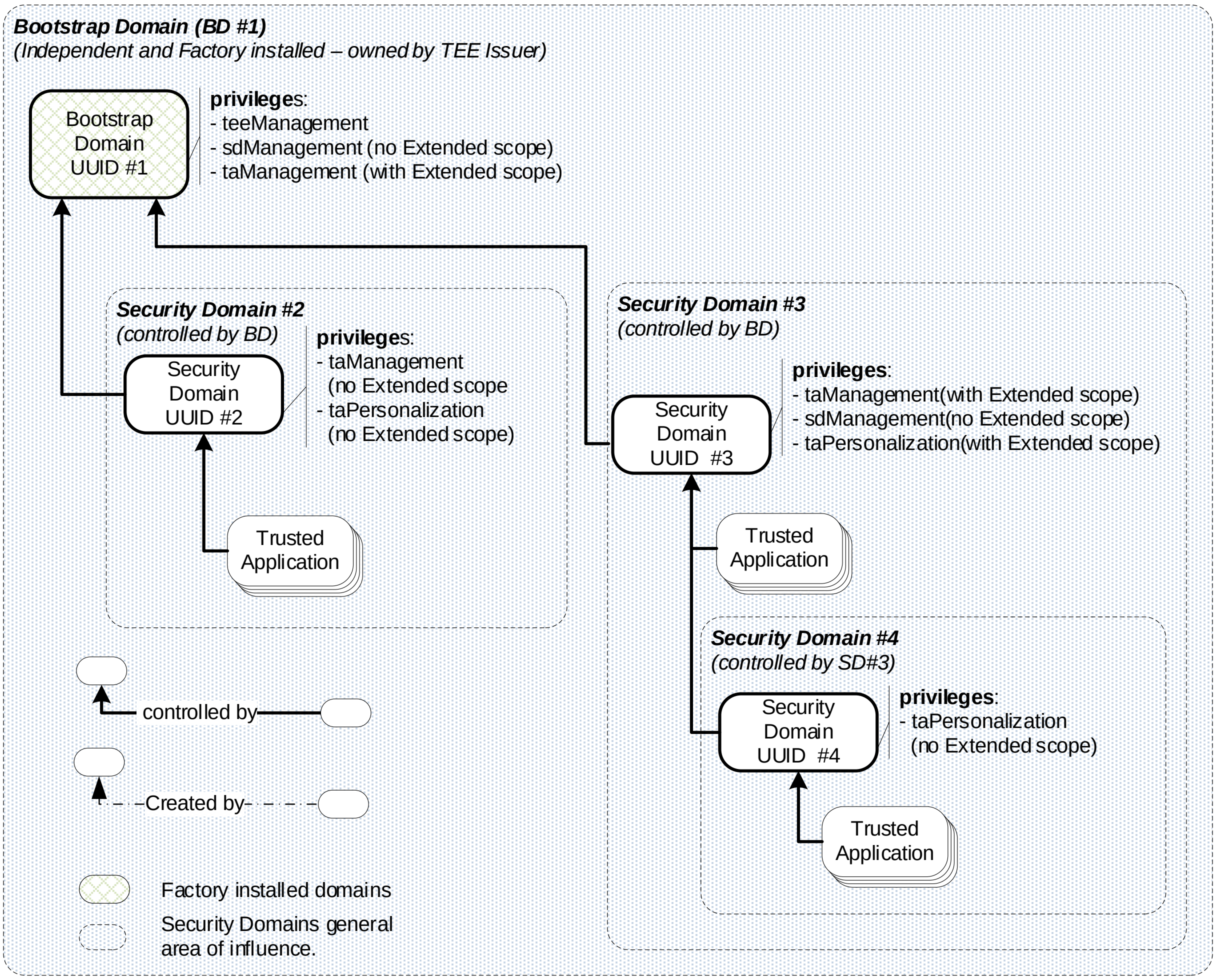
Note: The presence of a BD does not exclude the presence of factory installed SDs and rSDs.

B.1.4 Security Domain Associations – Bootstrap Domain Example 2

Figure B-4 illustrates an example of Security Domain associations and configurations:

- The TEE issuer has an initial root Bootstrap Domain (BD #1)
  - The domain is installed and initially personalized in the factory and no entity has any control over BD#1. This limitation is what enables BD#1 to claim to also be a root Security Domain.
  - This domain does NOT use the GlobalPlatform TMF command protocols and so does not qualify as a GlobalPlatform domain.
  - While BD#1 does not use the GlobalPlatform TMF command protocols, BD#1's capabilities can be mapped on to the GlobalPlatform TMF privileges set.
  - BD#1's owner is able to authorize proprietary administration commands creating GlobalPlatform TMF compliant sub-domains and has the ability to manage the TEE life cycle.
  - BD#1's owner is able to authorize proprietary administration commands capable of creating GlobalPlatform Trusted Applications that can be deployed in this domain.
  - BD#1's owner is not able to authorize TA personalization commands.
  - Because BD#1 lacks the `gpd.privilege.rsdManagement` privilege, no Security Domain created by BD#1 or its children will qualify as a Root Security Domain.
- Another Security Domain (SD#2) exists.
  - This domain is installed in the field by BD#1.
  - The Trusted Applications in this Security Domain are only managed by commands authorized by the owners of BD#1 and SD#2.
  - The Trusted Applications in this Security Domain are only personalized by commands authorized by the owners of SD#2.
  - SD#2 owner can neither change its initial settings nor create further Security Domains.
  - SD#2 owner is not able to manage the TEE life cycle.
- Another Security Domain (SD#3) also exists.
  - This domain is installed in the field by BD#1.
  - SD#3's owner able to authorize management commands for its own set of Trusted Applications or any of its child domains.
  - SD#3's owner is able to authorize management commands for creation of child Security Domains.
  - Neither the owner of SD#3 nor its child Security Domains are able to manage the TEE life cycle.
  - The Trusted Applications in this Security Domain are not subject to commands authorized by owner of BD#1.
- Finally, a Security Domain (SD#4) has been created in the field as a sub-domain of SD#3.
  - SD#4 itself may only be managed by commands authorized by the owners of BD#1 or SD#3.
  - SD#4's owner can authorize commands to create further domains.
  - SD#4's owner cannot authorize commands to manage the TEE.
  - Management of TAs in this domain can only be authorized by the owners of BD#1 or SD#3.
  - Personalization of TAs in this domain can be authorized by the owners of SD#3 or SD#4.

Figure B-4: Example of Security Domain Associations – Bootstrap Domain Example 2



Bootstrap Domain (BD #1)
(Independent and Factory installed – owned by TEE Issuer)
Bootstrap Domain UUID #1
privileges:
- teeManagement
- sdManagement (no Extended scope)
- taManagement (with Extended scope)
Security Domain #2
(controlled by BD)
Security Domain UUID #2
privileges:
- taManagement (no Extended scope
- taPersonalization (no Extended scope)
Trusted Application
Security Domain #3
(controlled by BD)
Security Domain UUID #3
privileges:
- taManagement(with Extended scope)
- sdManagement(no Extended scope)
- taPersonalization(with Extended scope)
Trusted Application
Security Domain #4
(controlled by SD#3)
Security Domain UUID #4
privileges:
- taPersonalization (no Extended scope)
Trusted Application
controlled by
Created by
Factory installed domains
Security Domains general area of influence.

B.1.5 Security Domain Associations – Further Examples

The previous examples are just a limited set of what may be created.

Some other ways a device design might affect the Security Domain structure include:

- A device design may only have ONE factory installed Security Domain, and no ability to add further Security Domains.
- A device design may restrict the numbers of Security Domains due to resource restrictions.
- A device design may restrict the numbers of TAs to all those Security Domain due to resource restrictions.
- A device design may restrict the numbers of TAs installable by a particular Security Domain due to resource restrictions.
- A Security Domain can theoretically have great depth, but again a resource restriction may limit the “depth” of the tree on a given device to far fewer than shown.

B.2 Section Moved

The content of this section has been moved to section 8.3.4, CryptoProcID, and made normative.

The former Tables B-1 through B-3 are now Table 8-20c through Table 8-20e.

B.3 Bootstrapping the Security Domain Keys

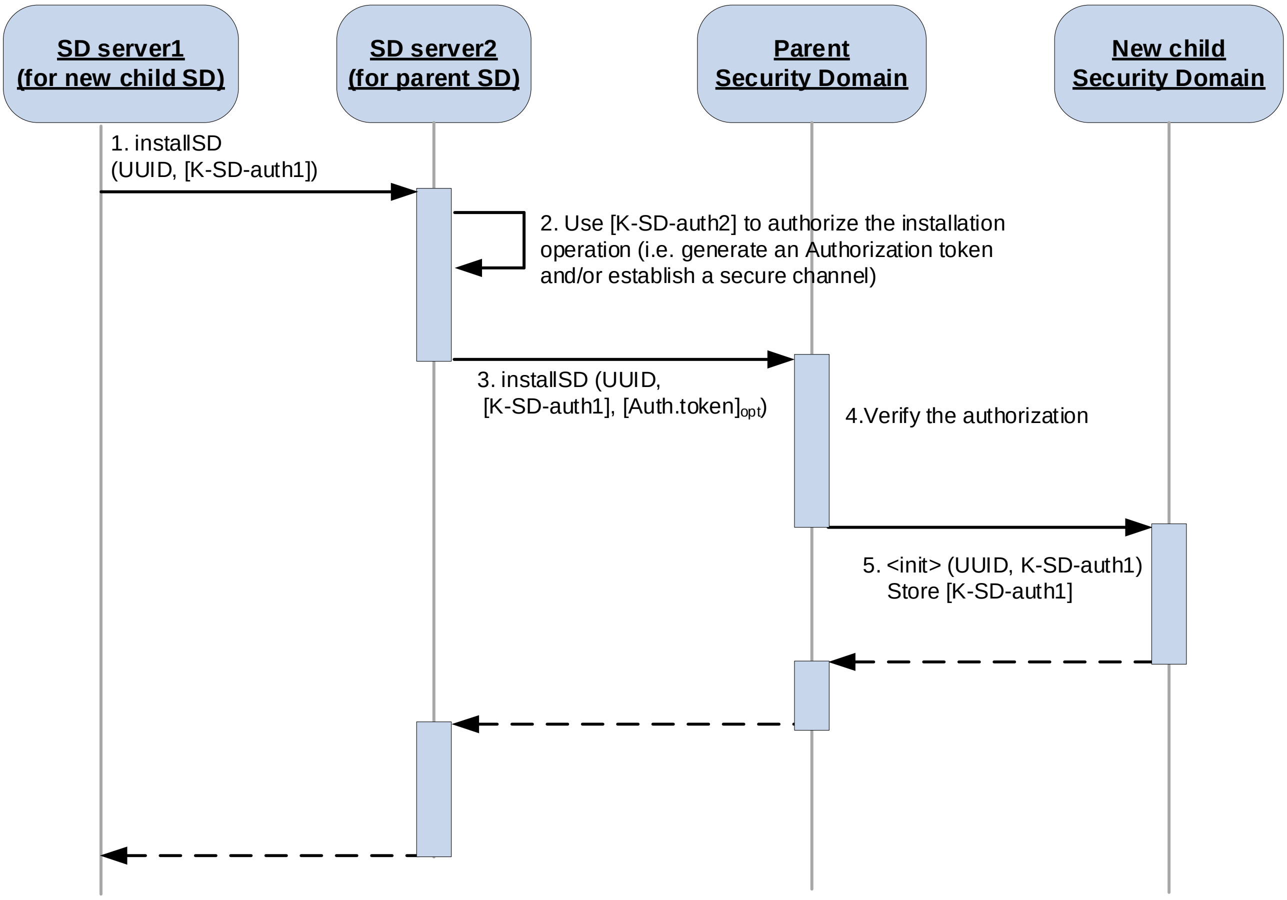
B.3.1 Initial Key Provisioning for Security Domains

When a new Security Domain is installed, it can be provisioned with at least one key to be used by this Security Domain to authenticate the issuer and/or to verify authorizations.

In the following figure, the key denoted [K-SD-auth2] is shared between the Security Domain (‘Parent Security Domain’) and its SD server (‘SD server2’) authorizing this operation. Such authorization can be ‘implicit’ (a secure channel is established) and/or ‘explicit’ (an Authorization Token is delivered with the installation command).

A first key, named [K-SD-auth1], is provisioned when the new Security Domain is installed, using a cryptographic procedure based on the example in section 8.3.4.1, as illustrated below.

Figure B-5: Initial Key Provisioning for a Security Domain


SD server1 (for new child SD)
SD server2 (for parent SD)
Parent Security Domain
New child Security Domain
1. installSD (UUID, [K-SD-auth1])
2. Use [K-SD-auth2] to authorize the installation operation (i.e. generate an Authorization token and/or establish a secure channel)
3. installSD (UUID, [K-SD-auth1], [Auth.token]opt)
4.Verify the authorization
5. <init> (UUID, K-SD-auth1) Store [K-SD-auth1]

1. The entity that wants to create a child Security Domain (‘New child SD’) sends a request to an Authority that already has a personalized Security Domain in the TEE (e.g. the parent Security Domain in this figure). The information provided is:
   - o The UUID of the SD to install
   - o The key [K-SD-auth1] that will be used by this new SD for authentication of the backend
2. The Authority receiving the request generates an Authorization and/or establishes a secure channel to the parent SD using the key material corresponding to the addressed parent Security Domain.
3. The Install SD command is forwarded to the corresponding Security Domain.

4. The authorization to perform the command is verified by this Security Domain (the command is protected by a Security Layer and/or the token is verified).
5. A new child Security Domain ('New child SD') is created and populated with a persistent key object initialized with the key material ([K-SD-auth1]) provided by SD server1 and given in the install command.

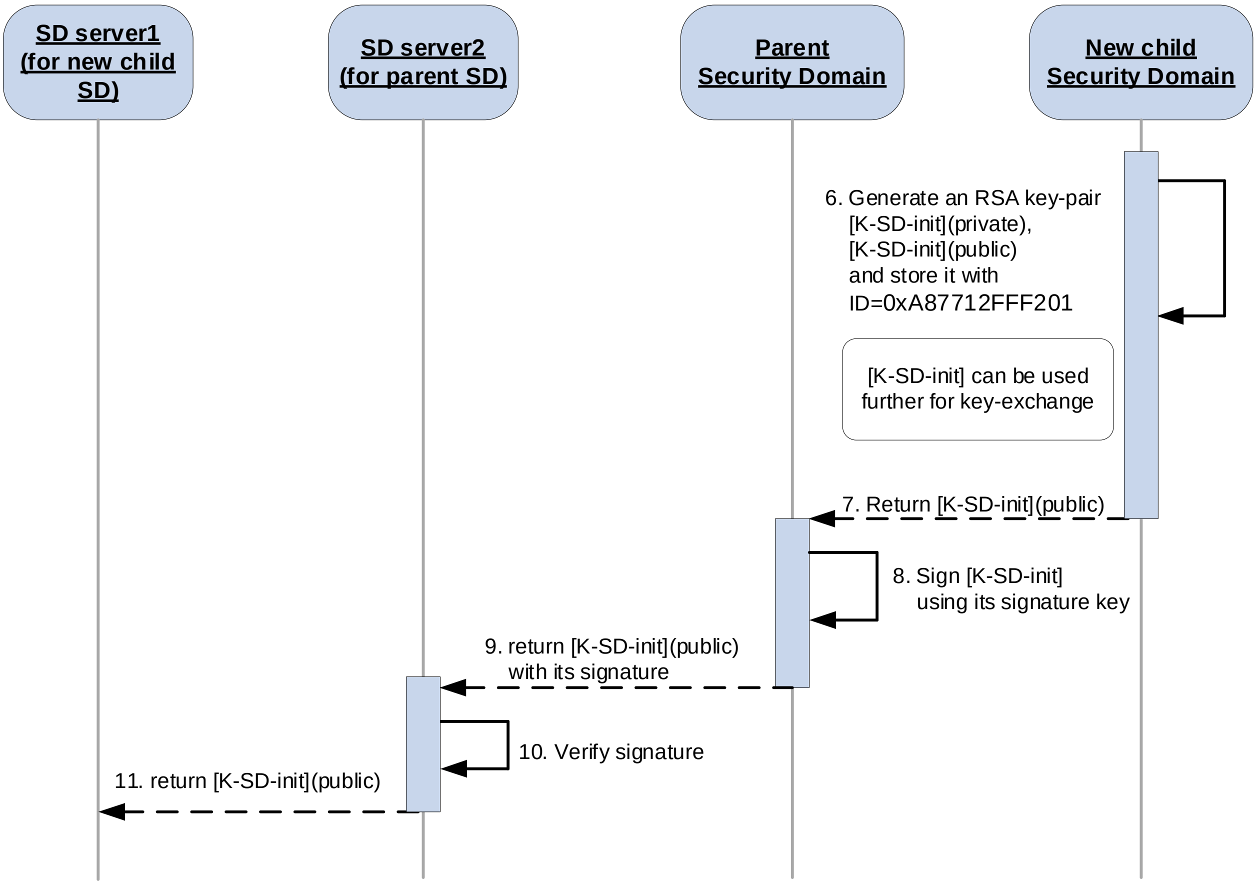
Key Identifier:	`0x0000FFFFF8` (or any object identifier value)
Key Type:	<any key type: public RSA…>
Key Size:	<length of the key in bits>

B.3.2 Key Generation for Key Exchange

In addition to the procedure described in Figure B-5, the owner of the newly created Security Domain (SD server1) would like to obtain a key returned during the installation operation for further provisioning operations.

In the following figure, the scenario is based on the procedure described in section 8.3.4.2 where a public RSA key is returned.

Figure B-6: Key Provisioning Preparation


SD server1 (for new child SD)
SD server2 (for parent SD)
Parent Security Domain
New child Security Domain
6. Generate an RSA key-pair [K-SD-init](private), [K-SD-init](public) and store it with ID=0xA87712FFF201
[K-SD-init] can be used further for key-exchange
7. Return [K-SD-init](public)
8. Sign [K-SD-init] using its signature key
9. return [K-SD-init](public) with its signature
10. Verify signature
11. return [K-SD-init](public)

6. The newly installed Security Domain generates an RSA key-pair [K-SD-init] and stores it as a persistent key. This key-pair will be used later to perform a key-exchange with SD server1. The *Key Identifier* value of [K-SD-init] has been provided by SD server1.

    Key Identifier: `0xA87712FFF201` (or any object identifier value)

    Key Type: `TEE_TYPE_RSA_KEYPAIR`

    Key Size: 2048 bits

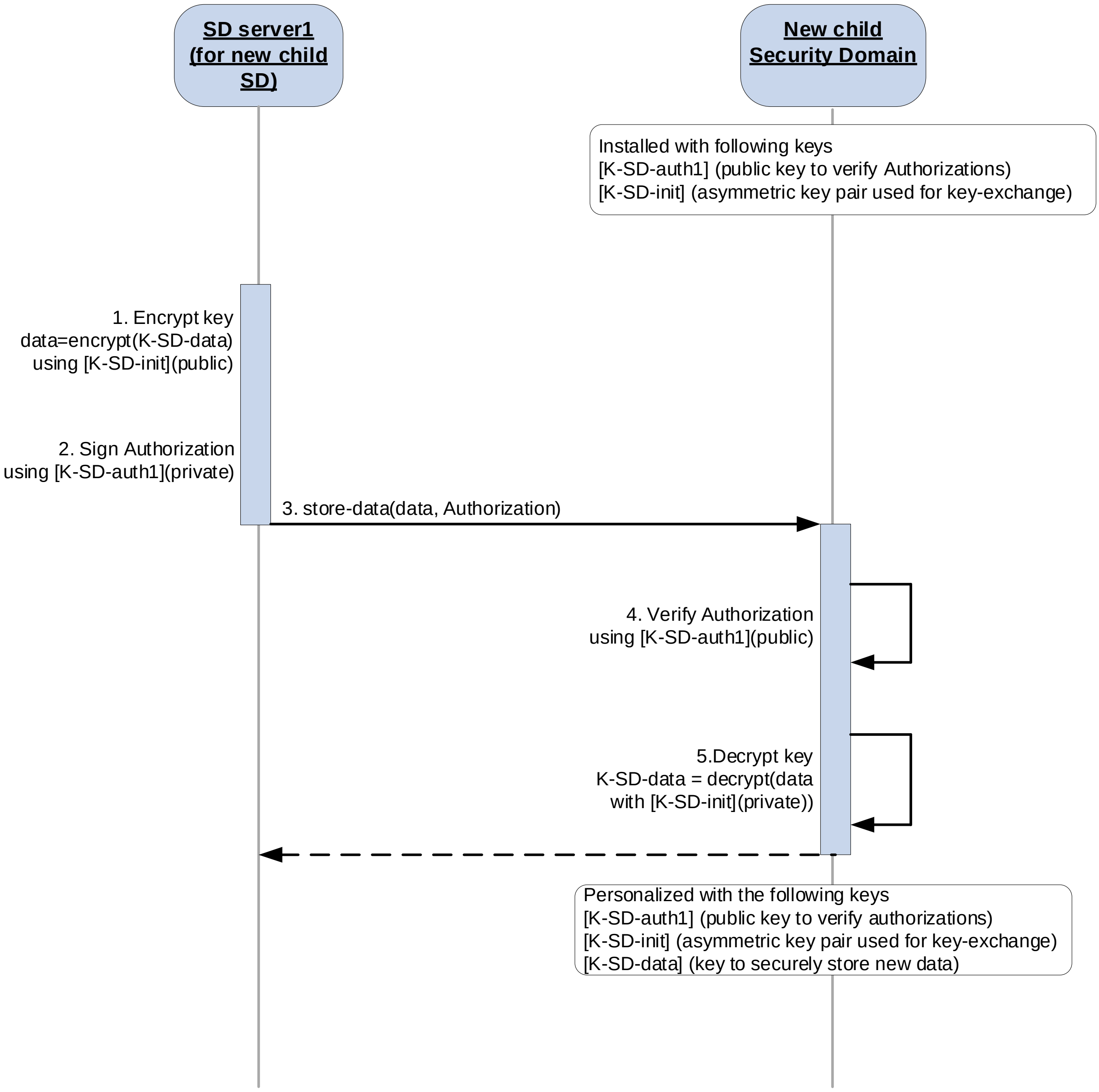
7. The public part of the [K-SD-init] key is returned to the Security Domain performing the operation.
8. The public part of the [K-SD-init] key is signed by the Security Domain performing the operation to authenticate its origin using a previously provisioned key of this Security Domain.
9. The public part of the [K-SD-init] key and its signature are returned to the administration server (SD server2).
10. The signature of the [K-SD-init] key is verified to authenticate its origin.
11. The public part of the [K-SD-init] key is then forwarded to the requester (SD server1).

B.3.3 Provisioning New Keys

The asymmetric key-pair generated during the Security Domain installation can now be used to perform the key-exchange protocol between the newly created Security Domain ('New child SD') and its management server (SD server1).

In the following scheme, the SD server uses the public key generated and returned by the child SD during its installation to encrypt the key K-SD-data (see section B.3.2).

Figure B-7: Key Provisioning for Data Confidentiality


SD server1 (for new child SD)
New child Security Domain
Installed with following keys
[K-SD-auth1] (public key to verify Authorizations)
[K-SD-init] (asymmetric key pair used for key-exchange)
1. Encrypt key data=encrypt(K-SD-data) using [K-SD-init](public)
2. Sign Authorization using [K-SD-auth1](private)
3. store-data(data, Authorization)
4. Verify Authorization using [K-SD-auth1](public)
5.Decrypt key K-SD-data = decrypt(data with [K-SD-init](private))
Personalized with the following keys
[K-SD-auth1] (public key to verify authorizations)
[K-SD-init] (asymmetric key pair used for key-exchange)
[K-SD-data] (key to securely store new data)

12. Encrypt the key to provision using the public key [K-SD-init] generated by the newly created Security Domain during its installation.

13. Generate the Authorization to perform a Store Data and sign it using the private key corresponding to the public key [K-SD-auth1] provisioned during the installation of the newly created Security Domain.

14. Send the Store Data command with the encrypted data. The *Key Identifier* parameter passed with the command for decryption is set to `0xA87712FFF201`, referring to [K-SD-init].

15. The Security Domain verifies the Authorization first.

16. The Security Domain decrypts the data using the private key [K-SD-init] referred by the identifier `0xA87712FFF201` given as a parameter of the Store Data and creates the corresponding key object.

B.4 Encoding Examples

Note: Some example encoding tables may still include incorrect lengths; these will be addressed in a later release.

To create the example ASN.1 we have replaced some context dependent fields with generic strings. In real ASN.1 packets, these strings would be replaced with material of the appropriate data type and format, and the lengths adjusted appropriately.

B.4.1 Command Request Message

Given the above grammar, a hypothetical `RequestMessage` record can formally be described as follows:

```
RequestMessage {
     version               1.0.0.0,
     AuthorizationToken    "some encoded authorization token",
     command               LockTEE
}
```

and encoded as follows.

Table B-1: Command Request Message Encoding Values

Tag	Length	Value (in hex)	Description
`0x60`	`0x2b`		`RequestMessage` structure of length 43 octets
`0x02`	`0x04`	`01 00 00 00`	`version`: 1.0.0.0
`0x76`	`0x20`	`73 6f 6d 65 20 65 6e 63 6f 64 65 64 20 61 75 74 68 6f 72 69 7a 61 74 69 6f 6e 20 74 6f 6b 65 6e`	A signed Authorization Token: Here substituted with the dummy value "some encoded authorization token"
`0x7f5a`	`0x00`		Lock TEE command

B.4.2 Command Response Message

Given the above grammar, a hypothetical `ResponseMessage` record can formally be described as follows:

```
ResponseMessage {
        status TEE_SUCCESS,
        response "some encoded response"
}
```

and encoded as follows.

Table B-2: Command Response Message Encoding Values

Tag	Length	Value (in hex)	Description
`0x61`	`0x1a`		`ResponseMessage` structure of length 26 octets
`0x02`	`0x01`	`00`	`status`: `TEE_SUCCESS`
<responseTag> **(response-dependent)**	`0x15`	`73 6f 6d 65 20 65 6e 63 6f 64 65 64 20 72 65 73 70 6f 6e 73 65`	`response`: Here substituted with the dummy value "some encoded response"

B.4.3 Install TA Command

Given the above grammar, a hypothetical `InstallTA` command can formally be described as follows:

```
InstallTA {
        ta "abcdef01-2345-6789-abcd-ef0123456789",
        targetSD "abcdef02-2345-6789-abcd-ef0123456789",
        initialState Executable,
        applicationFile "some encrypted value",
        encryptionParams {
                keyID "my key",
                cryptoParams {
                        algorithmID TEE_ALG_AES_CBC_MAC_PKCS5
                        operationMode  TEE_MODE_DECRYPT,
                        algoParams "IV value"
                }
        },
        uuidVerificationParams {
                protocol 0x6bc2de43501248559c8eeaaf0cb9fde7,
                version 0x01,
                uuidV5Params {
                    keyType TEE_TYPE_RSA_PUBLIC_KEY,
                    keySize  2048,
                    keyAttributes {
                          Attribute {
                                  id          TEE_ATTR_RSA_MODULUS,
                                  value    "modulus"
                          },
                          Attribute {
                                  id          TEE_ATTR_RSA_PUBLIC_EXPONENT,
                                  value   "exponent"
                          }
                     },
                    signatureParams {
                          algorithmID        TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA256,
                          operationMode   TEE_MODE_VERIFY
                    },
                    signature "some signature value"
                  }
          }
  }
```

and encoded as follows.

Table B-3: Install TA Command Encoding Values

Tag	Length	Value (in hex)	Description
0x7f41	0xc5		InstallTA structure of length 197 octets
0x43	0x10	ab cd ef 01 23 45 67 89 ab cd ef 01 23 45 67 89	ta: "abcdef01-2345-6789-abcd-ef0123456789"
0x43	0x10	ab cd ef 02 23 45 67 89 ab cd ef 01 23 45 67 89	targetSD: "abcdef02-2345-6789-abcd-ef0123456789"
0x53	0x01	01	initialState: Executable
0x04	0x14	73 6f 6d 65 20 65 6e 63 72 79 70 74 65 64 20 76 61 6c 75 65	applicationFile: Here substituted with the dummy value "some encrypted value"
0x66	0x1d		encryptionParams structure of length 29 octets
0x44	0x06	6d 79 20 6b 65 79	Encryption Key ID: Here substituted with the dummy value “my key”
0x65	0x13		cryptoParams structure of length 19 octets
0x02	0x04	30 00 05 10	Algorithm identifier: TEE_ALG_AES_CBC_MAC_PKCS5
0x02	0x01	01	operationMode: TEE_MODE_DECRYPT
0x04	0x08	49 56 20 76 61 6c 75 65	Initial Vector: Here substituted with the dummy value “IV value”
0x68	0x67		uuidVerificationParams structure of length 103 octets
0x43	0x10	6b c2 de 43 50 12 48 55 9c 8e ea af 0c b9 fd e7	protocol: (UUID v5 verification)
0x02	0x01	01	Version of protocol
0xa0	0x50		uuidV5Params structure of length 80 octets
0x02	0x04	A0 00 00 30	Key type: TEE_TYPE_RSA_PUBLIC_KEY
0x02	0x02	08 00	Key size: 2048
0x30	0x23		SEQUENCE of Attribute structure of length 35 octets
0x62	0x0f		Attribute structure of length 15 octets
0x02	0x04	D0 00 01 30	Attribute id: TEE_ATTR_RSA_MODULUS
0x04	0x07	6d 6f 64 75 6c 75 73	Modulus attribute: Here substituted with the dummy value "modulus"
0x62	0x10		Attribute structure of length 16 octets

Tag	Length	Value (in hex)	Description
0x02	0x04	D0 00 02 30	Attribute id: TEE_ATTR_RSA_PUBLIC_EXPONENT
0x04	0x08	65 78 70 6f 6e 65 6e 74	Exponent attribute: Here substituted with the dummy value "exponent"
0x65	0x09		signatureParams structure of length 9 octets
0x02	0x04	70 41 49 30	Algorithm identifier: TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA256
0x02	0x01	03	operationMode: TEE_MODE_VERIFY
0x04	0x14	73 6f 6d 65 20 73 69 67 6e 61 74 75 72 65 20 76 61 6c 75 65	signature: Here substituted with the dummy value "some signature value"

B.4.4 Install SD Command

We use an example of the procedure described in section 8.3.4 to illustrate the encoding of the Install SD command.

Given the above grammar, a hypothetical Install SD command can formally be described as follows:

```
InstallSD {
    sd "abcdef01-2345-6789-abcd-ef0123456789",
    targetSD "abcdef02-2345-6789-abcd-ef0123456789",
    initialState Active,
    SDPrivileges {
        listOfPrivileges {
            { gpd.privilege.teeManagement },
            { gpd.privilege.sdManagement },
            { gpd.privilege.sdPersonalization }
          },
        isRootSD TRUE
    },
    authority   {
        name “”
    },
    cryptographicData {
        cryptoProcID      INST_SD_GEN_RSA_KEYPAIR_PROC,
        inputRSAPubKey {
            keyID    "key1",
            keyType TEE_TYPE_RSA_KEYPAIR,
            accessAndShareRights  (TEE_DATA_FLAG_ACCESS_WRITE |
                                   TEE_DATA_FLAG_ACCESS_READ)
            keyAttributes {
                Attribute {
                    type TEE_ATTR_RSA_MODULUS,
                    value "modulus"
                 },
                Attribute {
                    type TEE_ATTR_RSA_PUBLIC_EXPONENT,
                    value "exponent"
                }
            },
                          metadata {
                                   sizeInBits   2048
                                   usageFlags TEE_USAGE_VERIFY
                         }
                 },
                 genKeyDesc  {
                          keyId         "key2",
                          keyType       TEE_TYPE_RSA_KEYPAIR,
                          keyUsage     TEE_USAGE_ENCRYPT,
                          keySize       2048
                 }
                 signatureInfos {
                         keyID "my signature key",
                         signatureParams {
                             algorithmID           TEE_ALG_RSASSA_PKCS1_V1_5_SHA256
                             operationMode       TEE_MODE_VERIFY
                         }
                 }
         },
         uuidVerificationParams {
                protocol 0x6bc2de43501248559c8eeaaf0cb9fde7,
                version 0x01,
                uuidV5Params {
                    keyType TEE_TYPE_RSA_PUBLIC_KEY,
                    keySize  2048,
                    keyAttributes  {
                         Attribute {
                                 id          TEE_ATTR_RSA_MODULUS,
                                  "modulus"
                         },
                         Attribute {
                                 id          TEE_ATTR_RSA_PUBLIC_EXPONENT,
                                 value   "exponent"
                         }
                     },
                   signatureParams {
                    algorithmID        TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA256,
                    operationMode   TEE_MODE_VERIFY
                },
                signature "some signature value"
            }
      }
  }
```

and encoded as follows.

Table B-4: Install SD Command Encoding Values

Tag	Length	Value (in hex)	Description
0x7f4a	0x0141		InstallSD structure of length 321 octets
0x43	0x10	ab cd ef 01 23 45 67 89 ab cd ef 01 23 45 67 89	sd: "abcdef01-2345-6789-abcd-ef0123456789"
0x43	0x10	ab cd ef 02 23 45 67 89 ab cd ef 01 23 45 67 89	targetSD: "abcdef02-2345-6789-abcd-ef0123456789"
0x53	0x01	01	initialState: Active
0x7b	0x14		SDPrivileges structure of length 20 octets
0x30	0x0f		SEQUENCE of Privilege structure of length 15 octets
0x30	0x03		Privilege #1 structure of length 3 octets
0x02	0x01	40	gpd.privilege.teeManagement
0x30	0x03		Privilege #2 structure of length 3 octets
0x02	0x01	41	gpd.privilege.sdManagement
0x30	0x03		Privilege #3 structure of length 3 octets
0x02	0x01	42	gpd.privilege.sdPersonalization
0x01	0x01	01	isRootSD = TRUE
0x7c	0x02		Authority structure of length 2 octets
0x0c	0x00		Authority name (empty string)
0x69	0x95		CryptographicData structure of length 149 octets
0x02	0x01	02	cryptoProcID: INST_SD_GEN_RSA_KEYPAIR_PROC
0x04	0x1d	52 53 41 47 65 6e 4b 65 79 44 61 74 61 20 73 74 72 75 63 74 75 72 65 20 76 61 6c 75 65	OCTET STRING containing the RSAGenKeyData structure value (DER-encoded): Here substituted with the dummy value “RSAGenKeyData structure value”
0x67	0x3d		inputRSAPubKey: StoredDataObject structure of length 61 octets

Tag	Length	Value (in hex)	Description
0x44	0x04	6b 65 79 31	keyID value: Here substituted with the dummy value "key1"
0x02	0x04	A1 00 00 30	keyType value: TEE_TYPE_RSA_KEYPAIR
0x02	0x01	03	Access And Share Rights: READ&WRITE
0x30	0x23		SEQUENCE of Attribute structure of length 35 octets
0x62	0x0f		Attribute structure of length 15 octets
0x02	0x04	D0 00 01 30	Attribute id: TEE_ATTR_RSA_MODULUS
0x04	0x07	6d 6f 64 75 6c 75 73	Modulus attribute value: Here substituted with the dummy value "modulus"
0x62	0x10		Attribute structure of length 16 octets
0x02	0x04	D0 00 02 30	Attribute id: TEE_ATTR_RSA_PUBLIC_ EXPONENT
0x04	0x08	65 78 70 6f 6e 65 6e 74	Exponent attribute value: Here substituted with the dummy value "exponent"
0x30	0x07		Metadata structure of length 7 octets
0x02	0x02	08 00	Size in bits: 2048
0x02	0x01	20	key usage: TEE_USAGE_VERIFY
0x30	0x13		genKeyDesc structure of length 19 octets
0x44	0x04	6b 65 79 32	keyId: Here substituted with the dummy value "key2"
0x02	0x04	A1 00 00 30	keyType: TEE_TYPE_RSA_KEYPAIR
0x02	0x01	02	keyUsage: TEE_USAGE_ENCRYPT
0x02	0x02	08 00	keySize: 2048
0x66	0x1d		signatureInfos structure of length 29 octets
0x44	0x10	6d 79 20 73 69 67 6e 61 74 75 72 65 20 6b 65 79	keyId: Here substituted with the dummy value "my signature key"
0x66	0x09		signatureParams structure of length 9 octets
0x02	0x04	70 00 48 30	algorithmID: TEE_ALG_RSASSA_PKCS1_V1_5_SHA256
0x02	0x01	03	Operation Mode: TEE_MODE_VERIFY
0x68	0x6a		uuidVerificationParams structure of length 103 octets
0x43	0x10	6b c2 de 43 50 12 48 55 9c 8e ea af 0c b9 fd e7	protocol (UUID v5 verification)
0x02	0x01	01	Version of protocol
0xa0	0x50		uuidV5Params structure of length 80 octets
0x02	0x04	A0 00 00 30	Key type: TEE_TYPE_RSA_PUBLIC_KEY
0x02	0x02	08 00	Key size: 2048
0x30	0x23		SEQUENCE of Attribute structure of length 35 octets

Tag	Length	Value (in hex)	Description
0x62	0x0f		Attribute structure of length 15 octets
0x02	0x04	D0 00 01 30	Attribute id: TEE_ATTR_RSA_MODULUS
0x04	0x07	6d 6f 64 75 6c 75 73	Modulus attribute: Here substituted with the dummy value "modulus"
0x62	0x10		Attribute structure of length 16 octets
0x02	0x04	D0 00 02 30	Attribute id: TEE_ATTR_RSA_PUBLIC_EXPONENT
0x04	0x08	65 78 70 6f 6e 65 6e 74	Exponent attribute: Here substituted with the dummy value "exponent"
0x65	0x09		Crypto operation parameters structure of length 9 octets (signature verification)
0x02	0x04	70 41 49 30	Algorithm identifier: TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA256
0x02	0x01	03	Operation Mode: TEE_MODE_VERIFY
0x04	0x14	73 6f 6d 65 20 73 69 67 6e 61 74 75 72 65 20 76 61 6c 75 65	signature: Here substituted with the dummy value "some signature value"

B.4.5 Install SD Response

Given the above grammar, a hypothetical Install SD Response can formally be described as follows:

```
InstallSDResp {
   CryptographicData {
      CryptoProcID    INST_SD_GEN_RSA_KEYPAIR_PROC,
      RSAGenProcOutput {
          genKeyValue { - - the generated RSA public key
             Attribute {
                  id          TEE_ATTR_RSA_MODULUS,
                  refValue  "modulus"
             },
            Attribute {
                  id          TEE_ATTR_RSA_PUBLIC_EXPONENT,
                  refValue  "exponent"
            }
         },
         signature "signature over the genKeyValue"  - - using signature algo given in the Install SD cmd
      }
   }
}
```

and encoded as follows.

Table B-5: Install SD Response Encoding Values

Tag	Length	Value (in hex)	Description
0x69	0x6f		`InstallSDResp`: a `CryptographicData` structure of length 108 octets
0x02	0x01	02	cryptoProcID: INST_SD_GEN_RSA_KEYPAIR_PROC
0x04	0x20	52 53 41 47 65 6e 50 72 6f 63 4f 75 74 70 75 74 20 73 74 72 75 63 74 75 72 65 20 76 61 6c 75 65	OCTET STRING containing the `RSAGenProcOutput` structure value (DER-encoded): Here substituted with the dummy value “RSAGenProcOutput structure value”
0x30	0x47		An `RSAGenProcOutput` structure of length 69 octets
0x30	0x225		A list of Attributes of length 35 octets
0x62	0x10		`Attribute` structure of length 15 octets
0x02	0x04	D0 00 01 30	Attribute `id`: `TEE_ATTR_RSA_MODULUS`

Tag	Length	Value (in hex)	Description
`0x04`	`0x07`	`6d 6f 64 75 6c 75 73`	Modulus attribute: Here substituted with the dummy value "modulus"
`0x62`	`0x10`		`Attribute` structure of length 16 octets
`0x02`	`0x04`	`D0 00 02 30`	Attribute `id`: `TEE_ATTR_RSA_PUBLIC_EXPONENT`
`0x04`	`0x08`	`65 78 70 6f 6e 65 6e 74`	Exponent attribute: Here substituted with the dummy value "exponent"
`0x04`	`0x1e`	`73 69 67 6e 61 74 75 72 65 20 6f 76 65 72 20 74 68 65 20 67 65 6e 4b 65 79 56 61 6c 75 65`	`signature`: Here substituted with the dummy value "signature over the genKeyValue"

B.4.6 TEE Characteristics

Given the above definitions and grammar, a hypothetical `Tee` record can formally be described as follows:

```
tee {
device {
        name "aDevice", id "abcdef01-2345-6789-abcd-ef0123456789",
        manufacturer "acompany", version "3.25.6", type "aType"
},
trustedOS {
        name "OS name", manufacturer "manufacturer name", version "1.23.256",
        isaSet {
            ISA {
                name "ISX V7 32 bit", processorType "ISX",
                instructionSet "T32", addressSize 32,
                ABI "ISXV7", endianness 1
            },
            ISA {
                name "ISX V8 64 bit", processorType "ISX",
                instructionSet "A64", addressSize 64,
                abi "ISXV8", endianness 1
            }
        },
        options {
            Option { name "aaa", version 2.0 },
            Option { name "ccc", version 2.1.1.2}
        },
                    protocols {
                                { protocol "abcdef01-2345-6789-abcd-ef0123456789" }
                    }
  },
  state secure,
  roots { } -- no rSD
  optionalApis {
                    Option {   name "TMF", version 1.1.0.0 },
                    Option {   name "TrustedUI", version 1.0.0.0 },
                    Option {   name "SE", version 1.0.0.0 },
                    Option {   name "Debug-PMR", version 1.0.0.0 },
                    Option {   name "Sockets", version 1.0.0.0 }
  }
          teeImplementationProperties {
               Property { name "gpd.tee.apiversion", value (UTF-8 string) "1.1" },
               Property { name "gpd.tee.internalCore.version", value (integer) 0x01010200 }, -- 1.1.2
               Property { name "gpd.tee.description", value "Trustonic's latest and greatest" },
               Property { name "gpd.tee.deviceID", UUID {"abcdef01-2345-6789-abcd-ef0123456789"} },
               Property { name "gpd.tee.systemTime.protectionLevel", value 1000 },
               Property { name "gpd.tee.TAPersistentTime.protectionLevel", value 100 },
               Property { name "gpd.tee.trustedos.implementation.version", value "1.3pl94" },
               Property { name "gpd.tee.firmware.manufacturer", value "XXXYYYY" },
               Property { name "gpd.tee.tmf.resetpreserved.entities", value LIST of UUIDs {"abcdef01-2345-
  6789-abcd-ef0123456789", "abcdef02-2345-6789-abcd-ef0123456789"}}
            },
            teePlatformLabel   "GP x.y"
  }
```

and encoded as follows.

Table B-6: TEE Encoding Values

Tag	Length	Value (in hex)	Description
0x70	0x02f6		Tee structure of length 758 octets
0x6d	0x34		Device structure of length 52 octets
0x0c	0x07	61 44 65 76 69 63 65	name: "aDevice"
0x43	0x10	ab cd ef 01 23 45 67 89 ab cd ef 01 23 45 67 89	id: "abcdef01-2345-6789-abcd-ef0123456789"
0x0c	0x08	61 63 6f 6d 70 61 6e 79	manufacturer: "acompany"
0x12	0x06	33 2e 32 35 2e 36	version: "3.25.6"
0x0c	0x05	61 54 79 70 65	type: "aType"
0x6f	0xaa		TrustedOS structure of length 170 octets
0x0c	0x07	4f 53 20 6e 61 6d 65	name: "OS name"

Tag	Length	Value (in hex)	Description
0x0c	0x11	6d 61 6e 75 66 61 63 74 75 72 65 72 20 6e 61 6d 65	manufacturer: "manufacturer name"
0x12	0x08	31 2e 32 33 2e 32 35 36	version: "1.23.256"
0x30	0x50		SEQUENCE of ISA structure of length 80 octets
0x6e	0x26		ISA structure of length 38 octets
0x0c	0x0d	49 53 58 20 56 37 20 33 32 20 62 69 74	name: "ISX V7 32 bit"
0x0c	0x03	49 53 58	processorType: "ISX"
0x12	0x03	5A 33 32	instructionSet: "T32"
0x02	0x01	20	addressSize: 32
0x12	0x05	49 53 58 56 37	ABI: "ISXV7"
0x02	0x01	01	endianness
0x6e	0x26		ISA structure of length 38 octets
0x0c	0x0d	49 53 58 20 56 38 20 36 34 20 62 69 74	name: "ISX V8 64 bit"
0x0c	0x03	49 53 58	processorType: "ISX"
0x12	0x03	41 36 34	instructionSet: "A64"
0x02	0x01	40	addressSize: 64
0x12	0x05	49 53 58 56 38	ABI: "ISXV8"
0x02	0x01	01	endianness
0xa0	0x1a		SEQUENCE of Option structure of length 26 octets
0x6e	0x0b		Option structure of length 11 octets
0x0c	0x03	61 61 61	name: "aaa"
0x02	0x04	02 00 00 00	version: 2.0
0x6c	0x0b		Option structure of length 11 octets
0x0c	0x03	63 63 63	name: "ccc"
0x02	0x04	02 01 01 02	version: 2.1.1.2
0xa1	0x14		protocols: a SEQUENCE of SecureLayerAuditInfo structures of length 20 octets
0x7d	0x12		A SecureLayerAuditInfo structure of length 18 octets
0x43	0x10	ab cd ef 01 23 45 67 89 ab cd ef 01 23 45 67 89	protocol: "abcdef01-2345-6789-abcd-ef0123456789"
0x02	0x01	01	state of TEE: secure
0x30	0x00		Empty list of rSD UUIDs
0xa0	0x50		OptionalApis: a SEQUENCE of Option structure of length 80 octets

Tag	Length	Value (in hex)	Description
0x6c	0x0b		Option structure of length 11 octets
0x0c	0x03	54 4d 46	name: "TMF"
0x02	0x04	01 01 00 00	version: 1.1.0.0
0x6c	0x11		Option structure of length 17 octets
0x0c	0x09	54 72 75 73 74 65 64 55 49	name: "TrustedUI"
0x02	0x04	01 00 00 00	version: 1.0.0.0
0x6c	0x0a		Option structure of length 10 octets
0x0c	0x02	53 45	name: "SE"
0x02	0x04	01 00 00 00	version: 1.0.0.0
0x6c	0x11		Option structure of length 17 octets
0x0c	0x09	44 65 62 75 67 2d 50 4d 52	name: "Debug-PMR"
0x02	0x04	01 00 00 00	version: 1.0.0.0
0x6c	0x0f		Option structure of length 15 octets
0x0c	0x07	53 6f 63 6b 65 74 73	name: "Sockets"
0x02	0x04	01 00 00 00	version: 1.0.0.0
0xa1	0x01b2		teeImplementationProperties: a SEQUENCE of Property structure of length 434 octets
0x6c	0x19		Property structure #1 of length 25 octets
0x0c	0x12	67 70 64 2e 74 65 65 2e 61 70 69 76 65 72 73 69 6f 6e	name: "gpd.tee.apiversion"
0x0c	0x03	31 2e 31	value (UTF-8 string) "1.1"
0x6c	0x24		Property structure #2 of length 36 octets
0x0c	0x1c	67 70 64 2e 74 65 65 2e 69 6e 74 65 72 6e 61 6c 43 6f 72 65 2e 76 65 72 73 69 6f 6e	name: "gpd.tee.internalCore.version"
0x0c	0x04	01 01 02 00	value (integer) equivalent to version 1.1.2
0x6a	0x36		Property structure #3 of length 54 octets
0x0c	0x13	67 70 64 2e 74 65 65 2e 64 65 73 63 72 69 70 74 69 6f 6e	name: "gpd.tee.description"
0x0c	0x1f	54 72 75 73 74 6f 6e 69 63 27 73 20 6c 61 74 65 73 74 20 61 6e 64 20 67 72 65 61 74 65 73 74	value (UTF-8): "Trustonic's latest and greatest"
0x6a	0x24		Property structure #4 of length 36 octets
0x0c	0x10	67 70 64 2e 74 65 65 2e 64 65 76 69 63 65 49 44	name: "gpd.tee.deviceID"

Tag	Length	Value (in hex)	Description
0x43	0x10	ab cd ef 01 23 45 67 89 ab cd ef 01 23 45 67 89	UUID: "abcdef01-2345-6789-abcd-ef0123456789"
0x6a	0x29		Property structure #5 of length 41 octets
0x0c	0x23	67 70 64 2e 74 65 65 2e 73 79 73 74 65 6d 54 69 6d 65 2e 3f 70 72 6f 74 65 63 74 69 6f 6e 4c 65 76 65 6c	name: "gpd.tee.systemTime. protectionLevel"
0x02	0x02	03 e8	value (integer) 1000
0x6a	0x2e		Property structure #6 of length 46 octets
0x0c	0x29	67 70 64 2e 74 65 65 2e 54 41 50 65 72 73 69 73 74 65 6e 74 54 69 6d 65 2e 3f 70 72 6f 74 65 63 74 69 6f 6e 4c 65 76 65 6c	name: "gpd.tee.TAPersistentTime. protectionLevel"
0x02	0x01	64	value (integer): 100
0x6a	0x35		Property structure #7 of length 53 octets
0x0c	0x2A	67 70 64 2e 74 65 65 2e 74 72 75 73 74 65 64 6f 73 2e 3f 69 6d 70 6c 65 6d 65 6e 74 61 74 69 6f 6e 2e 3f 76 65 72 73 69 6f 6e	name: "gpd.tee.trustedos. implementation.version"
0x0c	0x07	31 2e 33 70 6c 39 34	value (UTF-8): "1.3pl94"
0x6a	0x29		Property structure #8 of length 41 octets
0x0c	0x1e	67 70 64 2e 74 65 65 2e 66 69 72 6d 77 61 72 65 2e 3f 6d 61 6e 75 66 61 63 74 75 72 65 72	name: "gpd.tee.firmware.manufacturer"
0x0c	0x07	58 58 58 59 59 59 59	value (UTF-8): "XXXYYYY"
0x6a	0x54		Property structure #9 of length 84 octets
0x0c	0x24	67 70 64 2e 74 65 65 2e 74 6d 66 2e 72 65 73 65 74 70 72 65 73 65 72 76 65 64 2e 3f 65 6e 74 69 74 69 65 73	name: "gpd.tee.tmf.resetpreserved. entities"
0x04	0x2c	71 38 33 76 41 53 4e 46 5a 34 6d 72 7a 65 38 42 49 30 56 6e 69 61 76 4e 37 77 49 6a 52 57 65 4a 71 38 33 76 41 53 4e 46 5a 34 6b 3d	value (binary) Base64({ArrayOfBytes("abcdef01-2345-6789-abcd-ef0123456789")||ArrayOfBytes("abcdef02-2345-6789-abcd-ef0123456789")})
0xc	0x06	47 50 20 78 2e 79	teePlatformLabel: "GP x.y"

B.4.7 SD Characteristics

Given the above definitions and grammar, a hypothetical Security Domain record can formally be described as follows:

```
SecurityDomain {
   id "abcdef02-2345-6789-abcd-ef0123456789",
   parent  "abcdef01-2345-6789-abcd-ef0123456789",
   lifecycleState Active,
   authority {
         name  "acme corp."
         urlInfo "http://d/e/f/g"
   },
   subdomains {
                    UUID { "abcdef01-2345-6789-abcd-ef0123456789" },
                    UUID { "abcdef02-2345-6789-abcd-ef0123456789" }
   }
}
```

and encoded as follows.

Table B-7: Security Domain Encoding Values

Tag	Length	Value (in hex)	Description
0x72	0x6b		SecurityDomain structure of length 107 octets
0x43	0x10	ab cd ef 02 23 45 67 89 ab cd ef 01 23 45 67 89	id: "abcdef02-2345-6789-abcd-a1ef0123456789"
0x43	0x10	ab cd ef 01 23 45 67 89 ab cd ef 01 23 45 67 89	parent: "abcdef01-2345-6789-abcd-ef0123456789"
0x51	0x01	01	lifecycleState: Active
0x7c	0x1c		Authority structure of length 28 octets
0x0c	0x0a	61 63 6d 65 20 63 6f 72 70 2e	name: "acme corp."
0x0c	0x0e	68 74 74 70 3a 2f 2f 64 2f 65 2f 66 2f 67	urlInfo: "http://d/e/f/g"
0xa0	0x24		SEQUENCE OF UUID (subdomains) structure of length 36 octets
0x43	0x10	ab cd ef 01 23 45 67 89 ab cd ef 01 23 45 67 89	subdomain: "abcdef01-2345-6789-abcd-ef0123456789"
0x43	0x10	ab cd ef 02 23 45 67 89 ab cd ef 01 23 45 67 89	subdomain: "abcdef02-2345-6789-abcd-ef0123456789"

B.4.8 TA Characteristics

Given the above definitions and grammar, a hypothetical `TrustedApplication` record can formally be described as follows:

```
        TrustedApplication {
                id "abcdef03-2345-6789-abcd-ef0123456789",
                parent "abcdef02-2345-6789-abcd-ef0123456789",
                lifecycleState Locked
                version "3.1"
          }
```

and encoded as follows.

Table B-8: Trusted Application Encoding Values

Tag	Length	Value (in hex)	Description
`0x76`	`0x2c`		`TrustedApplication` structure of length 44 octets

`0x43`	`0x10`	`ab cd ef 03 23 45 67 89`
		`ab cd ef 01 23 45 67 89`
			`id`: "abcdef03-2345-6789-abcd-ef0123456789"

`0x43`	`0x10`	`ab cd ef 02 23 45 67 89`
		`ab cd ef 01 23 45 67 89`
			`parent`: "abcdef02-2345-6789-abcd-ef0123456789"

`0x53`	`0x01`	`02`	`lifecycleState`: Locked
`0x12`	`0x03`	`33 2e 31`	`version`: "3.1"

B.4.9 Authorization Token

Given the above grammar, a hypothetical Authentication Token record can formally be described as follows:

```
AuthorizationToken {
    payload {
        version 1.0,
        authorizingSd "abcdef01-2345-6789-abcd-ef0123456789",
        constraintsList {
            ConstraintDeviceId {
                "abcdef01-2345-6789-abcd-ef0123456789"
             },
            ConstraintMinVersion {
                1
             },
            ConstraintParamsDigest  {
              algorithm “algorithm1”,
              bitmap “bitmap3”,
              digest "01020304010203040102030401020304"
            }           },
       signatureParams {
            keyId "my key",
            cryptoOperationParams {
               algorithmID     “algorithmId7”,
               operationMode   TEE_MODE_VERIFY
              }
       }
  },
  signature "01020304010203040102030401020304"
}
```

and encoded as follows.

Table B-9: Authorization Token Encoding Values

Tag	Length	Value (in hex)	Description
`0x76`	`0x89`		`AuthorizationToken` structure of length 137 octets
`0x75`	`0x53`		Token payload structure of length 119 octets

Tag	Length	Value (in hex)	Description
0x02	0x04	01 00 00 00	version: 1.0
0x43	0x10	ab cd ef 01 23 45 67 89 ab cd ef 01 23 45 67 89	authorizingSd: "abcdef01-2345-6789-abcd-ef0123456789"
0x30	0x3e		SEQUENCE OF constraints of length 62 octets
0xc1	0x10	ab cd ef 01 23 45 67 89 ab cd ef 01 23 45 67 89	ConstraintDeviceId: "abcdef01-2345-6789-abcd-ef0123456789"
0xc3	0x01	01	ConstraintMinVersion
0xe0	0x27		ConstraintParamsDigest structure of length 39 octets
0x02	0x0a	61 6c 67 6f 72 69 74 68 6d 31	algorithm: Here substituted with the dummy value “algorithm1”
0x02	0x07	62 69 74 6d 61 70 33	bitmap: Here substituted with the dummy value “bitmap3”
0x04	0x10	01 02 03 04 01 02 03 04 01 02 03 04 01 02 03 04	digest: Here substituted with the dummy value "01020304010203040102030401020304"
0x66	0x1b		signatureParams structure of length 27 octets
0x44	0x06	6d 79 20 6b 65 79	keyId: Here substituted with the dummy value "my key"
0x65	0x11		cryptoOperationParams structure of length 17 octets
0x02	0x0C	61 6c 67 6f 72 69 74 68 6d 49 64 37	algorithmID: Here substituted with the dummy value “algorithmId7”
0x02	0x01	03	Operation Mode: TEE_MODE_VERIFY
0x04	0x10	01 02 03 04 01 02 03 04 01 02 03 04 01 02 03 04	signature: Here substituted with the dummy value "01020304010203040102030401020304"

4130

B.5 Client Application: Code Example Using TEEC Protocol

```
#define GP_ADMIN_ENVELOPE_CMD_ID 0x00C20000

/* Declared variables */

  static const TEEC_UUID uuidISDService = { 0x09a193b3, 0x688d, 0x567f, {0x88, 0xb4, 0x6c, 0xd7,
  0xda, 0x93, 0x22, 0x21 } };   /* the targeted Security Domain UUID to which the administrative command
  request is submitted */

TEEC_Result     libraryAdminFunction (

            uint8_t const * cmdReqBuffer,

            size_t              cmdReqSize,

            uint8_t *           cmdRespBuffer,

            size_t  *           p_cmdReqSize,

)

{

  TEEC_SharedMemory p0_InputParam, p1_OutputParam;

  TEEC_Context context;

  TEEC_Session session;

  TEEC_Result result;

  TEEC_Operation operation;

  /* Connect the TEE */

  result = TEEC_InitializeContext ( NULL, &context);

  if (result != TEEC_SUCCESS) goto cleanup0;

  /* Open a session with the targeted Security Domain */

  result = TEEC_OpenSession(&context,&session,&uuidISDService,TEEC_LOGIN_USER,NULL,NULL,NULL);

  if (result != TEEC_SUCCESS) goto cleanup1;

  /* Initialize the Shared memory buffers : P0 input param  & P1 output param */

  p0_InputParam.flags = TEEC_MEM_INPUT ;

  p0_ InputParam.size =  cmdReqSize;

  p0_inputParam.buffer = (uint8_t *) cmdReqBuffer;

  result = TEEC_RegisterSharedMemory(&context, &p0_InputParam);

  if (result != TEEC_SUCCESS) goto cleanup2;

  p1_OutputParam.flags = TEEC_MEM_OUTPUT ;

  p1_OutputParam.size =  *p_cmdReqSize;

  p1_OutputParam.buffer = cmdRespBuffer;

  result = TEEC_RegisterSharedMemory(&context, & p1_OutputParam);

  if (result != TEEC_SUCCESS) goto cleanup3;
```

```
   /*          Prepare Operation parameters  for the Invoke call        */
   memset(&operation, 0, sizeof(TEEC_Operation));
   operation.paramTypes =
TEEC_PARAM_TYPES(TEEC_MEMREF_PARTIAL_INPUT,  TEEC_MEMREF_PARTIAL_OUTPUT,  TEEC_NONE,
TEEC_NONE);
   operation.params[0].memref.parent = &p0_InputParam;
   operation.params[0].memref.offset = 0;
   operation.params[0].memref.size   = p0_InputParam.size;
   operation.params[1].memref.parent = &p1_OutputParam;
   operation.params[1].memref.offset = 0;
   operation.params[1].memref.size   = *p_cmdReqSize;
   /*          Invoke the Envelope command containing the administration
               command request/response buffers as operation parameters        */
   result = TEEC_InvokeCommand(&session, GP_ADMIN_ENVELOPE_CMD_ID, &operation);
    if (result != TEEC_SUCCESS) goto cleanup4;
      *p_cmdRespSize = operation.params[1].memref.size  ; /* output size may be less than the required
   size */
     return result;
cleanup4:
    TEEC_ReleaseSharedMemory(&p1_OutputParam);
cleanup3:
    TEEC_ReleaseSharedMemory(&p0_InputParam);
cleanup2:
    TEEC_CloseSession(&session);
cleanup1:
    TEEC_FinalizeContext(&context);
cleanup0:
    return result;
}
```

Annex C Index of Types

The following data types are commonly referred to in command and response operations encoding.